

A Beginner's Guide to Designing Embedded System Applications on Arm Cortex-M Microcontrollers

TEXTBOOK

Ariel Lutenberg, Pablo Gomez, Eric Pernia



Embedded Systems Design



A Beginner's Guide to Designing Embedded System Applications on Arm® Cortex®-M Microcontrollers

A Beginner's Guide to Designing Embedded System Applications on Arm® Cortex®-M Microcontrollers

ARIEL LUTENBERG
PABLO GOMEZ
ERIC PERNIA

Arm Education Media is an imprint of Arm Limited, 110 Fulbourn Road, Cambridge, CBI 9NJ, UK

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or any other information storage and retrieval system, without permission in writing from the publisher, except under the following conditions:

Permissions

- You may download this book in PDF format for personal, non-commercial use only.
- You may reprint or republish portions of the text for non-commercial, educational or research purposes but only if there is an attribution to Arm Education.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods and professional practices may become necessary.

Readers must always rely on their own experience and knowledge in evaluating and using any information, methods, project work, or experiments described herein. In using such information or methods, they should be mindful of their safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent permitted by law, the publisher and the authors, contributors, and editors shall not have any responsibility or liability for any losses, liabilities, claims, damages, costs or expenses resulting from or suffered in connection with the use of the information and materials set out in this textbook.

Such information and materials are protected by intellectual property rights around the world and are copyright © Arm Limited (or its affiliates). All rights are reserved. Any source code, models or other materials set out in this reference book should only be used for non-commercial, educational purposes (and/or subject to the terms of any license that is specified or otherwise provided by Arm). In no event shall purchasing this textbook be construed as granting a license to use any other Arm technology or know-how.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. For more information about Arm's trademarks, please visit <https://www.arm.com/company/policies/trademarks>.

Arm is committed to making the language we use inclusive, meaningful, and respectful. Our goal is to remove and replace non-inclusive language from our vocabulary to reflect our values and represent our global ecosystem.

Arm is working actively with our partners, standards bodies, and the wider ecosystem to adopt a consistent approach to the use of inclusive language and to eradicate and replace offensive terms. We recognise that this will take time. This book contains references to non-inclusive language; it will be updated with newer terms as those terms are agreed and ratified with the wider community.

Contact us at education@arm.com with questions or comments about this course. You can also report non-inclusive and offensive terminology usage in Arm content at terms@arm.com.

ISBN: 978-1-911531-42-5 (ePDF)
978-1-911531-41-8 (print)

Version: ePDF

For information on all Arm Education Media publications, visit our website at
<https://www.arm.com/resources/education/books>

To report errors or send feedback, please email edumedia@arm.com

Contents

Preface	xiii
Acknowledgments	xxi
Authors' Biographies	xxiii
Authors' Contributions	xxv
Book Organization	xxvi
How This Book Can Be Used for Teaching in Engineering Schools	xxix
Bill of Materials	xxxi
List of Figures	xxxiii
List of Tables	xli
<hr/>	
1 Introduction to Embedded Systems	1
1.1 Roadmap	2
1.1.1 What You Will Learn	2
1.1.2 Contents of This Chapter	2
1.2 Fundamentals of Embedded Systems	3
1.2.1 Main Components of Embedded Systems	3
1.2.2 First Implementation of the Smart Home System	5
1.2.3 Getting Ready to Program the First Implementation of the Smart Home System	10
Example 1.1: Activate the Alarm when Gas is Detected	11
Example 1.2: Activate the Alarm on Gas Presence or Over Temperature	17
Example 1.3: Keep the Alarm Active After Gas or Over Temperature Were Detected	20
Example 1.4: Secure the Alarm Deactivation by Means of a Code	24
Example 1.5: Block the System when Five Incorrect Codes are Entered	28
1.3 Under the Hood	33
1.3.1 Brief Introduction to the Cortex-M Processor Family and the NUCLEO Board	33
1.4 Case Study	39
1.4.1 Smart Door Locks	39
References	40

2	Fundamentals of Serial Communication	43
2.1	Roadmap	44
2.1.1	What You Will Learn	44
2.1.2	Review of Previous Chapter	44
2.1.3	Contents of This Chapter	44
2.2	Serial Communication Between a PC and the NUCLEO Board	45
2.2.1	Connect the Smart Home System to a PC	45
2.2.2	Modularization of a Program Into Functions	47
	Example 2.1: Monitor the Alarm State with a PC	60
	Example 2.2: Monitor Over Temperature and Gas Detection with a PC	66
	Example 2.3: Deactivate the Alarm Using the PC	69
	Example 2.4: Improve the Code Maintainability Using Arrays	71
	Example 2.5: Change the Alarm Turn Off Code Using the PC	76
2.3	Under the Hood	79
2.3.1	Basic Principles of Serial Communication	79
2.4	Case Study	82
2.4.1	Industrial Transmitter	82
	References	84
3	Time Management and Analog Signals	85
3.1	Roadmap	86
3.1.1	What You Will Learn	86
3.1.2	Review of Previous Chapters	86
3.1.3	Contents of This Chapter	86
3.2	Analog Signals Measurement with the NUCLEO Board	87
3.2.1	Connect Sensors, a Potentiometer, and a Buzzer to the Smart Home System	87
3.2.2	Test the Operation of the Sensors, the Potentiometer, and the Buzzer	92
	Example 3.1: Indicate which Sensor has Triggered the Alarm	95
	Example 3.2: Increase the Responsiveness of the Program	97
	Example 3.3: Activate the Over Temperature Alarm by Means of the Potentiometer	100
	Example 3.4: Usage of Functions to Compute the Temperature Value	105
	Example 3.5: Measure Temperature and Detect Gas using the Sensors	109

3.3	Under the Hood	115
3.3.1	Basic Principles of Analog to Digital Conversion	115
3.4	Case Study	123
3.4.1	Vineyard Frost Prevention	123
	References	124
4	Finite-State Machines and the Real-Time Clock	125
4.1	Roadmap	126
4.1.1	What You Will Learn	126
4.1.2	Review of Previous Chapters	126
4.1.3	Contents of This Chapter	126
4.2	Matrix Keypad Reading with the NUCLEO Board	127
4.2.1	Connect a Matrix Keypad and a Power Supply to the Smart Home System	127
4.2.2	Test the Operation of the Matrix Keypad and the RTC	132
	Example 4.1: Turn Off the Incorrect Code LED by Double-Pressing the Enter Button	133
	Example 4.2: Introduce the Usage of the Matrix Keypad	138
	Example 4.3: Implementation of Numeric Codes using the Matrix Keypad	144
	Example 4.4: Report Date and Time of Alarms to the PC Based on the RTC	149
4.3	Under the Hood	156
4.3.1	Graphical Representation of a Finite-State Machine	156
4.4	Case Study	160
4.4.1	Smart Door Locks	160
	References	171
5	Modularization Applied to Embedded Systems Programming	173
5.1	Roadmap	174
5.1.1	What You Will Learn	174
5.1.2	Review of Previous Chapters	174
5.1.3	Contents of This Chapter	174
5.2	Basic Principles of Modularization	174
5.2.1	Modularity Principle	174

5.3	Applying Modularization to the Program Code of the Smart Home System	176
5.3.1	Refactoring the Program Code of the Smart Home System	176
5.3.2	Detailed Implementation of the Refactored Code of the Smart Home System	183
5.4	Organizing the Modules of the Smart Home System into Different Files	210
5.4.1	Principles Followed to Organize the Modules into Files: Variables and Functions	210
5.4.2	Detailed Implementation of the Code of the Smart Home System in Different Files	216
	References	220
<hr/>		
6	LCD Displays and Communication between Integrated Circuits	221
6.1	Roadmap	222
6.1.1	What You Will Learn	222
6.1.2	Review of Previous Chapters	222
6.1.3	Contents of This Chapter	222
6.2	LCD Display Connection using GPIOs, I2C, and SPI Buses	223
6.2.1	Connect a Character LCD Display to the Smart Home System using GPIOs	223
6.2.2	Basic Principles of Character LCD Displays	227
	Example 6.1: Indicate Present Temperature, Gas Detection, and Alarm on the Display	233
	Example 6.2: Use of a 4-Bit Mode to Send Commands and Data to the Display	243
6.2.3	Connect a Character LCD Display to the Smart Home System using the I2C Bus	251
6.2.4	Fundamentals of the Inter-Integrated Circuit (I2C) Communication Protocol	255
	Example 6.3: Control the Character LCD Display by means of the I2C Bus	258
6.2.5	Connect a Graphical LCD Display to the Smart Home System using the SPI Bus	265
6.2.6	Basics Principles of Graphical LCD Displays	267
6.2.7	Fundamentals of the Serial Peripheral Interface (SPI) Communication Protocol	272
	Example 6.4: Control the Graphical LCD Display by means of the SPI Bus	274
	Example 6.5: Use of the Graphic Capabilities of the Graphical LCD Display	281

6.3	Under the Hood	290
6.3.1	Comparison between UART, SPI, and I2C	290
6.4	Case Study	292
6.4.1	LCD Usage in Mbed-Based Projects	292
	References	293
7	DC Motor Driving using Relays and Interrupts	295
7.1	Roadmap	296
7.1.1	What You Will Learn	296
7.1.2	Review of Previous Chapters	296
7.1.3	Contents of This Chapter	296
7.2	Motion Detection and DC Motor Control using Relays and Interrupts	297
7.2.1	Connect a DC Motor and a PIR Sensor to the Smart Home System	297
7.2.2	Fundamentals of Interrupt Service Routines	305
	Example 7.1: Control a DC Motor using Interrupts	307
	Example 7.2: Use a DC Motor to Open and Close a Gate	313
	Example 7.3: Use of a PIR Sensor to Detect Intruders	319
	Example 7.4: Use of the PIR Sensor as an Intruder Detection Alarm	327
7.3	Under the Hood	336
7.3.1	Basic Principles of a Relay Module	336
7.4	Case Study	338
7.4.1	Smart Street Lighting	338
	References	340
8	Advanced Time Management, Pulse-Width Modulation, Negative Feedback Control, and Audio Message Playback	341
8.1	Roadmap	342
8.1.1	What You Will Learn	342
8.1.2	Review of Previous Chapters	342
8.1.3	Contents of This Chapter	342
8.2	Analog Signal Generation with the NUCLEO Board	343
8.2.1	Connect an RGB LED, a Light Sensor, and an Audio Plug to the Smart Home System	343
8.2.2	Fundamentals of Timers, Pulse-Width Modulation, and Audio Message Playback	353

Example 8.1: Implementation of PWM to Control the Brightness of an RGB LED	356
Example 8.2: Implementation of PWM using the PwmOut Class	364
Example 8.3: Control the Siren and Strobe Light using PWM	367
Example 8.4: Adjustment of the Color of the Decorative RGB LED	370
Example 8.5: Use of the Light Sensor Reading to Control the RGB LED	373
Example 8.6: Playback of an Audio Message using the PWM Technique	379
8.3 Under the Hood	383
8.3.1 Fundamentals of Control Theory	383
8.4 Case Study	384
8.4.1 Smart City Bike Lights	384
References	385
<hr/>	
9 File Storage on SD Cards and Usage of Software Repositories	387
9.1 Roadmap	388
9.1.1 What You Will Learn	388
9.1.2 Review of Previous Chapters	388
9.1.3 Contents of This Chapter	388
9.2 File Storage with the NUCLEO Board	388
9.2.1 Connect an SD Card to the Smart Home System	388
9.2.2 A Filesystem to Control how Data is Stored and Retrieved	392
Example 9.1: Create a File with the Event Log on the SD Card	393
Example 9.2: Save a File on the SD Card with only New Events that were not Previously Saved	401
Example 9.3: Get the List of Event Log Files Stored on the SD Card	404
Example 9.4: Choose and Display One of the Event Log Files Stored on the SD Card	407
9.3 Under the Hood	413
9.3.1 Fundamentals of Software Repositories	413
9.4 Case Study	416
9.4.1 Repository Usage in Mbed-Based Projects	416
References	418

10	Bluetooth Low Energy Communication with a Smartphone	419
10.1	Roadmap	420
10.1.1	What You Will Learn	420
10.1.2	Review of Previous Chapters	420
10.1.3	Contents of This Chapter	420
10.2	Bluetooth Low Energy Communication between a Smartphone and the NUCLEO Board	420
10.2.1	Connect the Smart Home System to a Smartphone	420
10.2.2	Messages Exchanged with the Smartphone Application	424
	Example 10.1: Control the Gate Opening and Closing from a Smartphone	426
	Example 10.2: Report the Smart Home System State to a Smartphone	429
	Example 10.3: Implement the Smart Home System State Report Using Objects	432
	Example 10.4: Implement Non-Blocking Delays using Pointers and Interrupts	438
10.3	Under the Hood	444
10.3.1	Basic Principles of Bluetooth Low Energy Communication	444
10.4	Case Study	447
10.4.1	Wireless Bolt	447
	References	449
11	Embedded Web Server over a Wi-Fi Connection	451
11.1	Roadmap	452
11.1.1	What You Will Learn	452
11.1.2	Review of Previous Chapters	452
11.1.3	Contents of This Chapter	452
11.2	Serve a Web Page with the NUCLEO Board	452
11.2.1	Connect a Wi-Fi Module to the Smart Home System	452
11.2.2	Fundamentals of the Web Server to be Implemented	457
	Example 11.1: Implement the AT Command to Detect the Wi-Fi Module	464
	Example 11.2: Configure the Credentials to Connect to the Wi-Fi Access Point	471
	Example 11.3: Serve a Simple Web Page using the Wi-Fi Connection	480
	Example 11.4: Serve a Web Page that Shows the Smart Home System Information	486

11.3	Under the Hood	491
11.3.1	Basic Principles of Wi-Fi and TCP Connections	491
11.4	Case Study	492
11.4.1	Indoor Environment Monitoring	492
	References	494
12	Guide to Designing and Implementing an Embedded System Project	495
12.1	Roadmap	496
12.1.1	What You Will Learn	496
12.1.2	Review of Previous Chapters	496
12.1.3	Contents of This Chapter	496
12.2	Fundamentals of Embedded System Design and Implementation	497
12.2.1	Proposed Steps to Design and Implement an Embedded System Project	497
	Example 12.1: Select the Project that will be Implemented	498
	Example 12.2: Elicit Project Requirements and Use Cases	501
	Example 12.3: Design the Hardware	504
	Example 12.4: Design the Software	511
	Example 12.5: Implement the User Interface	519
	Example 12.6: Implement the Reading of the Sensors	526
	Example 12.7: Implement the Driving of the Actuators	530
	Example 12.8: Implement the Behavior of the System	533
	Example 12.9: Check the System Behavior	545
	Example 12.10: Develop the Documentation of the System	547
12.3	Final Words	549
12.3.1	The Projects to Come	549
	References	553
	Glossary of Abbreviations	555
	Index	561

Preface

In 2009, a small group of professors, teaching assistants, and students gathered together in order to create the Embedded Systems Laboratory at the School of Engineering of Universidad de Buenos Aires. The aim was to study and teach embedded system technologies, a topic that was not very developed at the university at that time. Pablo Gomez and Ariel Lutenberg were among this group.

During the following years, many undergraduate and graduate courses on embedded systems, as well as related courses, were organized by the group. Events on embedded systems were also held, and, in this way, a network of embedded systems professors was organized in Argentina.

In this context, an open hardware and software project named “Proyecto CIAA” (Computadora Industrial Abierta Argentina, Argentine Open Industrial Computer) was developed. Eric Pernia arrived as an expert on embedded systems programming.

Since then, many courses have been organized, including for people who had never programmed an embedded system before. After this experience, the idea of writing this book arose as a way to disseminate our ‘learn-by-doing’ teaching approach applied to embedded system more broadly.

This book follows our “learn-by-doing” approach, supported by hands-on activities. Basic ideas are explained and then demonstrated by means of examples that progressively introduce the fundamental concepts, techniques, and tools. In this way, a range of knowledge of electronics, informatics, and computers is introduced.

Theoretical concepts are kept to a minimum while still allowing students to properly understand the proposed solutions. Thus, the target audience of this book is beginners who have never before programmed embedded systems, or even had any prior knowledge of electronics.

Arm technology and C/C++ technology was chosen for this book because of the remarkable results we got during all our years of using them and the prevalence of Arm-based microcontrollers in embedded system design. The NUCLEO-F429ZI board was selected because of its ubiquity and low cost, and because it provides a broad set of interfaces that allows us to connect a wide variety of devices, as will be shown in the examples.

Through the examples, a smart home system that is shown in Figure 1 is gradually built. It is provided with an over temperature detector and a gas detector in order to implement a fire alarm. It also has a motion sensor that is used to detect intruders. If a fire or an intruder is detected, an alarm that is provided with a siren and a strobe light is turned on. To turn off the alarm, a code should be entered using the alarm control panel. If the code is incorrect, then the Incorrect code LED is turned on. Up to five codes can be entered before the system is blocked, which is indicated by means of the System blocked LED.

The alarm control panel has an LCD display, which is used to indicate the readings of the sensors and the status of the alarm (see Figure 1). It has also a slot for an SD memory card, where the events are stored, and the capability to play back a welcome audio message.

There is also a gate control panel, which allows the opening and closing of a gate, as shown in Figure 1. This panel is also provided with a light intensity control that is used to regulate the intensity of a decorative light. The intensity of this light is monitored using a light sensor, in order to control its brightness. The color of this light can also be changed using the alarm control panel.

The whole system can be monitored and configured using a PC by means of serial communication over a USB connection. Also, the most relevant information on the system can be accessed using an application on a smartphone, which is connected to the smart home system over a Bluetooth Low Energy connection. The application can also be used to open and close the gate.

It is also possible to monitor the smart home system using a smartphone or a PC by means of a web page that is served by the smart home system. In this case, the connection is made using the Wi-Fi protocol, as shown in Figure 1.

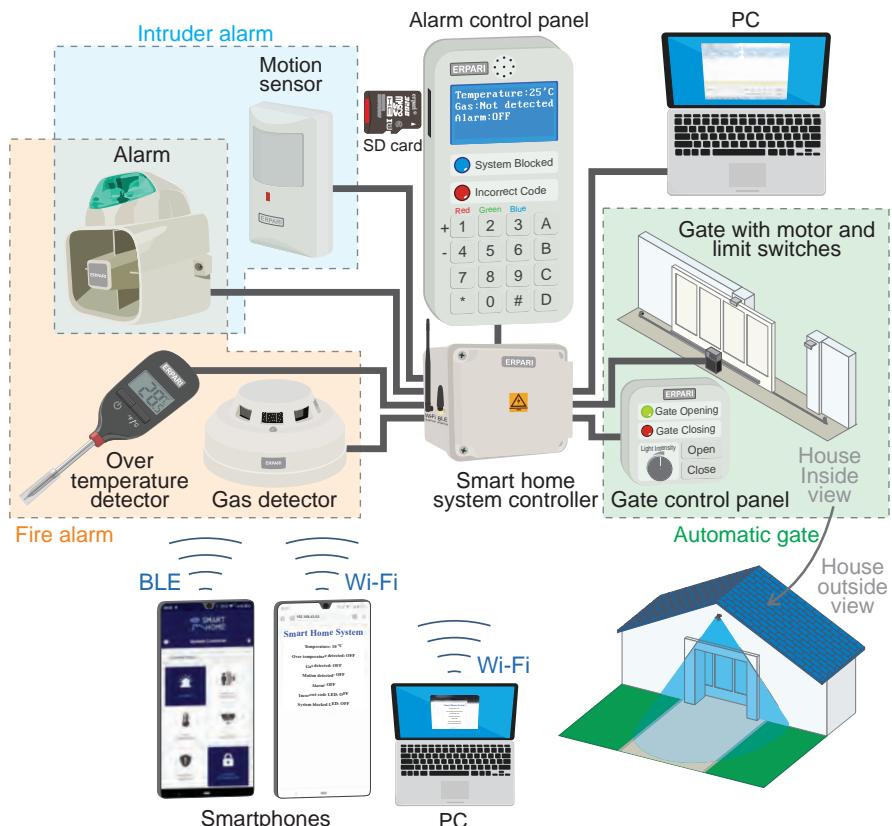


Figure 1 Smart home system that is built in this book.

Figure 2 shows how the NUCLEO board and a set of modules are used to implement the smart home system. Figure 3 shows how the pins in the ST Zio connectors of the NUCLEO board are occupied as the elements are gradually connected through the chapters. In this way, it is possible to present the reader with the difficulties that arise when a project starts to become bigger and bigger, and

show how these can be addressed by means of appropriate techniques. The proposed solutions are used to discuss how to implement efficient software design for embedded systems and how to make appropriate pin assignments for the elements.

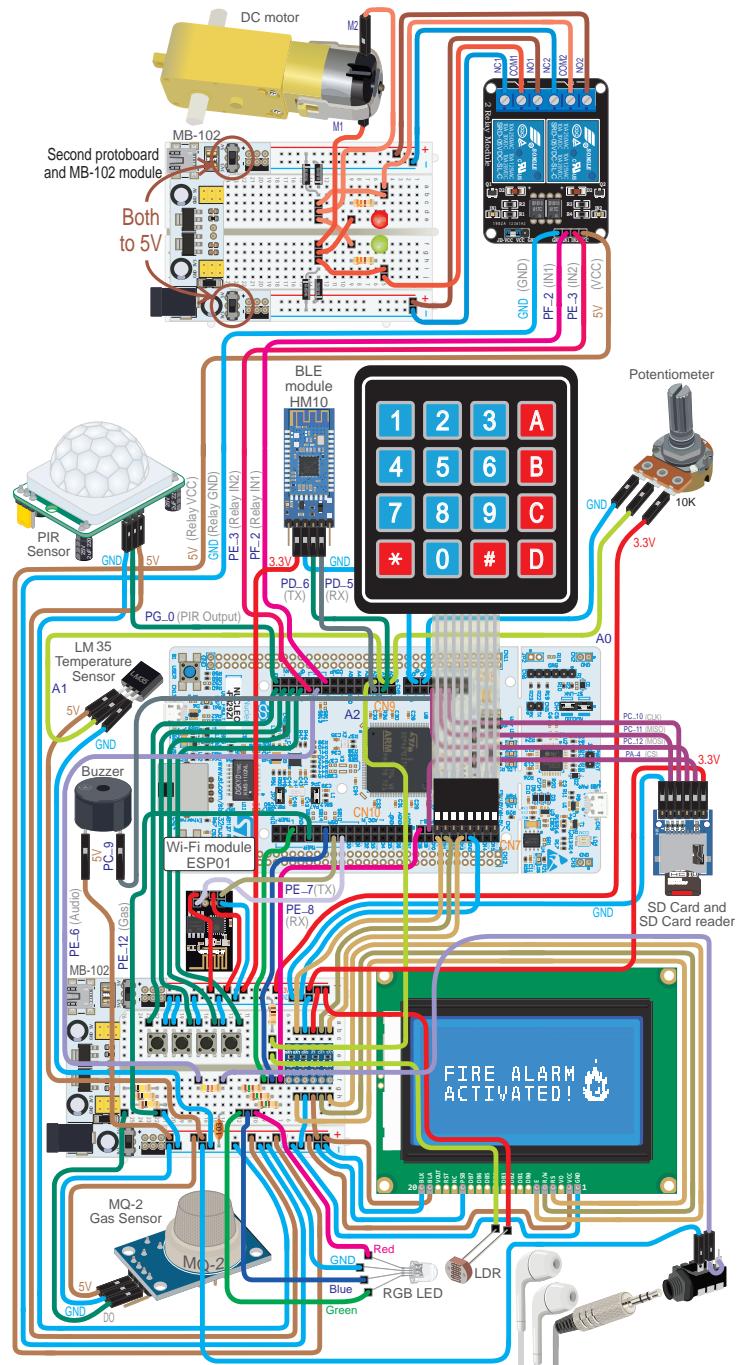


Figure 2 Diagram of the elements that are connected through the chapters.

In this way, Arm processor architecture and, in particular, the main peripherals of the Cortex-M4 processor are introduced, with the different elements (UARTs, timers, interrupts, etc.) that are required to efficiently implement the features of the smart home system. As the examples incorporate more and more functionality, the fundamentals of time management and multitasking operation in embedded systems are explained.

In addition, different methods for how an embedded system gets, receives, sends, manages, and stores data are shown, as well as many interfaces between the development board and the external devices being described and implemented. During this process, different embedded system designs for a given application are reviewed, and their features are compared.

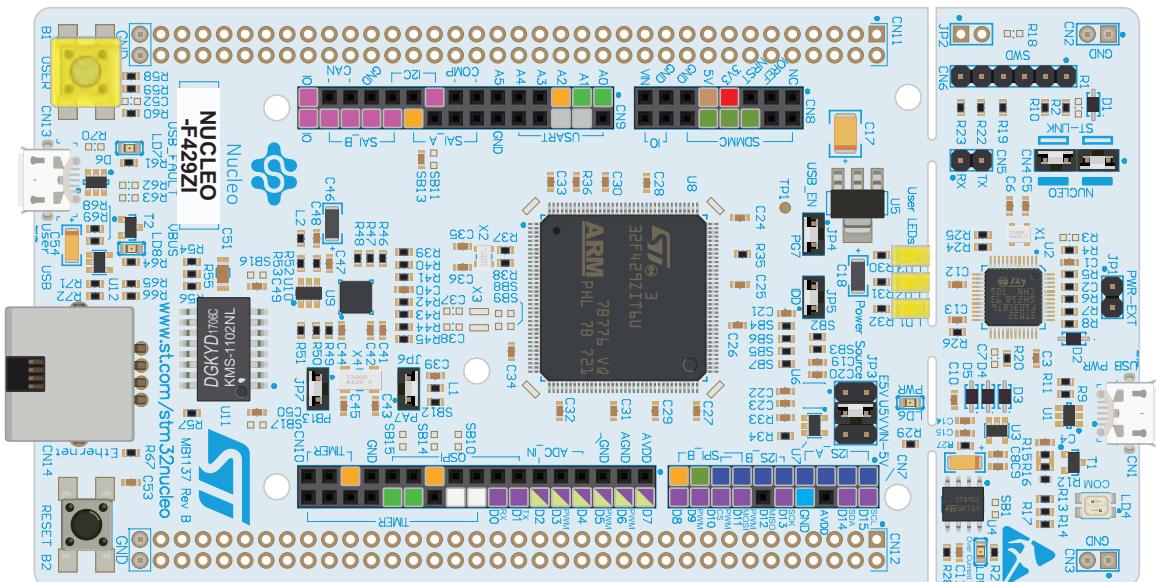
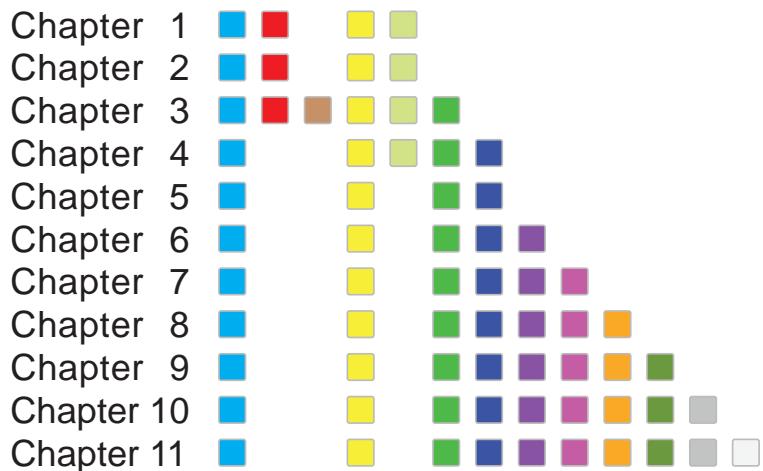


Figure 3 Diagram showing how the elements are gradually connected through each chapter.

In Figures 4 to 7, some of the user interfaces that are developed and used in the book are shown, which include a serial terminal on a PC, a character LCD display, an application for a smartphone, and a website that is accessed using a Wi-Fi connection and a web browser.

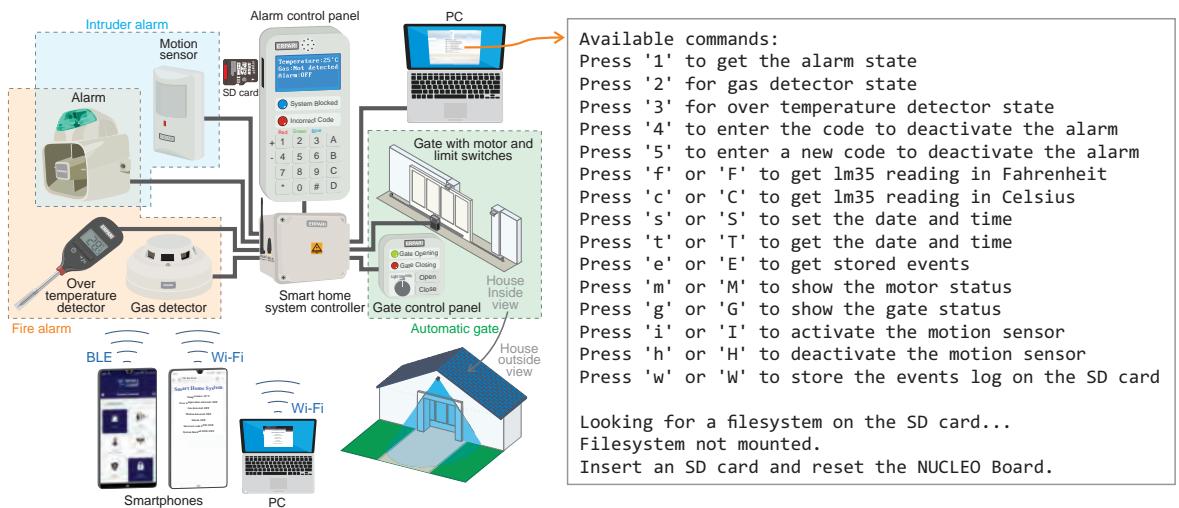


Figure 4 User interface implemented using a UART connection and a serial terminal running on a PC.

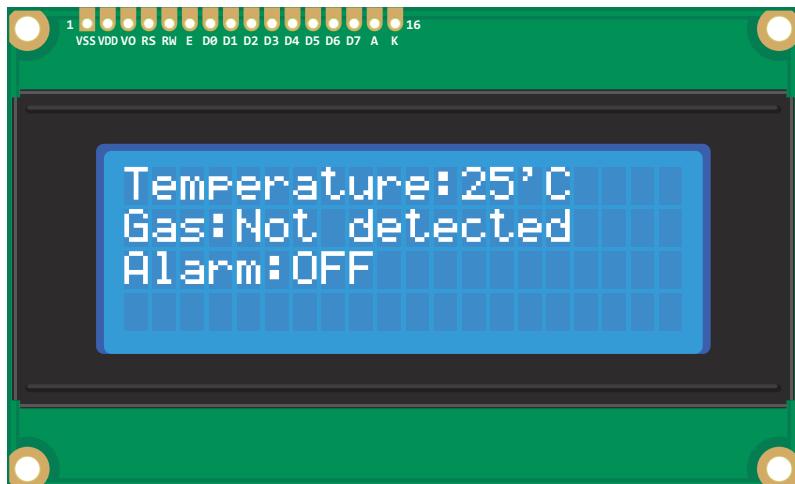


Figure 5 User interface implemented using I2C bus connection and a character LCD display.

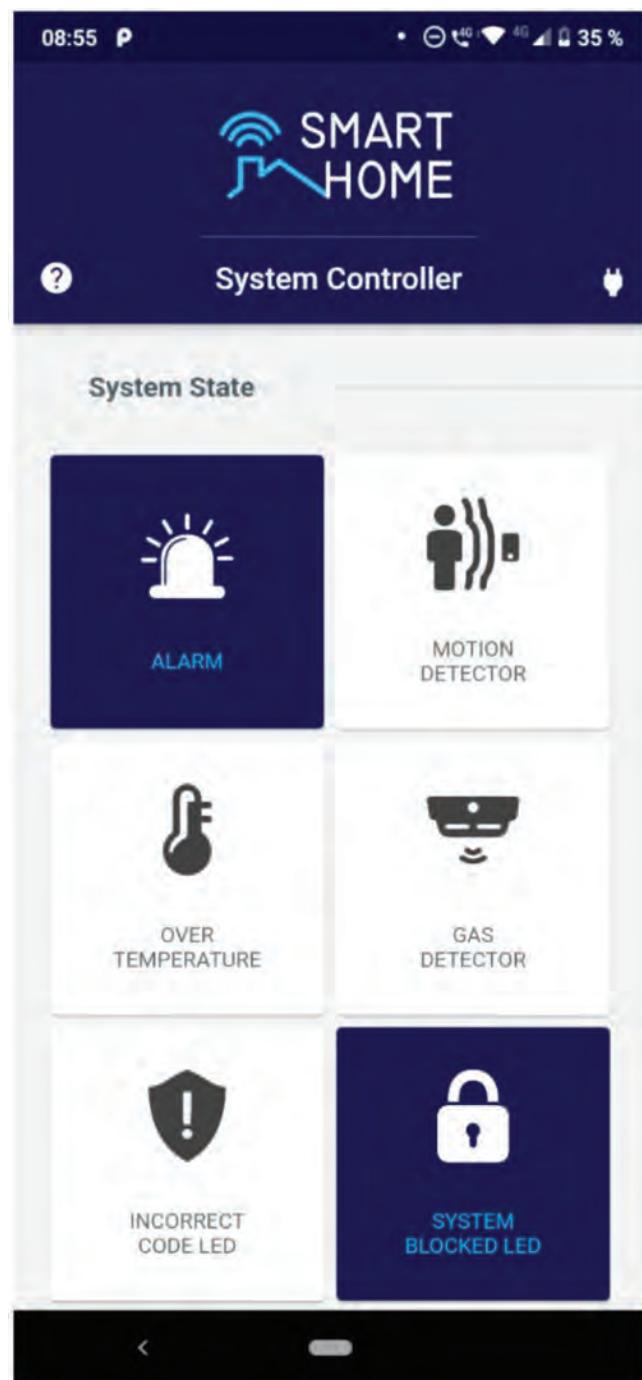


Figure 6 User interface implemented using a Bluetooth connection and a smartphone application.

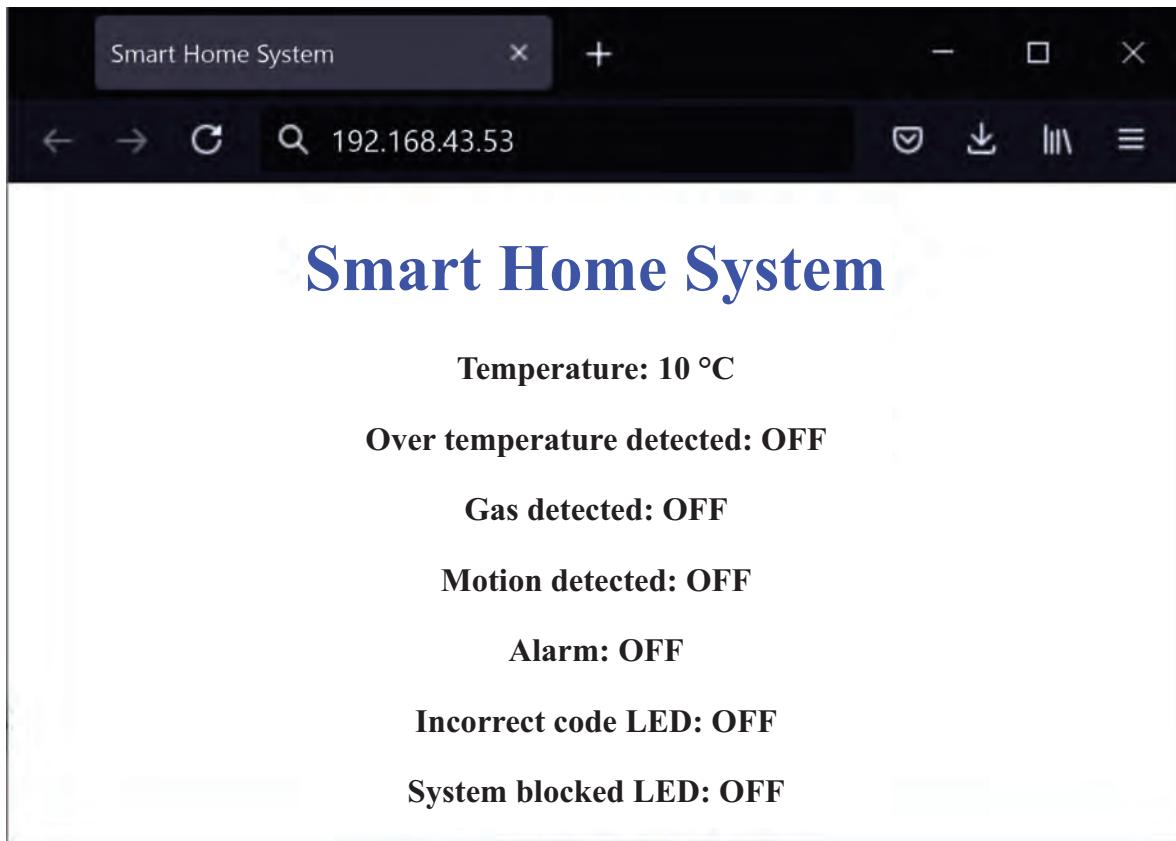


Figure 7 User interface implemented using a Wi-Fi connection and a web browser.

In this process, the Mbed™ OS 6 platform core generic software components, plus the HAL (Hardware Abstraction Layer) ports that allow Mbed to transparently run on microcontrollers from different manufacturers, are introduced.

In the final chapter, the main ideas are summarized by means of an irrigation system, shown in Figure 8, that is developed from scratch. In this way, a guide to designing and implementing an embedded system project is provided for the reader.

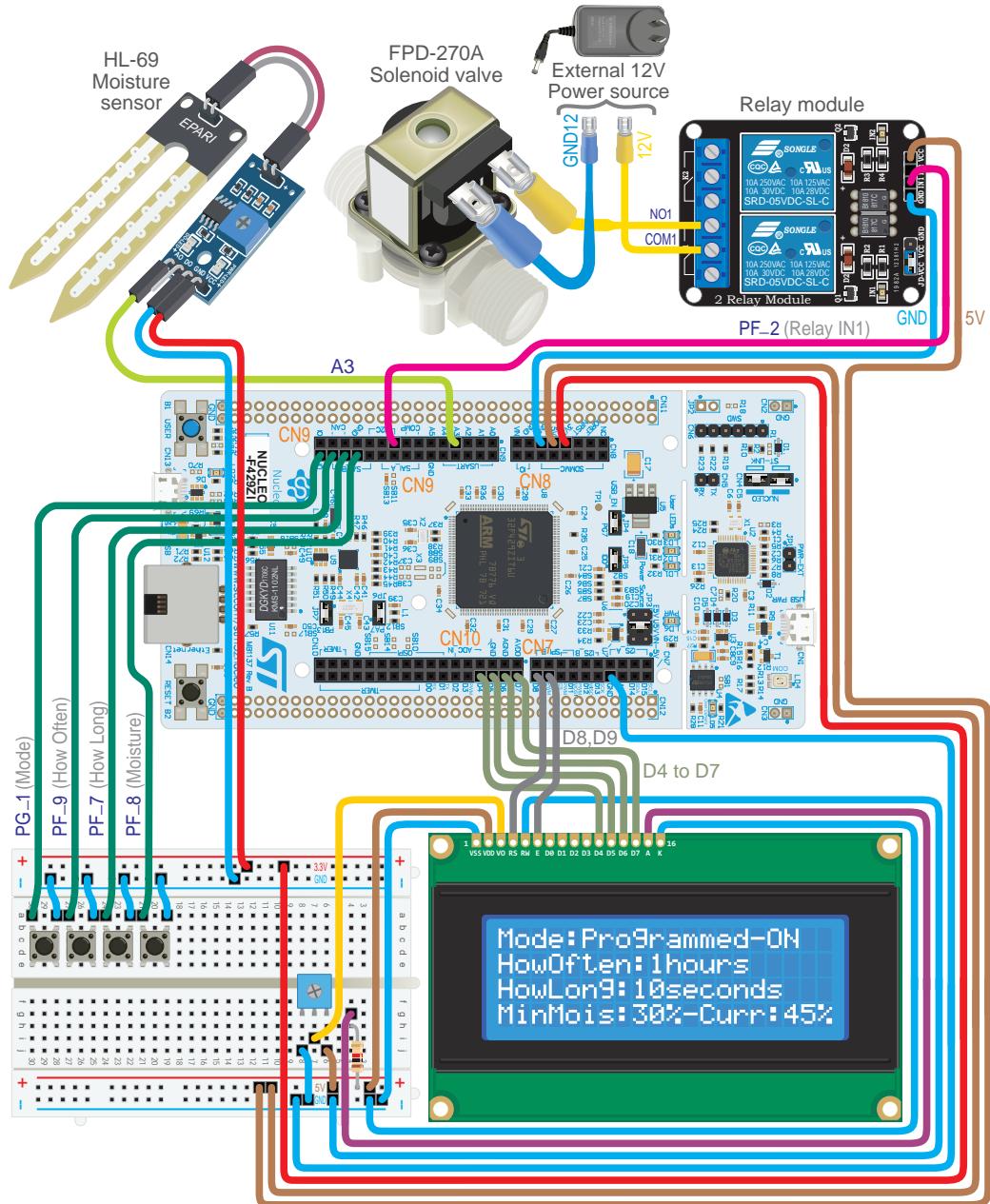


Figure 8 Home irrigation system that is implemented in Chapter 12.

To the best of our knowledge, there is no other book regarding Arm technology that follows the approach used in this book. We believe that the reader can benefit from it and combine learning skills with having fun.

We hope you enjoy the book!

Acknowledgments

We particularly thank our partners for supporting us during this year. Without their understanding, it wouldn't have been possible to write this book.

We would like to thank Arm Ltd for trusting us, especially Liz Warman, Director of Educational Content & Technology, for her support and guidance during the long year that it took us to write this book. The follow-up meetings with Liz, as well as her reviews, helped us to enhance the book and its narrative. We appreciate her candor and organization, being firm and kind in her suggestions.

We also thank our professors, colleagues, and students for teaching us so much. During recent years, the *Facultad de Ingeniería* (School of Engineering) of the *Universidad de Buenos Aires* (UBA) allowed us to develop a set of master's programs on embedded systems, the Internet of Things, and Artificial Intelligence that enriched our vision of how to teach using the learn-by-doing approach. Also, our research activities in the field of railway systems design, with the support of CONICET (the *National Scientific and Technical Research Council of Argentina*), prompted us to be up to date in a broad set of technologies, many of them included in this book. Colleagues of the *Departamento de Ciencia y Tecnología* (Science and Technology Department) of the *Universidad Nacional de Quilmes* (UNQ) shared their knowledge and advice.

The assistance of ACSE (*Civil Association for Research, Promotion and Development of Embedded Electronic Systems*) was extremely important for this publication. The immense number of activities organized by ACSE (symposiums, training for teachers and professors, calls for papers, courses for beginners, the *Argentine Open Industrial Computer*, etc.) allowed us to meet many people who enlightened us in different ways. ACSE also supported this book by providing most of the hardware used to implement and test the examples.

CADIEEL, the *Argentine Chamber of Electronic, Electromechanical and Lighting Industries*, promoted all the aforementioned activities. Especially noteworthy is the commitment to education and technological development of the small- and medium-sized companies that make up the chamber. CADIEEL has collaborated for years with the postgraduate programs and outreach activities mentioned above, as well as with this book in particular.

This book has significantly improved from the suggestions and comments of Carlos Pantelides. He is the one who has undoubtedly done the most detailed review of the book, and also gave us valuable ideas on how to address his suggestions on the book. We are very grateful to María Eloísa Tourret and Camila Belén Silva Sabarots, from UTN-FRBB, for developing the "Smart Home System App" that is used in this book, as well for their very useful comments about the chapter where Bluetooth Low Energy is explained. We are also thankful to Enrique Sergio Burgos, from UTN-FRP, for his very valuable help regarding the use of Doxygen for documentation and developing the corresponding example that is included in this book and setting up the continuous integration environment in GitLab. Juan Manuel Reta and Eduardo Filomena, from UNER, provided us with very enriching conversations, which we greatly appreciate, regarding how to use this book for teaching. The contribution of Martín Alejandro Ribeiro was very important in order to improve the *Brief Introduction to the Cortex-M*

Dedications

Processor Family and the NUCLEO Board section. Finally, the reading by Alejandro Salvatierra helped us to correct details, and the conversations with Juan Manuel Cruz (UBA, UTN-FRBA, ACSE) about how the examples in this book can be adapted to other boards and the revisions of the circuits used in this book by Adrián Laiuppa (UTN-FRBB) were very enlightening.

Dedications

Ariel dedicates this book to Sebastián, his first child who will be born in March 2022.

Pablo dedicates this book to Germán, his child who was born in October 2020.

Eric dedicates this book to the memory of his father, Jorge Sergio Pernia, who passed away in October 2021.

Authors' Biographies



Ariel Lutenberg is currently a full-time Professor at the School of Engineering of the University of Buenos Aires (UBA), Researcher at the National Council of Scientific and Technical Research (CONICET), and Director of the master's degree on the Internet of Things and the master's degree on Embedded Artificial Intelligence at the same University. He has supervised dozens of graduate, postgraduate, and doctoral students. He has published almost a hundred papers in international journals and conferences and done work for important Argentine and international companies and institutions. He started and led for many years Proyecto CIAA (Argentine Open Industrial Computer), where dozens of Argentinean universities, companies, and institutions together developed embedded computers, including their hardware and software.

He received an Electronic Engineering degree from the University of Buenos Aires in 2006, obtaining the prize "To the Best Graduates of Engineering Careers of Argentine Universities" from the National Academy of Engineering as the best graduate. In 2009, he obtained his diploma of Doctor of Engineering from the UBA, with honorable mention "Summa Cum Laude." In 2018, he won the INNOVAR Award of the Argentine Ministry of Science, Technology and Productive Innovation in the Researchers category and the INNOVAR Grand Prize of the Jury for the development of a remote monitoring system for automatic rail barriers.



Pablo Martín Gómez is full-time researcher at the School of Engineering of the University of Buenos Aires (UBA). He directs the master's program on Embedded Systems, coordinates the master's programs on the Internet of Things and Embedded Artificial Intelligence, and is teaching assistant of Acoustics at the same university. He received his diploma in Electronic Engineering in 2007 and finished his doctoral studies in 2015 with honorable mention "Summa Cum Laude." Pablo also studied at the National University of Lanús (UNLa), where he received a "University Technician on Sound and Recording" degree in 2008. He is editor of the "Acoustics and Audio" section of the Elektron journal published by the School of Engineering of UBA.

He has been working on embedded systems projects since 2003, designing products for companies and institutions in Argentina and the USA. In 2018, he won the INNOVAR Award of the Argentine Ministry of Science, Technology, and Productive Innovation in the Researchers category and the INNOVAR Grand Prize of the Jury for the development of a remote monitoring system for automatic rail barriers. He is author of several papers in the fields of embedded systems and acoustics in journals and for conferences. Teaching has a central role in his life. For almost 15 years he has been giving courses on acoustics, embedded system programming, protocols, real-time operating systems, and rapid prototyping for a wide audience, from novices to graduate students.



Eric Nicolás Pernia is currently a research Professor at the Science and Technology department (CyT) of the National University of Quilmes (UNQ) and a Field Application Engineer on Quectel wireless solutions. He has supervised several graduate and postgraduate students. He has published many papers in conferences, led the Proyecto CIAA (Argentine Open Industrial Computer) program for two years, and has broad experience in hardware, software, and firmware development for Argentine and international companies. He has a large portfolio of contributions in open-source hardware, firmware, and software, which is available on GitHub.

He received an Industrial Automation and Control Engineer degree from the National University of Quilmes (UNQ) in 2013, where he developed as a graduation project a free software application for PLC Ladder Diagram programming (IEC 61131-3 compliant), named IDE4PLC. In 2015, he obtained his diploma of Specialist on Embedded Systems from the University of Buenos Aires (UBA), where he created a Java SCJ implementation for CIAA Project boards as his graduation project. In 2018, he obtained his master's in Embedded Systems from the University of Buenos Aires (UBA), where he created an open-source, abstract, and highly portable embedded system programming library named sAPI as his graduation project.

Authors' Contributions

Ariel Lutenberg wrote most of the explanations in this book. This includes the preface, the introductory section of each chapter, the indications on how to connect the elements, the explanations of the program code examples, and the Under the Hood and Case Study sections. In particular, he selected which content to include in the introductory section of each chapter, as well as in the Under the Hood and Case Study sections. He also wrote the final versions of the program code of Chapter 6 and wrote all of Chapter 12 including the corresponding program code. In the other chapters, he revised the program code and made suggestions in order to simplify the implementation, to make it easier for the reader to understand the concepts introduced in each chapter. He also made important suggestions about how to rename the functions and variables in order to use more meaningful names. He contributed with many figures and in the revision of all the figures. He also started the relationship with Arm and was responsible for communicating with the editor and addressing the comments of all the reviewers. Perhaps his most important contributions were to propose the learn-by-doing approach followed in this book and to use the smart home system project as the common thread of the book, and the inclusion of a different project following a structured approach in the last chapter.

Pablo Martín Gomez proposed most of the examples that gradually built the smart home system and wrote most of the final versions of the program code of this book. This includes all the changes in the program code related to the reordering of the book chapters after concluding the first draft and the migration from Mbed OS 5 to Mbed OS 6, which was a very relevant contribution in order to update the preliminary versions of the program code. He also wrote many explanations of the examples (in particular in Chapters 7, 8, and 10, where he wrote all the code) and also contributed to and reviewed all the other explanations. He also tested all the examples on the NUCLEO board and kept the program code updated in the repositories using continuous integration and various automated tools that he set up. These tools also eased the process of maintaining, repairing, enhancing, and updating the program code, which grows in complexity from Chapters 1 to 11 as the smart home system increases its functionality. In this way, he identified many aspects to improve on the preliminary versions of the program code, making it easier to explain the fundamental concepts to the reader, as he made implementations clearer and more straightforward. He also carried out the migration to Mbed Online and Keil Studio Cloud when Arm suggested using those programs.

Eric Nicolás Pernía produced most of the figures in this book. He had a central role in writing the first versions of the program code of Chapters 1 to 5 and Chapter 9 and wrote the preliminary version of the program code in Chapters 6 and 11. He also reviewed many of the code samples and explanations. He suggested which hardware modules to use and made relevant contributions to include in the book the topics of modularization of a program into functions and files, finite-state machines, and different non-blocking delay techniques. He also contributed with a proposal about how to reorder the book chapters, which improved the reader experience.

Book Organization

This book is organized into five parts that take the reader, who has never before programmed embedded systems, from an introduction to embedded systems to the implementation of wireless communications and a set of proposed steps to develop embedded systems:

Part 1 – Introduction and Primary Tools

- Chapter 1: Introduction to Embedded Systems
- Chapter 2: Fundamentals of Serial Communication
- Chapter 3: Time Management and Analog Signals

Part 2 – Advanced Concepts in Embedded Systems

- Chapter 4: Finite-State Machines and the Real-Time Clock
- Chapter 5: Modularization Applied to Embedded Systems Programming
- Chapter 6: LCD Displays and Communication between Integrated Circuits

Part 3 – Solutions Based on the Processor Peripherals and Filesystems

- Chapter 7: DC Motor Driving using Relays and Interrupts
- Chapter 8: Advanced Time Management, Pulse-Width Modulation, Negative Feedback Control, and Audio Message Playback
- Chapter 9: File Storage on SD Cards and Usage of Software Repositories

Part 4 – Wireless Communications in Embedded Systems

- Chapter 10: Bluetooth Low Energy Communication with a Smartphone
- Chapter 11: Embedded Web Server over a Wi-Fi Connection

Part 5 – Proposed Steps to Develop Embedded Systems

- Chapter 12: Guide to Designing and Implementing an Embedded System Project

Chapter 1 addresses a general introduction to embedded systems and the smart home system project using LEDs and buttons. The basic concepts of embedded systems communications are tackled in Chapter 2, by means of connecting the NUCLEO board with a PC using serial communication.

Chapter 3 explains how to manage time intervals and analog signals, and some sensors and connectors are connected to the board.

Chapter 4 explains how to organize complex programs using finite-state machines. Modularization in files, which allows organization of large and complex programs, is introduced in Chapter 5. In Chapter 6, a character-based display and a graphical LCD display are connected and controlled using the same code for four different interfaces: 4/8-bit parallel interface, I2C bus, and SPI bus. In this way, the concept of a hardware abstraction layer is introduced.

In Part 3, more advanced concepts are explained, such as the use of interrupts in Chapter 7; microcontroller timers, pulse-width modulation, negative feedback control, and audio message playback in Chapter 8; and filesystems and software repositories in Chapter 9.

In Chapter 10, the smart home system is communicated with via a smartphone using a Bluetooth Low Energy connection, and an embedded web server is developed in Chapter 11 in order to provide web pages to a computer by means of a Wi-Fi network.

Lastly, in Chapter 12, a complete example of a different embedded system development is presented in order to show the reader in detail how a project can be addressed from beginning to end.

In this way, a multitude of useful concepts, techniques, and tools are presented in just twelve chapters, starting from the level of a complete beginner and building up to relatively complex and advanced topics.

More advanced concepts, such as integration of RTOS (Real-Time Operating Systems) with an embedded system, learning of the lower-level details, USB protocol, and applications related to the Internet of Things and Artificial Intelligence, are not included in this book.

Each chapter starts with a Roadmap section, where the corresponding learning objectives are described in “What You Will Learn,” followed by “Review of Previous Chapters,” and then “Contents of This Chapter.”

Usually, this is followed by a section that explains how to connect the devices that are incorporated into the smart home system setup, as well as the fundamentals of their operation.

This is followed by Examples that are used to explain the practical use of the elements. The Examples are tackled in seven steps:

- Objective of the Example
- Summary of the Expected Behavior
- Test the Proposed Solution on the Board

Book Organization

- Discussion of the Proposed Solution
- Implementation of the Proposed Solution
- Proposed Exercises
- Answers to the Exercises.

At the end of each chapter, there is an Under the Hood section, where more advanced concepts are introduced, to motivate the reader to explore beyond the limits of this book.

Most of the chapters conclude with a Case Study section, whose aim is to illustrate to the target audience of this book (beginners who have never programmed embedded systems before and have little or no prior knowledge of electronics) how the learned technologies are used in real-world applications.

How This Book Can Be Used for Teaching in Engineering Schools

This book was planned considering that in many countries, educational institutions hold classes for about twelve weeks. Given that this book is organized into twelve chapters, one chapter can be addressed to the students every week by means of the organization shown in Table 1 (in the case of two 90-minute or 120-minute lessons per week) or in Table 2 (if there is one 180-minute lesson per week).

In both cases, students are requested to connect and test a given setup every week, prior to the lessons, following the steps indicated in the first column of Table 1 and Table 2. This homework is estimated to take the students about one or two hours every week and is aimed at developing curiosity and enthusiasm for learning about how those systems work.

Table 1 and Table 2 show how the lessons can be organized. It is important to note that the setups that are connected and tested prior to the lessons every week are used in the examples in that week without the need for any change in the setup. This increases productivity during the lessons, because in this way all the lesson time is used to introduce new ideas and to test and discuss the examples in each chapter.

The activities each week are closed by the discussion of the Under the Hood and Case Study sections of each chapter, which can lead to different types of activities, for example asking the students to:

- look on the internet for more information about the topics discussed in the Under the Hood section;
- look on the internet for more case studies where the topics introduced in the chapter are applied.

For several reasons, Chapters 5, 6, and 12 are organized in a slightly different way than the other chapters. In these particular cases, Table 1 and Table 2 can be considered one option among many other possibilities.

Lastly, it is suggested to ask the students to implement a short project every two or three weeks, applying the concepts introduced in the lessons. Also, it is recommended to ask the students to implement a final project at the end of the course, following the process and techniques that are introduced in Chapter 12.

Table 1 Organization in the case of two 90-minute or 120-minute lessons per week.

Week	Pre-lesson activities	First lesson	Second lesson
1	1.2.3	1.1.1, 1.1.2, 1.2.1, 1.2.2 Examples 1.1, 1.2 and 1.3	Examples 1.4 and 1.5, 1.3.1, 1.4.1
2	2.2.1	2.1.1, 2.1.2, 2.1.3, 2.2.2, Examples 2.1, 2.2 and 2.3	Examples 2.4 and 2.5, 2.3.1, 2.4.1
3	3.2.1, 3.2.2	3.1.1, 3.1.2, 3.1.3, Examples 3.1, 3.2 and 3.3	Examples 3.4 and 3.5, 3.3.1, 3.4.1

How This Book Can Be Used for Teaching in Engineering Schools

Week	Pre-lesson activities	First lesson	Second lesson
4	4.2.1, 4.2.2	4.1.1, 4.1.2, 4.1.3 Examples 4.1 and 4.2	Examples 4.3 and 4.4 4.3.1, 4.4.1
5	-	5.1.1, 5.1.2, 5.1.3, 5.2.1	5.3.1, 5.3.2, 5.4.1, 5.4.2
6	6.2.1, 6.2.3, 6.2.4	6.1.1, 6.1.2, 6.1.3, 6.2.2 Examples 6.1 and 6.2	Examples 6.3, 6.4, and 6.5 6.3.1, 6.4.1
7	7.2.1	7.1.1, 7.1.2, 7.1.3, 7.2.2 Examples 7.1 and 7.2	Examples 7.3 and 7.4 7.3.1, 7.4.1
8	8.2.1	8.1.1, 8.1.2, 8.1.3, 8.2.2 Examples 8.1, 8.2, 8.3, and 8.4	Examples 8.5 and 8.6 8.3.1, 8.4.1
9	9.2.1	9.1.1, 9.1.2, 9.1.3, 9.2.2 Examples 9.1 and 9.2	Examples 9.3 and 9.4 9.3.1, 9.4.1
10	10.2.1	10.1.1, 10.1.2, 10.1.3, 10.2.2 Examples 10.1 and 10.2	Examples 10.3 and 10.4 10.3.1, 10.4.1
11	11.2.1	11.1.1, 11.1.2, 11.1.3, 11.2.2 Examples 11.1 and 11.2	Examples 11.3 and 11.4 11.3.1, 11.4.1
12	-	12.1.1, 12.1.2, 12.1.3, 12.2.1 Examples 12.1 to 12.4	Examples 12.5 to 12.10, 12.3.1

Table 2 Organization in the case of one 180-minute lesson per week.

Week	Pre-lesson activities	First part of the lesson	Second part of the lesson	Third part of the lesson
1	1.2.3	1.1.1, 1.1.2, 1.2.1, 1.2.2 Examples 1.1, 1.2, and 1.3	Examples 1.4 and 1.5	1.3.1, 1.4.1
2	2.2.1	2.1.1, 2.1.2, 2.1.3, 2.2.2 Examples 2.1, 2.2 and 2.3	Examples 2.4 and 2.5	2.3.1, 2.4.1
3	3.2.1, 3.2.2	3.1.1, 3.1.2, 3.1.3 Examples 3.1, 3.2 and 3.3	Examples 3.4 and 3.5	3.3.1, 3.4.1
4	4.2.1, 4.2.2	4.1.1, 4.1.2, 4.1.3 Examples 4.1 and 4.2	Examples 4.3 and 4.4	4.3.1, 4.4.1
5	-	5.1.1, 5.1.2, 5.1.3, 5.2.1	5.3.1, 5.3.2	5.4.1, 5.4.2
6	6.2.1, 6.2.3, 6.2.4	6.1.1, 6.1.2, 6.1.3, 6.2.2 Examples 6.1 and 6.2	Examples 6.3, 6.4 and 6.5	6.3.1, 6.4.1
7	7.2.1	7.1.1, 7.1.2, 7.1.3, 7.2.2 Examples 7.1 and 7.2	Examples 7.3 and 7.4	7.3.1, 7.4.1
8	8.2.1	8.1.1, 8.1.2, 8.1.3, 8.2.2 Examples 8.1, 8.2, 8.3, and 8.4	Examples 8.5 and 8.6	8.3.1, 8.4.1
9	9.2.1	9.1.1, 9.1.2, 9.1.3, 9.2.2 Examples 9.1 and 9.2	Examples 9.3 and 9.4	9.3.1, 9.4.1
10	10.2.1	10.1.1, 10.1.2, 10.1.3, 10.2.2 Examples 10.1 and 10.2	Examples 10.3 and 10.4	10.3.1, 10.4.1
11	11.2.1	11.1.1, 11.1.2, 11.1.3, 11.2.2 Examples 11.1 and 11.2	Examples 11.3 and 11.4	11.3.1, 11.4.1
12	-	12.1.1, 12.1.2, 12.1.3, 12.2.1 Examples 12.1 to 12.4	Examples 12.5 to 12.8	Examples 12.9 and 12.10, 12.3.1

Bill of Materials

The materials that are used in this book from Chapter 1 to Chapter 11 are summarized in Table 3. The aim is to help the reader to get all the components needed to implement the setups used in each chapter.

There is an additional bill of materials in Table 4, which lists the materials used in Chapter 12. These are not essential because the concepts that are introduced in this chapter can be learned without connecting all of the elements. For example, if the solenoid valve is not connected, the reader will still be able to see the relay module switching on and off, and the moisture sensor can be mocked using a potentiometer.

In addition, many tools are included in the listing shown in Table 5 because they may be useful.

Table 3 Bill of materials (BOM) used in Chapters 1 to 11.

Chapter	Component/Module	Quantity
1	NUCLEO-F429ZI	1
	Jumper wires (male–male)	80
	USB–micro USB cable	1
	Breadboard	2
	Tactile switches	10
	(The same elements are used as in Chapter 1.)	-
	LM35 temperature sensor	1
	10 kΩ potentiometer	2
	MQ-2 gas sensor	1
	5V buzzer	1
3	Resistors: 100 Ω, 150 Ω, 330 Ω, 1 kΩ, 10 kΩ, 47 kΩ, 100 kΩ (100 kΩ are used in this chapter and the other resistors in different chapters. See note below.)	10 of each value
	Jumper wires (male–female)	80
	Matrix keypad (4 × 4)	1
	90-degrees 2.54 mm (.1") pitch pin header	40
	MB102 breadboard power supply (One is used in this chapter and the other in Chapter 7.)	2
4	USB–mini-USB cable	2
5	BC548C NPN transistor (One is used in this chapter and the other in Chapter 8.)	2
	Character-based LCD display 20 × 4, based on HD44780	1
6	I2C PCF8574 expander	1
	10 kΩ trimpot	1
	Bidirectional logic level converter	1
	Jumper wires (female–female)	40
	Graphic LCD display, based on ST7920	1

Bill of Materials

Chapter	Component/Module	Quantity
7	3 mm red LED	1
	3 mm green LED	1
	5 V DC motor	1
	HC-SR501 PIR sensor module	1
	Relay module	1
	1N5819 diodes	4
	LED RGB	1
	LDR	1
	10 nF ceramic capacitor	1
	Female audio jack 3.5 mm (for PCB)	1
8	SD card module	1
9	Micro SD memory card	1
10	Bluetooth HM-10 module	1
11	Wi-Fi ESP8266 ESP-01 module	1



NOTE: The resistors are used as follows: Chapter 3: $3 \times 100 \text{ k}\Omega$; Chapter 5: $1 \times 1 \text{ k}\Omega$; Chapter 6: $1 \times 1 \text{ k}\Omega$; Chapter 7: $2 \times 330 \Omega$; Chapter 8: $3 \times 150 \Omega$, $1 \times 10 \text{ k}\Omega$, $1 \times 47 \text{ k}\Omega$, $1 \times 1 \text{k } \Omega$, and $1 \times 100 \Omega$.

Table 4 Bill of materials (BOM) used in Chapter 12.

Chapter	Component/Module	Quantity
12	Moisture sensor	1
	12 V solenoid valve	1
	12 V \times 1 A power supply	1

Table 5 Tools for soldering, measuring, and setting the modules.

Chapter	Quantity
Soldering iron	1
Wire solder	100 grams
Plier	1
Multimeter	1
Small screwdriver	1

List of Figures

Figure 1	Smart home system that is built in this book.	xiv
Figure 2	Diagram of the elements that are connected through the chapters.	xv
Figure 3	Diagram showing how the elements are gradually connected through each chapter.	xvi
Figure 4	User interface implemented using a UART connection and a serial terminal running on a PC.	xvii
Figure 5	User interface implemented using I2C bus connection and a character LCD display.	xvii
Figure 6	User interface implemented using a Bluetooth connection and a smartphone application.	xviii
Figure 7	User interface implemented using a Wi-Fi connection and a web browser.	xix
Figure 8	Home irrigation system that is implemented in Chapter 12.	xx
Figure 1.1	Four different block diagram representations of a smart home system.	3
Figure 1.2	Diagram of the proposed smart home system.	5
Figure 1.3	The smart home system that is implemented in this chapter.	6
Figure 1.4	The NUCLEO-F429ZI board used in this book.	6
Figure 1.5	The elements of the smart home system introduced in Figure 1.3 with their corresponding components.	7
Figure 1.6	Detail of the connection made in the CN8 ST Zio connector of the NUCLEO board.	9
Figure 1.7	Detail of the connections made in the CN10 ST Zio connector of the NUCLEO board.	9
Figure 1.8	How to connect the NUCLEO board to a PC.	11
Figure 1.9	Structure of the proposed solution.	12
Figure 1.10	Conceptual diagram of how to connect a button to the NUCLEO board using pull-down and pull-up resistors.	15
Figure 1.11	Main parts of the program of Example 1.2.	18
Figure 1.12	Details of the blocks that make up the repetitive block of Example 1.2.	18
Figure 1.13	Main parts of the program of Example 1.3.	22
Figure 1.14	Details of the blocks that compose the repetitive block of Example 1.3.	22
Figure 1.15	Main parts of the program of Example 1.4.	25
Figure 1.16	Details of the blocks that make up the repetitive block of Example 1.4.	26
Figure 1.17	Main parts of the program of Example 1.5.	29
Figure 1.18	Details of the blocks that make up the repetitive block of Example 1.5.	31

List of Figures

Figure 1.19	Simplified diagram of the Cortex processor family.	33
Figure 1.20	Simplified diagram of the Cortex M0, M3, and M4 processors, and details of the corresponding cores.	34
Figure 1.21	Arm Cortex M0, M3, and M4 Instruction Set Architecture (ISA).	35
Figure 1.22	STM32F429ZI block diagram made using information available from [9].	36
Figure 1.23	ST Zio connectors of the NUCLEO-F429ZI board.	37
Figure 1.24	Hierarchy of different elements introduced in this chapter.	38
Figure 1.25	“Smart door locks” built with Mbed contains elements introduced in this chapter.	39
Figure 2.1	The smart home system is now connected to a PC.	45
Figure 2.2	Website with the program documentation generated with Doxygen.	57
Figure 2.3	Detailed description of functions and variables of the program that is available on the website (Part 1/3).	57
Figure 2.4	Detailed description of functions and variables of the program that is available on the website (Part 2/3).	58
Figure 2.5	Detailed description of functions and variables of the program that is available on the website (Part 3/3).	58
Figure 2.6	Interactive view of the code.	59
Figure 2.7	DigitalIn class reference.	59
Figure 2.8	Website of Mbed with detailed information about DigitalIn and the whole Application Programming Interface.	60
Figure 2.9	Details of the function uartTask() used in this proposed solution to Example 2.1.	62
Figure 2.10	Details of the function uartTask() used in the proposed solution to Example 2.2.	67
Figure 2.11	Details of the ‘4’ of the function uartTask().	73
Figure 2.12	Basic setup for a serial communication between two devices.	79
Figure 2.13	Basic sequence of a serial communication between two devices.	80
Figure 2.14	The bits transmitted or received by the NUCLEO board UART can be seen by connecting an oscilloscope or a logic analyzer on CN5. These bits do not correspond to the USB protocol.	82
Figure 2.15	Top, the smart home system. Bottom, a diagram of the “Industrial transmitter.”	83
Figure 3.1	The smart home system is now connected to a temperature sensor, a gas detector, a potentiometer, and a buzzer.	87
Figure 3.2	A typical potentiometer and its corresponding electrical diagram.	88
Figure 3.3	Diagram of the connection of the potentiometer to the NUCLEO board.	89
Figure 3.4	The LM35 temperature sensor in a TO-92 package.	89
Figure 3.5	Basic setup for the LM35 temperature sensor.	90

Figure 3.6	Diagram of the connection of the LM35 to the NUCLEO board.	90
Figure 3.7	Diagram of the connection of the MQ-2 to the NUCLEO board.	91
Figure 3.8	Diagram of the connection of the buzzer to the NUCLEO board.	92
Figure 3.9	Diagram of the connection of the MQ-2 gas sensor module.	94
Figure 3.10	Simplified diagram of a Successive Approximation Register analog to digital converter.	115
Figure 3.11	“Vineyard frost prevention” built with Mbed contains elements introduced in this chapter.	123
Figure 4.1	The smart home system is now connected to a matrix keypad.	127
Figure 4.2	The smart home system is now connected to a matrix keypad.	128
Figure 4.3	Detail showing how to prepare the matrix keypad connector using a pin header.	129
Figure 4.4	Diagram of the connections of the matrix keypad.	129
Figure 4.5	Diagram of the connections between the matrix keypad and the NUCLEO board.	130
Figure 4.6	Voltage signal over time for a given button, including typical glitches and bounces.	131
Figure 4.7	Diagram of the MB102 module.	132
Figure 4.8	Voltage signal over time for a given button as it is pressed or released.	157
Figure 4.9	Diagram of the FSM implemented in Example 4.1.	158
Figure 4.10	Diagram of the FSM implemented in Example 4.2.	160
Figure 4.11	Smart door lock built with Mbed contains a keypad similar to the one introduced in this chapter.	161
Figure 5.1	Diagram of the first attempt to modularize the smart home system program.	177
Figure 5.2	Diagram of the modules used in the smart home system program.	178
Figure 5.3	Diagram of the relationships between the modules used in the smart home system.	183
Figure 5.4	Diagram showing how the definitions of the event_log module were made.	185
Figure 5.5	Diagram of the voltage through the buzzer pins when PE_10 is set to 0 V and 3.3 V.	187
Figure 5.6	Diagram of the circuit that can be used to completely turn on and off the buzzer.	187
Figure 5.7	Diagram of modularization in C/C++ using header files.	217
Figure 5.8	File organization proposed for the smart home system code.	219
Figure 6.1	The smart home system is now connected to an LCD display.	223
Figure 6.2	The smart home system connected to the character LCD display using GPIOs.	224

List of Figures

Figure 6.3	Diagram of the connections between the character LCD display and the NUCLEO board using GPIOs.	225
Figure 6.4	CN11 and CN12 headers of the NUCLEO-F429ZI board scheme made using information available from [4].	226
Figure 6.5	Addresses corresponding to each of the positions of a 20×4 LCD character display.	229
Figure 6.6	Addresses corresponding to each of the positions of a 16×2 LCD character display.	229
Figure 6.7	Addresses corresponding to each of the positions of an 8×2 LCD character display.	229
Figure 6.8	Addresses corresponding to each of the positions of an 8×1 LCD character display.	229
Figure 6.9	An 8×2 LCD character display where a left shift has been applied once.	230
Figure 6.10	Transfer timing sequence of writing instructions when an 8-bit interface is configured.	231
Figure 6.11	Initialization procedure of the graphic display when an 8-bit interface is used.	232
Figure 6.12	Position of the strings that are placed in the character LCD display.	235
Figure 6.13	The smart home system connected to the character LCD display using 4-bit mode interface.	244
Figure 6.14	Transfer timing sequence of writing instructions when a 4-bit interface is configured.	245
Figure 6.15	Initialization procedure of the graphic display when a 4-bit interface is used.	246
Figure 6.16	The smart home system is now connected to the character LCD display using the I ₂ C bus.	252
Figure 6.17	Diagram of the connections between the character LCD display and the NUCLEO board using the I ₂ C bus.	253
Figure 6.18	Simplified block diagram of the PCF8574 together with its connections to the LCD display.	254
Figure 6.19	Examples of other PCF8574 modules that do not include the potentiometer and the resistors.	255
Figure 6.20	Example of a typical I ₂ C bus connection between many devices.	255
Figure 6.21	Example of I ₂ C bus start and stop conditions.	256
Figure 6.22	Example of a typical I ₂ C bus address message.	256
Figure 6.23	Example of a typical I ₂ C bus communication.	257
Figure 6.24	Example of a writing operation to the PCF8574.	257
Figure 6.25	The smart home system is now connected to the graphical LCD display using the SPI bus.	265

Figure 6.26	Diagram of the connections between the graphical LCD display and the NUCLEO board using the SPI bus.	266
Figure 6.27	Addresses corresponding to each of the positions of a graphical LCD display in character mode.	267
Figure 6.28	Transfer timing sequence of the graphical LCD display when the serial mode is configured.	269
Figure 6.29	Simplified block diagram of a ST7920 and two ST7921 used to drive a 32 × 256 dot matrix LCD panel.	270
Figure 6.30	Simplified block diagram of a ST7920 and two ST7921 used to drive a 64 × 128 dot matrix LCD panel.	270
Figure 6.31	Diagram of the correspondence between the GDRAM addresses and the display pixels.	271
Figure 6.32	Example of a typical SPI bus connection between many devices.	273
Figure 6.33	Diagram of the four possible SPI modes, depending on the clock polarity and clock phase.	273
Figure 6.34	Diagram of the Hardware Abstraction Layer that is implemented in the display module.	276
Figure 6.35	Frames of the animation that are shown when the alarm is activated.	282
Figure 6.36	Examples of other systems based on Mbed that make use of LCD displays.	292
Figure 7.1	The smart home system is now connected to an LCD display.	297
Figure 7.2	Diagram of the limit switches that are considered in this chapter.	298
Figure 7.3	The smart home system is now connected to a PIR sensor and a set of four buttons.	298
Figure 7.4	The smart home system is now connected to a 5 V DC motor using a relay module.	299
Figure 7.5	Conceptual diagram of the circuit that is used to activate the DC motor, LED1, and LED2.	300
Figure 7.6	Diagram of the field of view and the effective range of the HC-SR501 PIR sensor.	302
Figure 7.7	Adjustments and connector of the HC-SR501 PIR sensor.	303
Figure 7.8	A typical limit switch. Note the connectors on the bottom: common, normally open and normally closed.	304
Figure 7.9	Conceptual diagram of the normal flow of a program altered by an interrupt service routine.	305
Figure 7.10	Conceptual diagram of the normal flow of a program altered by two interrupt service routines.	306
Figure 7.11	Pulse generated by PIR sensor when motion is detected and the corresponding initialization and callbacks.	320

List of Figures

Figure 7.12	Diagram of a typical circuit that is used in a relay module.	337
Figure 7.13	Diagram of a typical circuit that is used to turn on and off an AC motor using a relay module.	338
Figure 7.14	On the left, picture of the light system, with its light detector on top. On the right, a diagram of the system.	338
Figure 7.15	Diagram of a typical circuit that is used to turn on and off an AC lamp using a relay module.	339
Figure 8.1	The smart home system is now connected to an LCD display.	343
Figure 8.2	The smart home system has now an RGB LED, a light sensor, and a circuit for audio playback.	344
Figure 8.3	Diagram of the connection of the RGB LED.	345
Figure 8.4	Diagram of the light colors that result dependent on the LEDs that are turned on.	345
Figure 8.5	ST Zio connectors of the NUCLEO-F429ZI board.	346
Figure 8.6	Diagram of the connection of the LDR.	347
Figure 8.7	Diagram of the connection of the low pass filter and the audio jack.	347
Figure 8.8	Information shown on the serial terminal when program "Subsection 8.2.1.b" is running.	348
Figure 8.9	Connection of a high-brightness LED to a NUCLEO board pin (optional).	352
Figure 8.10	Simplified diagram of a timer.	353
Figure 8.11	Periodic signal generated by the built-in timer of a microcontroller.	354
Figure 8.12	Example of the variation of LED brightness by the pulse width modulation technique.	355
Figure 8.13	Detail on how the "Welcome to the smart home system" message is generated using PWM.	356
Figure 8.14	Output of serial terminal generated by proposed example 8.4.	378
Figure 8.15	Diagram of a negative feedback control system.	383
Figure 8.16	Diagram of the negative feedback control system implemented in Example 8.5.	383
Figure 8.17	On the left, smart city bike light mounted on a bike. On the right, rear and front lights and the mobile app.	384
Figure 9.1	The smart home system is now connected to an SD card.	389
Figure 9.2	The smart home system is now connected to an SD card.	390
Figure 9.3	Details of the SD card module pins and how to insert the SD card into the module.	391
Figure 9.4	Diagram of a typical organization of a filesystem.	392
Figure 9.5	Example of events storage messages.	393

Figure 9.6	Example of the content of an event file stored on the SD card as a .txt file.	394
Figure 9.7	Example of events storage messages.	401
Figure 9.8	Example of the content of an event file stored on the SD card as a .txt file.	402
Figure 9.9	Example of the file listing that is shown in the PC.	404
Figure 9.10	Two examples of opening a file: first, when the file exists, and second, when the file does not exist.	408
Figure 9.11	Diagram of a typical evolution of a repository.	414
Figure 9.12	Repository of an example of a game console based on Mbed.	417
Figure 10.1	The smart home system will be connected to a smartphone via Bluetooth Low Energy.	421
Figure 10.2	Connections to be made between the NUCLEO board and the HM-10 module.	422
Figure 10.3	Basic functionality of the HM-10 module pins.	423
Figure 10.4	Screenshots of the “Smart Home System App,” showing the sequence to connect and use the application.	424
Figure 10.5	Illustration of the names and behaviors of each device in the BLE startup process.	445
Figure 10.6	Illustration of the names and behaviors of each device in a typical BLE communication.	445
Figure 10.7	“Anybus wireless bolt” built with Mbed contains elements introduced in this chapter.	448
Figure 11.1	The smart home system is now able to serve a web page.	453
Figure 11.2	The smart home system is now connected to a ESP-01 module.	454
Figure 11.3	Basic functionality of the ESP-01 module pins.	455
Figure 11.4	Diagram of the communication that is implemented between the different devices.	456
Figure 11.5	Steps to follow in the test program used in this subsection.	456
Figure 11.6	Web page served by the ESP-01 module.	457
Figure 11.7	The “AT” command (attention) is sent to the ESP-01 module, which replies “OK”.	458
Figure 11.8	The “AT+CWMODE=1” command (mode configuration) is sent to the ESP-01 module, which replies “OK”.	458
Figure 11.9	The “AT+CWJAP” command (Join Access Point) is sent to the ESP-01 module.	459
Figure 11.10	The “AT+CIFSR” command (Get IP Address) is sent to the ESP-01 module.	459
Figure 11.11	The “AT+CIPMUX=1” command to enable multiple connections is sent to the ESP-01 module.	459
Figure 11.12	The “AT+CIPSERVER=1,80” command (creates a TCP server) is sent to the ESP-01 module.	460

List of Figures

Figure 11.13 The “AT+CIPSTATUS” command shows the connection status of the ESP-01 module.	460
Figure 11.14 A request to the ESP-01 module is sent by a web browser.	461
Figure 11.15 The ESP-01 module indicates that a network connection with ID of 0 has been established.	461
Figure 11.16 The “AT+CIPSTATUS” command shows the connection status of the ESP-01 module.	462
Figure 11.17 The “AT+CIPSEND=0,52” command (sends data) is sent to the ESP-01 module, and it responds “>”.	462
Figure 11.18 The “AT+CIPSEND=0,52” command (sends data) is sent to the ESP-01 module.	463
Figure 11.19 The “AT+CIPCLOSE=0” command (close a TCP connection) is sent to the ESP-01 module.	463
Figure 11.20 Web page served by the ESP-01 module.	463
Figure 11.21 Example of an Mbed-based system having a web service with real-time insights, alerts and reports.	493
Figure 12.1 First proposal of the hardware modules of the irrigation system.	505
Figure 12.2 Final version of the hardware modules of the Irrigation System.	507
Figure 12.3 Connection diagram of all the hardware elements of the irrigation system.	509
Figure 12.4 Software design of the irrigation system program.	511
Figure 12.5 Software modules of the irrigation system program.	512
Figure 12.6 Diagram of the .cpp and .h files of the irrigation system software.	512
Figure 12.7 Diagram of the proposed FSM.	516
Figure 12.8 Layout of the LCD for the Programmed irrigation mode when the system is waiting to irrigate.	518
Figure 12.9 Layout of the LCD for the Programmed irrigation mode when the system is irrigating.	518
Figure 12.10 Layout of the LCD for the Programmed irrigation mode when irrigation is skipped.	518
Figure 12.11 Layout of the LCD for the Continuous irrigation mode.	518
Figure 12.12 Image of the HC-SR04 ultrasonic module.	550
Figure 12.13 Image of a microphone module.	550
Figure 12.14 Image of a digital barometric pressure module.	550
Figure 12.15 Image of a micro servo motor.	551
Figure 12.16 Image of a GPS module. Note the GPS antenna on the left.	551
Figure 12.17 Image of an NB-IoT cellular module. Note the SIM card slot on the right.	552

List of Tables

Table 1	Organization in the case of two 90-minute or 120-minute lessons per week.	xxix
Table 2	Organization in the case of one 180-minute lesson per week.	xxx
Table 3	Bill of materials (BOM) used in Chapters 1 to 11.	xxxi
Table 1.1	Elements contained in the smart home system block diagrams illustrated in Figure 1.1.	4
Table 1.2.	Elements of the smart home system and the corresponding representation implemented.	8
Table 1.3	Behaviors of the program that is used in this subsection to test the buttons.	10
Table 1.4	New proposed implementation for Example 1.1.	16
Table 1.5	Proposed modifications of the code in order to achieve the new behavior.	16
Table 1.6	Proposed modifications in the code in order to achieve the new behavior.	20
Table 1.7	Proposed modifications in the code in order to achieve the new behavior.	24
Table 1.8	Proposed modifications in the code in order to achieve the new behavior.	28
Table 1.9	Proposed modifications in the code in order to achieve the new behavior.	28
Table 1.10	Summary of the smart home system buttons that should be pressed in each case.	28
Table 1.11	Proposed modification in the code in order to achieve the new behavior.	33
Table 1.12	Proposed modifications in the code in order to use the alternative names of D2 to D7.	39
Table 2.1	Sections in which lines were added to Code 2.4 and Code 2.5.	62
Table 2.2	Available configurations of the UnbufferedSerial object.	65
Table 2.3	Lines that were added to the program of Example 2.2.	70
Table 2.4	Lines that were added and removed from the code used in Example 2.3.	74
Table 2.5	Lines that were added from the code used in Example 2.4.	76
Table 3.1	Examples of the voltage at V_{OUT} using the connection shown in Figure 3.5.	90
Table 3.2	Available commands of the program used to test the LM35 temperature sensor and the potentiometer.	93
Table 3.3	Sections in which lines were added to Example 2.5.	96
Table 3.4	Lines that were modified from Example 3.1.	97
Table 3.5	Section where a line was added to Example 3.1.	98
Table 3.6	Lines that were modified from Example 3.2.	100
Table 3.7	Sections in which lines were added to Example 3.2.	101
Table 3.8	Sections in which lines were removed from Example 3.2.	101
Table 3.9	Functions in which lines were removed from Example 3.2.	101

Table 3.10	Lines added to the function availableCommands().	102
Table 3.11	New variables that are declared in the function uartTask().	103
Table 3.12	C language basic arithmetic type specifiers.	104
Table 3.13	C99 standard definitions of new integer types.	105
Table 3.14	Sections in which lines were modified and added to Example 3.3.	107
Table 3.15	Functions in which lines were added to Example 3.3.	107
Table 3.16	Sections in which lines were added or modified in Example 3.4.	111
Table 4.1	Available commands for the program used to test the matrix keypad and to configure the RTC.	133
Table 4.2	Sections and functions in which lines were added to Example 3.5.	135
Table 4.3	Sections and functions in which lines were added to Example 4.1.	140
Table 4.4	Sections in which lines were added to Example 4.2.	146
Table 4.5	Definitions, variables name, and variable initializations that were modified from Example 4.2.	146
Table 4.6	Sections in which lines were removed from Example 4.2.	146
Table 4.7	Functions in which lines were removed from Example 4.2.	147
Table 4.8	Sections in which lines were added to Example 4.3.	150
Table 4.9	Details of the struct tm.	151
Table 5.1	Functionalities and roles of the smart home system modules.	178
Table 5.2	Functions of the smart home system module.	179
Table 5.3	Functions of the fire alarm module.	179
Table 5.4	Functions of the code module.	179
Table 5.5	Functions of the user interface module.	180
Table 5.6	Functions of the PC serial communication module.	180
Table 5.7	Main functionality of the event log module.	181
Table 5.8	Functions of the siren module.	181
Table 5.9	Functions of the strobe light module.	182
Table 5.10	Functions of the gas sensor module.	182
Table 5.11	Functions of the temperature sensor module.	182
Table 5.12	Main functionality of the matrix keypad module.	182
Table 5.13	Functions of the date and time module.	182
Table 5.14	Variables that will be declared as static inside given functions.	211
Table 5.15	Public and private variables declared in each module.	212
Table 5.16	Public and private functions.	213
Table 5.17	Sections of the template that are used to write the .h file of each module.	217

Table 5.18	Sections of the template used to write the .cpp file of each module.	218
Table 5.19	Example of extern variables that are used in the implementation of the smart home system.	220
Table 6.1	A typical character set of an LCD character display.	228
Table 6.2	Part of the character set defined by ASCII and ISO/IEC 8859.	228
Table 6.3	Summary of the character LCD display instructions that are used in this chapter.	230
Table 6.4	Sections in which lines were added to user_interface.cpp.	234
Table 6.5	Address reference of the PCF8574 module. The addresses used in the proposed setup are highlighted.	254
Table 6.6	Summary of the graphical LCD display instructions that are used in this chapter.	268
Table 6.7	Connections of the graphical LCD display used in this book when the serial bus option is selected.	269
Table 6.8	Signals of the SPI bus.	272
Table 6.9	Summary of the NUCLEO board pins that are used to implement the SPI bus communication.	272
Table 6.10	Sections in which lines were added to user_interface.cpp.	283
Table 6.11	Sections in which lines were removed from user_interface.cpp.	283
Table 6.12	Comparison between UART, SPI, and I2C.	291
Table 7.1	Summary of the signals applied to the motor depending on IN1 and IN2, and the resulting behavior.	300
Table 7.2	Summary of the connections between the NUCLEO board and the relay module.	301
Table 7.3	Summary of other connections that should be made to the relay module.	301
Table 7.4	Summary of the connections between the NUCLEO board and the HC-SR501 PIR sensor.	303
Table 7.5	Summary of connections to the breadboard that should be made on the HC-SR501 PIR sensor.	303
Table 7.6	Summary of the buttons that are connected in Figure 7.3.	304
Table 7.7	Example of an interrupt service table.	306
Table 7.8	Sections in which lines were added to smart_home_system.cpp.	308
Table 7.9	Sections in which lines were added to user_interface.cpp.	311
Table 7.10	Functions in which lines were added in pc_serial_com.cpp.	312
Table 7.11	Sections in which lines were added in pc_serial_com.cpp.	312
Table 7.12	Sections in which lines were added to smart_home_system.cpp.	314
Table 7.13	Public global objects that were renamed in user_interface.cpp.	314

Table 7.14	Private functions that were renamed in user_interface.cpp.	315
Table 7.15	Sections in which lines were added to user_interface.cpp.	315
Table 7.16	Functions in which lines were added in pc_serial_com.cpp.	318
Table 7.17	Sections in which lines were added to pc_serial_com.cpp.	318
Table 7.18	Sections in which lines were added to smart_home_system.cpp.	321
Table 7.19	Functions in which lines were added in pc_serial_com.cpp	321
Table 7.20	Sections in which lines were added in pc_serial_com.cpp.	321
Table 7.21	Sections in which lines were added to event_log.cpp.	325
Table 7.22	Sections in which lines were added to user_interface.cpp.	325
Table 7.23	Sections in which lines were added to smart_home_system.cpp.	328
Table 7.24	Sections in which lines were added to user_interface.cpp.	335
Table 7.25	Variables that were renamed in user_interface.cpp.	336
Table 8.1	Summary of the PWM timers that are already in use or occupied by other functionalities.	349
Table 8.2	On time and off time of the LEDs used in the program “Subsection 8.2.2”.	354
Table 8.3	Sections in which lines were added to smart_home_system.cpp.	358
Table 8.4	Lines that were modified in siren.cpp.	358
Table 8.5	Sections in which lines were added to user_interface.cpp.	364
Table 8.6	Sections in which lines were removed from bright_control.cpp.	365
Table 8.7	Functions in which lines were removed from bright_control.cpp.	365
Table 8.8	Sections in which lines were added to user_interface.cpp.	372
Table 8.9	Sections in which lines were added to light_system.cpp.	373
Table 8.10	Sections in which lines were added to light_system.h.	373
Table 8.11	Sections in which lines were added to light_system.cpp.	375
Table 9.1	Summary of the connections between the NUCLEO board and the SD card.	391
Table 9.2	Sections in which lines were added to pc_serial_com.h.	395
Table 9.3	Sections in which lines were added to pc_serial_com.cpp.	395
Table 9.4	Functions in which lines were added in pc_serial_com.cpp.	395
Table 9.5	Sections in which lines were added to event_log.h.	396
Table 9.6	Sections in which lines were added to event_log.cpp.	396
Table 9.7	Sections in which lines were added to pc_serial_com.cpp.	405
Table 9.8	Functions in which lines were added in pc_serial_com.cpp.	405
Table 9.9	Sections in which lines were added to sd_card.h.	406
Table 9.10	Sections in which lines were added to pc_serial_com.cpp.	409

Table 9.11	Functions in which lines were added in pc_serial_com.cpp.	409
Table 9.12	Sections in which lines were added to sd_card.h.	412
Table 9.13	Summary of typical information available about a repository.	414
Table 10.1	Summary of the connections between the NUCLEO board and the HM-10 module.	423
Table 10.2	Summary of the messages from the NUCLEO board to the application.	425
Table 10.3	Summary of the messages from the application to the NUCLEO board.	425
Table 10.4	Sections in which lines were added to smart_home_system.cpp.	427
Table 10.5	Sections in which lines were added to event_log.cpp.	431
Table 10.6	Sections in which lines were added to ble_com.h.	431
Table 10.7	Sections in which lines were added to event_log.cpp.	434
Table 10.8	Sections in which lines were added to event_log.h.	434
Table 10.9	Examples of functions with parameters passed by reference and by value.	439
Table 10.10	Sections in which lines were smart_home_system.cpp.	440
Table 11.1	Summary of the connections between the NUCLEO board and the ESP-01 module.	455
Table 11.2	Summary of other connections that should be made to the ESP-01.	455
Table 11.3	Summary of the AT+CIPSTATUS return values.	460
Table 11.4	Sections in which lines were added to smart_home_system.cpp.	465
Table 11.5	Sections in which lines were added to pc_serial_com.cpp.	466
Table 11.6	Functions in which lines were added in pc_serial_com.cpp.	466
Table 11.7	Sections in which lines were added to pc_serial_com.cpp.	472
Table 11.8	Functions in which lines were added in pc_serial_com.cpp.	472
Table 11.9	Sections in which lines were added to wifi_com.cpp.	476
Table 11.10	Functions in which lines were added in wifi_com.cpp.	482
Table 11.11	Sections in which lines were added to wifi_com.cpp.	482
Table 11.12	Sections in which lines were added to wifi_com.cpp.	487
Table 11.13	Sections in which lines were removed from wifi_com.cpp.	487
Table 11.14	States of the FSM that were modified in wifi_com.cpp.	488
Table 11.15	Sections in which lines were modified in wifi_com.cpp.	490
Table 11.16	Summary of the main characteristics of the Wi-Fi versions.	491
Table 12.1	Summary of the proposed steps to design and implement an embedded system project.	497
Table 12.2	Selection of the project to be implemented.	500
Table 12.3	Summary of the SMART mnemonic acronym.	501

Table 12.4	Elements that will be used to define a use case.	502
Table 12.5	Summary of the main characteristics of two home irrigation systems currently available on the market.	502
Table 12.6	Initial requirements defined for the home irrigation system.	503
Table 12.7	Use Case #1 – Title: The user wants to irrigate plants immediately for a couple of minutes.	503
Table 12.8	Use Case #2 – Title: The user wants to program irrigation to take place for ten seconds every six hours.	504
Table 12.9	Use Case #3 – Title: The user wants the plants not to be irrigated.	504
Table 12.10	Comparison of moisture sensors.	506
Table 12.11	Comparison of solenoids valves.	506
Table 12.12	Summary of the connections between the NUCLEO board and the character-based LCD display.	507
Table 12.13	Summary of other connections that should be made to the character-based LCD display.	507
Table 12.14	Summary of the connections between the NUCLEO board and the buttons.	508
Table 12.15	Summary of connections that should be made to the HL-69 moisture sensor.	508
Table 12.16	Summary of connections that should be made to the relay module.	508
Table 12.17	Summary of connections that should be made to the FPD-270A.	508
Table 12.18	Bill of materials.	510
Table 12.19	Functionalities and roles of the home irrigation system modules.	513
Table 12.20	Private objects and variables of the moisture_sensor module.	513
Table 12.21	Private objects and variables of the buttons module.	513
Table 12.22	Private objects and variables of the irrigation_timer module.	514
Table 12.23	Private objects and variables of the irrigation_control module.	514
Table 12.24	Private objects and variables of the user_interface module.	514
Table 12.25	Private objects and variables of the relay module.	514
Table 12.26	Public functions of the irrigation_system module.	514
Table 12.27	Public functions of the moisture_sensor module.	514
Table 12.28	Public functions of the buttons module.	515
Table 12.29	Public functions of the irrigation_timer module.	515
Table 12.30	Public functions of the irrigation_control module.	515
Table 12.31	Public functions of the display module.	515
Table 12.32	Public functions of the relay module.	515
Table 12.33	Some of the initial requirements defined for the home irrigation system.	522

Table 12.34	Sections in which lines were modified in irrigation_control.h.	544
Table 12.35	Accomplishment of the requirements defined for the home irrigation system.	545
Table 12.36	Accomplishment of the use cases defined for the home irrigation system.	546
Table 12.37	Elements that summarize the most important information about the home irrigation system.	548

Chapter 1

Introduction to
Embedded Systems

1.1 Roadmap

1.1.1 What You Will Learn

After you have studied the material in this chapter, you will be able to:

- Implement basic programs on the NUCLEO board using flow control, logical operators, and variables.
- Explain the basic concepts about how microcontrollers operate.

1.1.2 Contents of This Chapter

In every embedded system there is an element that reads the inputs, drives the outputs, and controls the communications and interfaces. This element is usually a microcontroller. To perform its tasks, the microcontroller runs a program. In this chapter, the process of writing a program is explained as well as how to upload it to the NUCLEO-F429ZI board (hereinafter called the NUCLEO board) that is used in this book. Detailed information about this board can be found in [1].

The chapter also explains how to control the LEDs of the NUCLEO board and the connected buttons. These buttons are used in example problems related to a smart home system project.

The examples in this chapter make use of some elements of the Arm Mbed OS 6, which is a free, open-source, rapid development platform designed to help developers get started building *Internet of Things* (*IoT*) applications quickly. An introduction to Mbed OS 6 can be found in [2].

In this way, some of the main concepts about embedded programming using the C and C++ languages are introduced, such as *nested if* statements, OR and AND logical operators, and Boolean and non-Boolean variables.

Throughout the chapter, the reader is guided on how to *compile* the examples and load them onto the NUCLEO board using Keil Studio Cloud, which allows the developer to write code from scratch or import an existing project and modify it to suit the developer's requirements. An introduction to Keil Studio Cloud is available in [3].

To use Keil Studio Cloud, the only thing the reader needs is an Arm Mbed account. [4] explains how to create an Arm Mbed account or log in if an account is already created. Keil Studio Cloud is available from [5] and a quick-start guide is available from [6].



NOTE: Those readers who prefer to use a dedicated desktop setup can use the Mbed Studio IDE (*Integrated Development Environment*), which is available from [7], or any other IDE, for example the STM32CubeIDE, which is introduced in [8].

In the Under the Hood section, the basic principles of microcontrollers are introduced. Finally, in the Case Study section, a smart door lock based in Mbed OS is analyzed in order to show that many of the concepts introduced in this chapter are used in commercial products.

1.2 Fundamentals of Embedded Systems

1.2.1 Main Components of Embedded Systems

Throughout the chapters in this book, a smart home system project will be implemented. In this section, different smart home system implementations are analyzed in order to highlight the fundamentals of embedded systems.

Figure 1.1 shows four different implementations of a smart home system, adapted from different sources. The inputs are indicated in yellow, the microcontroller in light blue, the outputs in light red, the communication links in light gray, the human interfaces in blue, and the power supply in green. The elements contained in the block diagrams illustrated in Figure 1.1 are summarized in Table 1.1.

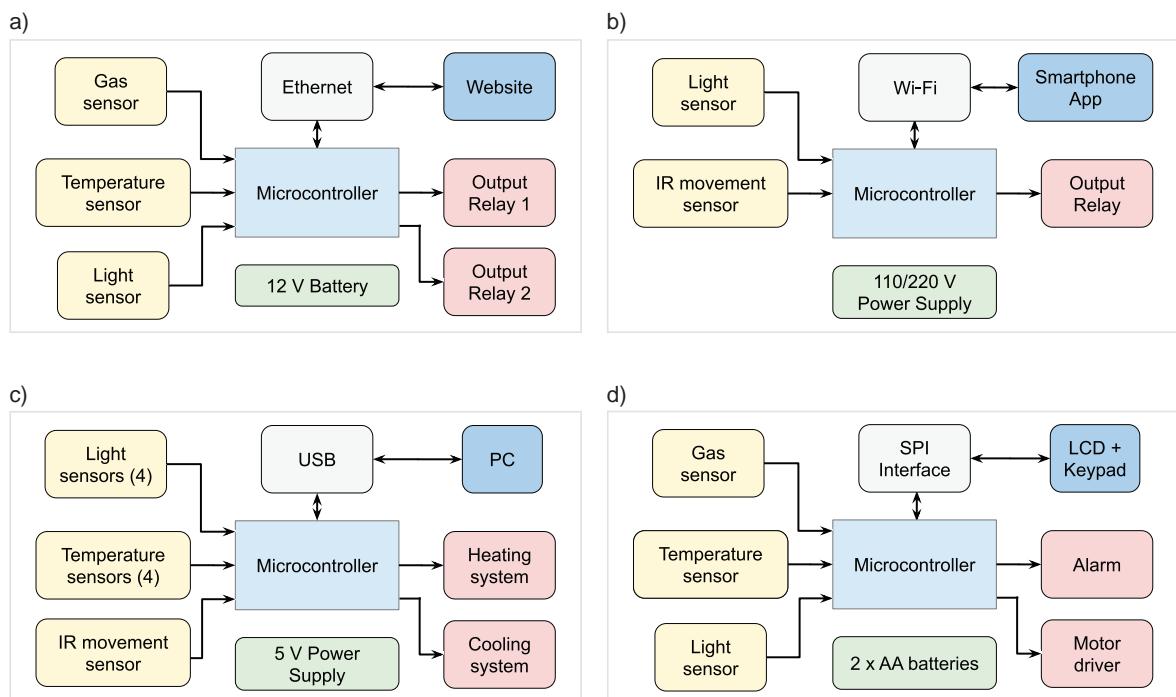


Figure 1.1 Four different block diagram representations of a smart home system.

Table 1.1 Elements contained in the smart home system block diagrams illustrated in Figure 1.1.

Smart Home	Inputs (yellow)	Microcontroller (light blue)	Outputs (light red)	Communication (light gray)	Interface (blue)	Power supply (green)
(a)	Gas, light, and temperature sensors	Yes	Two relays	Ethernet	Website	12 V battery
(b)	IR movement and light sensors	Yes	One relay	Wi-Fi	Smartphone App	110/220 V
(c)	Light, IR movement, and temperature sensors	Yes	Heating/cooling system	USB	PC	5 V
(d)	Gas, light, and temperature sensors	Yes	Alarm/motor driver	SPI	Keypad + LCD display	2 x AA batteries

From Table 1.1, the following conclusions can be made:

- All the implementations have a *microcontroller*.
- The type and number of sensors depends on the implementation.
- All the implementations have outputs, a relay being the most frequent one.
- Communications are implemented by different technologies (Ethernet, Wi-Fi, USB).
- Different interfaces are used: a website, a smartphone app, a keypad plus an LCD display.
- The power supply is indicated in the four diagrams (12 V battery, 110/220 V, etc.).

The microcontroller is the heart of the system: all the inputs are connected to the microcontroller and all the outputs are driven by the microcontroller, while the communications and the interfaces are controlled by the microcontroller. For this reason, every embedded system has one microcontroller (at least), or a similar element that plays the same role.



DEFINITION: A typical definition of an *embedded system* is a computer system that has a dedicated function within a larger mechanical or electrical system and is embedded as part of a complete device, often including electrical or electronic hardware and mechanical parts.

DEFINITION: A typical definition of a *microcontroller* is a small computer on a single integrated circuit (IC) chip which typically contains one or more processor cores along with memory and programmable peripherals.



NOTE: The Under the Hood section, at the end of this chapter, discusses in detail what is inside a microcontroller. The aim is to make the reader use a microcontroller before explaining how they work.

The microcontroller runs a program made up of instructions by means of which the programmer instructs the microcontroller what to do. This book aims to enable the reader to write this type of program and to connect the appropriate sensors, actuators, and interfaces to the inputs and outputs of the microcontroller. The sensors will provide measurements of the environment, while the actuators will allow the microcontroller to have control over elements and devices within the environment.

Proposed Exercises

1. Using Table 1.1 as a reference, propose a smart home system having a maximum of two types of sensors (inputs), one type of actuator (output), one communication link, and two human interfaces.
2. Make the block diagram of the proposed smart home system.

Answers to the Exercises

1. The smart home system may include a gas sensor, a temperature sensor, an alarm, a keypad, and a communication link with a PC.
2. The diagram of the proposed smart home system is shown in Figure 1.2.

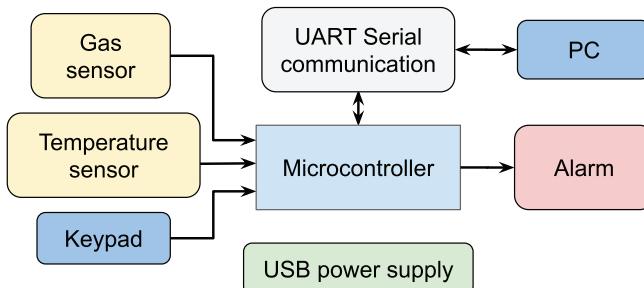


Figure 1.2 Diagram of the proposed smart home system.



NOTE: In Figure 1.2, the link with the PC is implemented by means of a UART serial communication (*Universal Asynchronous Receiver-Transmitter*). The UART peripheral is introduced later in Chapter 2.

1.2.2 First Implementation of the Smart Home System

This section will begin the implementation of the smart home system, following the diagram shown in Figure 1.3. This first implementation will detect fire using an over temperature detector and a gas detector and, if it detects fire, it will activate the alarm until a given code is entered. For this purpose, the NUCLEO board [1] provided with the STM32F429ZIT6U microcontroller [9] (referred to as the *STM32 microcontroller*) is used.

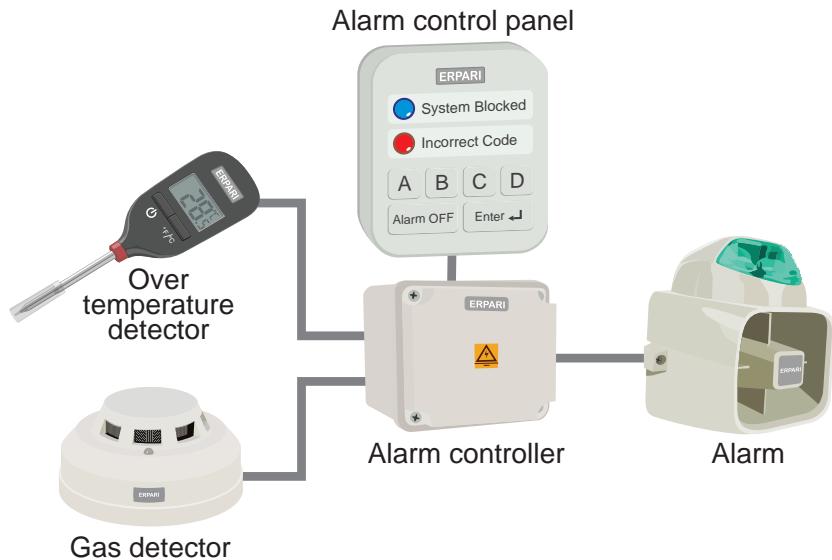


Figure 1.3 The smart home system that is implemented in this chapter.

The reader might notice that there are not as many buttons in the NUCLEO board as in the alarm control panel shown in Figure 1.3. Instead, the NUCLEO board is provided with only one user button and three LEDs, as shown in Figure 1.4. It can be concluded that a way to connect more buttons to the NUCLEO board is needed. For this purpose, the ST Zio connectors of the NUCLEO board shown in Figure 1.4 are used.

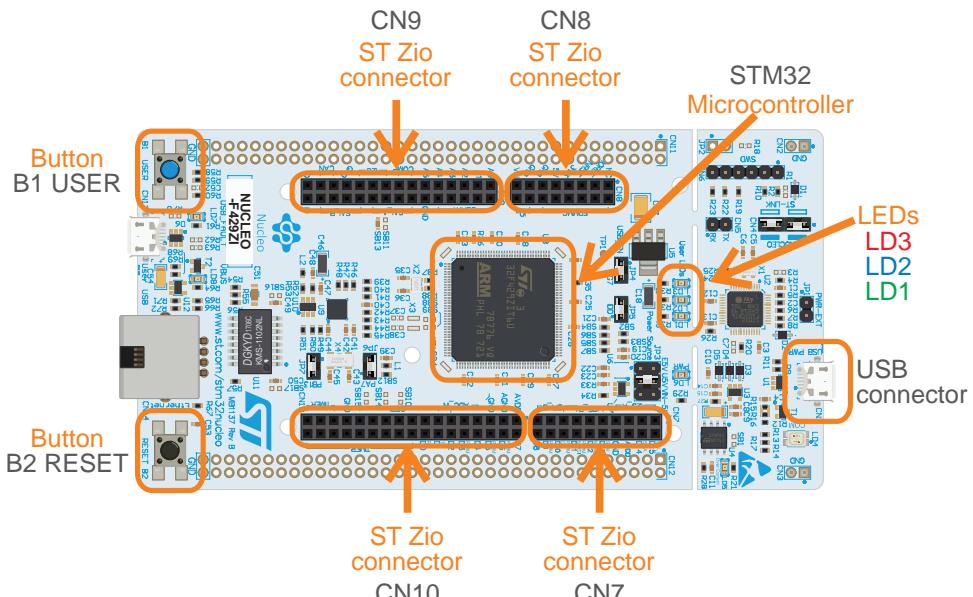


Figure 1.4 The NUCLEO-F429ZI board used in this book.

The overall diagram of the embedded system that will be connected in this chapter is shown in Figure 1.5. It can be seen that a breadboard is used, together with the most common buttons used in electronics, technically known as *tactile switches*. For breadboarding, any appropriate wire can be used, but it is recommended to use *jumper wires*.

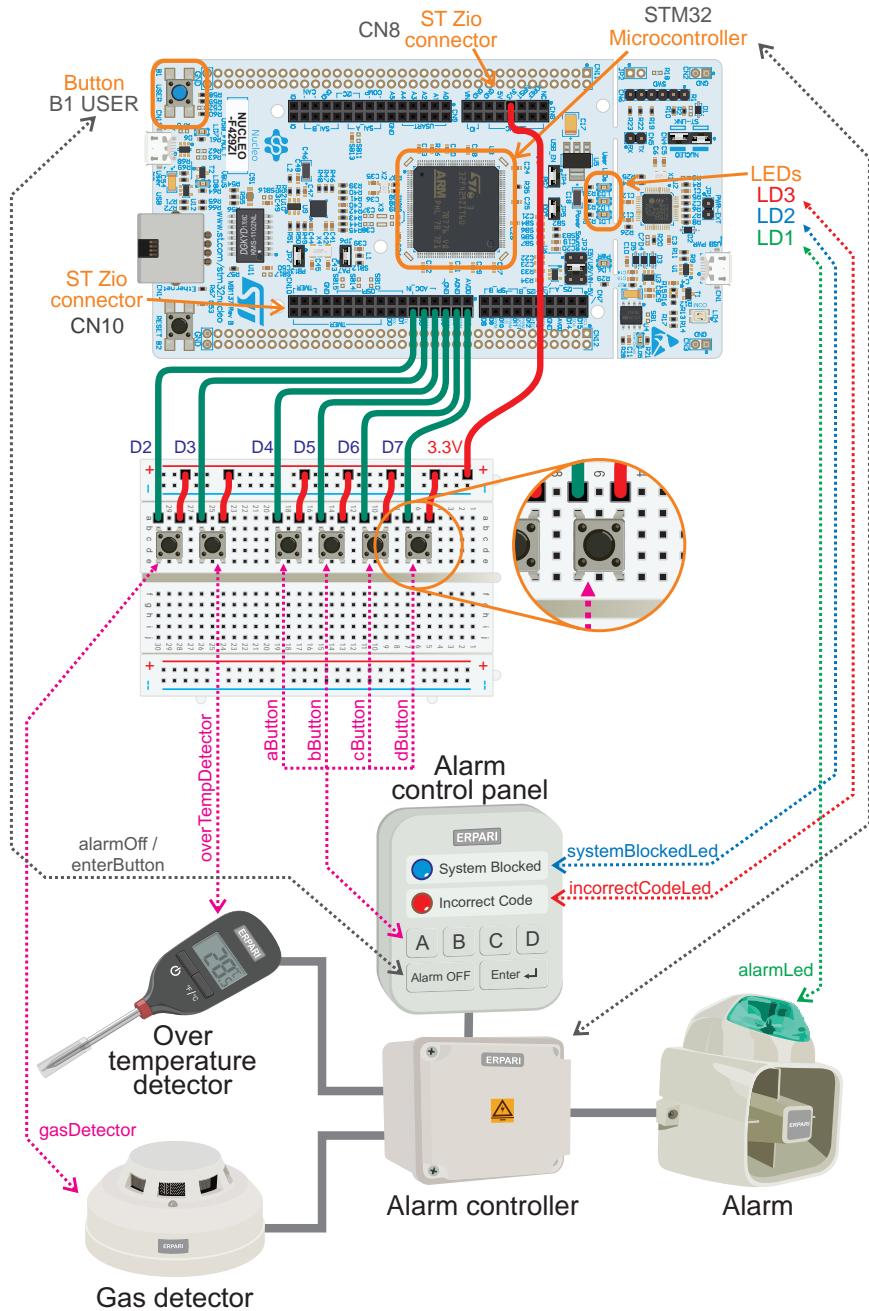


Figure 1.5 The elements of the smart home system introduced in Figure 1.3 with their corresponding components.

Figure 1.5 shows the elements of the smart home system introduced in Figure 1.3 and their proposed representation by means of the components connected. The names that will be used in the code to refer to each element are indicated in Figure 1.5 (i.e., gasDetector, alarmLed, etc.). It can be seen that the activation of the gas detector and the over temperature detector are simulated by buttons D2 and D3 placed over the breadboard. All these connections are summarized in Table 1.2.



TIP: It may be useful to indicate the function of each button and LED using a card next to each one.



NOTE: In Chapters 1 and 2, the alarm will be represented by an LED. In Chapter 3, a buzzer will be incorporated into the alarm in order to represent a siren. In later chapters it will be explained how to incorporate a strobe light into the alarm. In this way the alarm will have light and sound, as in Figure 1.5.

Table 1.2. Elements of the smart home system and the corresponding representation implemented.

Smart home system	Representation used
Alarm	LD1 LED (green)
System blocked LED	LD2 LED (blue)
Incorrect code LED	LD3 LED (red)
Alarm Off / Enter button	B1 USER button
Gas detector	Button connected to D2 pin
Over temperature detector	Button connected to D3 pin
A button	Button connected to D4 pin
B button	Button connected to D5 pin
C button	Button connected to D6 pin
D button	Button connected to D7 pin

In Table 1.2 it should be noticed that the B1 USER button is used for the Alarm Off and also for the Enter button. The specific use will vary during the examples and will be clearly indicated as it becomes necessary.

In Figure 1.6 and Figure 1.7 the details of the connections that should be made on the CN8 and CN10 ST Zio Connectors are shown.

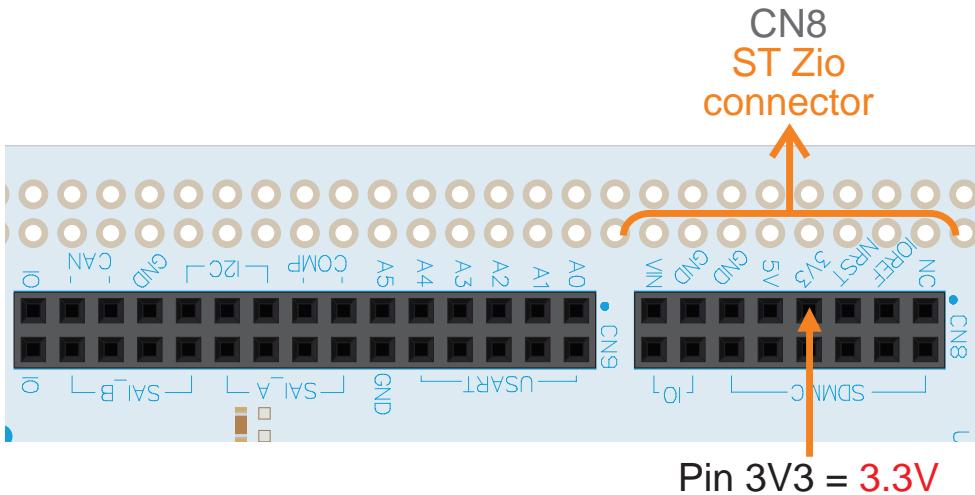


Figure 1.6 Detail of the connection made in the CN8 ST Zio connector of the NUCLEO board.

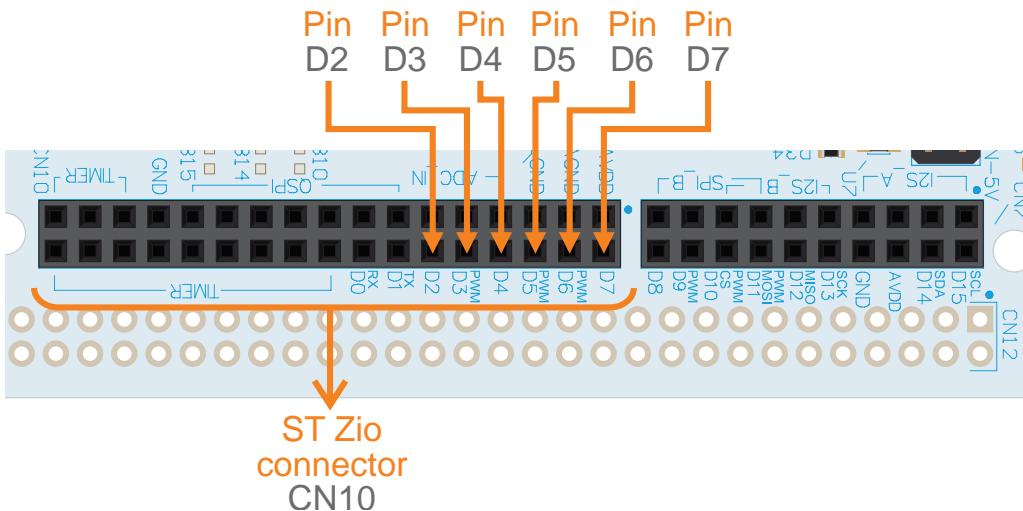


Figure 1.7 Detail of the connections made in the CN10 ST Zio connector of the NUCLEO board.



WARNING: It is crucial to connect the jumper wires and buttons exactly as indicated in Figure 1.5, Figure 1.6, and Figure 1.7. Otherwise, the programs presented in this chapter will not work.

To make all the connections, unplug the USB power supply from the NUCLEO board. Prior to reconnecting the USB power supply, check that there are no short circuits.

1.2.3 Getting Ready to Program the First Implementation of the Smart Home System

This subsection explains how to load a program onto the STM32 microcontroller of the NUCLEO board using Keil Studio Cloud in order to test if the buttons that were connected are working properly.

First, the reader must copy the URL of the repository of the “Subsection 1.2.3” program that is available in [10]. Then, in the “File” menu of Keil Studio Cloud, select “Import project”. The “Import project” window will be displayed in the web browser. Press “Add project” without modifying any of the configurations, and a new project named “Subsection 1.2.3” will be added to the list of available projects as the active project.



NOTE: If in the future changes are made in Keil Studio Cloud that alter the steps to compile and download the programs to the NUCLEO board, the corresponding instructions will be published in [10].

The program that was imported has the behaviors detailed in Table 1.3. When the B1 USER button of the NUCLEO board is pressed, LEDs LD1, LD2, and LD3 of the board are turned on. This is to ensure that the three LEDs and the program are working correctly. If the external buttons connected to D2 or D3 are pressed, LD1 is turned on. In this way it can be tested whether the buttons connected to D2 and D3 are properly connected and working. The same applies to the buttons connected to D4 and D5 by means of LD2, and the buttons connected to D6 and D7 by means of LD3.

Table 1.3 Behaviors of the program that is used in this subsection to test the buttons.

LED	Turns on if any of the following is pressed
LD1 (green)	B1 USER button, button connected to D2, button connected to D3
LD2 (blue)	B1 USER button, button connected to D4, button connected to D5
LD3 (red)	B1 USER button, button connected to D6, button connected to D7



NOTE: The code of the program that has been loaded onto the NUCLEO board to test if the six external buttons are working correctly will not be analyzed in this subsection. This is due to the fact that the behavior of this code is based on some elements that will be introduced and explained in the following subsections. For now, it is enough to be able to check if the six buttons are working.

In order to load the program onto the NUCLEO board, connect it to the computer as shown in Figure 1.8. If “NUCLEO-F429ZI” is not shown in the “Target hardware” section, press the “Find target” button. If a popup menu is displayed, select the “STM32 STLink” connection and follow the instructions. Once “NUCLEO-F429ZI” is shown in the “Target hardware” section, press the “Build project” button. The progress will be shown in the “Output” tab. Once the program has been compiled, a .bin file will be automatically downloaded by the web browser. Drag the downloaded .bin file to the drive assigned to the NUCLEO board (for example, D:\, named NODE_F429ZI). A “copying” window

will open, LED LD4 of the NUCLEO board will alternate red and green light, and when the window closes the program will already be running on the NUCLEO board.

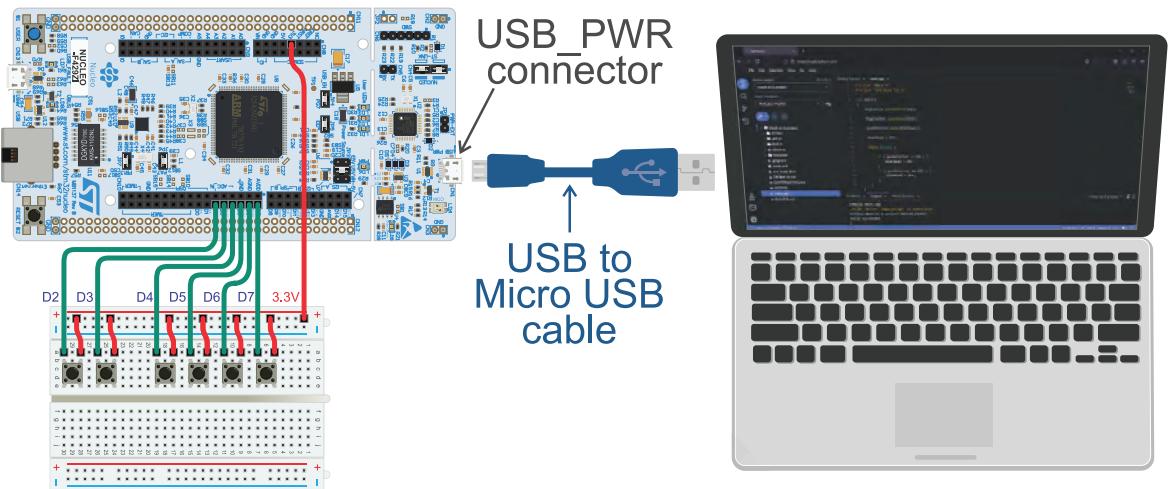


Figure 1.8 How to connect the NUCLEO board to a PC.

Next, press the B1 USER button and the buttons connected to D2, D3, D4, D5, D6, and D7, one after the other. If the behavior of the LEDs is as described in Table 1.3 then the buttons that were connected are working fine and the program has been correctly loaded onto the NUCLEO board.



TIP: If the B1 USER button is working as described in Table 1.3 but one of the switches D2 to D7 is not working as expected, check the corresponding connections. If the B1 USER button is not working as described in Table 1.3 then the program has not been properly loaded onto the NUCLEO board.

As stated in the preface, in this book selected problems, named Examples, are tackled in seven steps. The first set of examples are presented below.

Example 1.1: Activate the Alarm When Gas is Detected

Objective

Write the first program of the book, load it onto the NUCLEO board, and introduce the *if* statement.

Summary of the Expected Behavior

LD1 must turn on when the button connected to pin D2 of the NUCLEO board is pressed and turn off when this button is released.

Test the Proposed Solution on the Board

Import the project “Example 1.1” using the URL available in [10], build the project, and drag the .bin file onto the NUCLEO board. Then press and release the button connected to pin D2 and look at the behavior of LD1. The LED should turn on when the button is pressed and turn off when the button is released.

Discussion of the Proposed Solution

In embedded systems there is a set of statements that are executed forever until the power supply is removed. Figure 1.9 illustrates the general structure, which is composed of two main parts:

- *Libraries, Definitions and Global Declarations and Initializations*, indicated in blue.
- *Implementation of the main() function*, indicated in green.

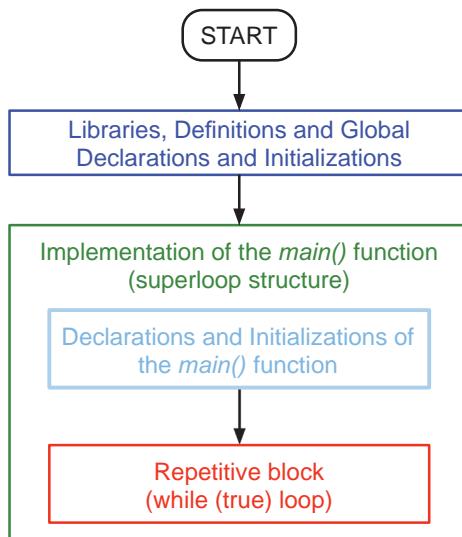


Figure 1.9 Structure of the proposed solution.

Libraries, Definitions and Global Declarations and Initializations includes the *software libraries* that are used. In addition, here some definitions of values can be established, and *global variables* can be declared and initialized, among other elements.

Implementation of the *main()* function consists of two parts, which are illustrated in Figure 1.9:

- *Declarations and Initializations of the main() function*, indicated in light blue.
- *Repetitive block*, indicated in red.

In Initializations of the *main()* function, the *inputs* and *outputs* of the board that are used are declared, and the initial states and other configurations are established.

The *repetitive block* is made by the statements that are executed repeatedly in the *while (true)* loop. This kind of structure is known as a *superloop*. That is, the code block corresponding to a *while* statement is executed as long as the condition of the *while* loop is true. If this condition is explicitly defined as *true* (i.e., *while (true)*) the code block of the *while* loop is executed forever.

Implementation of the Proposed Solution

The *main.cpp* file is where the main program is written in C or C++ language (the differences and history of C and C++ are discussed in Chapter 10 and in [11]). The *main.cpp* file of Example 1.1 is shown in Code 1.1. The parts corresponding to Figure 1.9 are highlighted in color. An explanation of each part is included over the code. In lines 1 and 2 of Code 1.1, `#include "mbed.h"` and `#include "arm_book_lib.h"` are used to instruct the Mbed Online Compiler to insert pre-written files that describe, among other things, the interfaces to external libraries that will then be included automatically by the tools, where *classes* and *macros* such as *DigitalIn*, *DigitalOut*, *ON*, and *OFF* are defined. The *main()* function is implemented from line 4 to line 23.

```

1 #include "mbed.h"
2 #include "arm_book_lib.h"
3
4 int main()
5 {
6     DigitalIn gasDetector(D2);
7
8     DigitalOut alarmLed(LED1);
9
10    gasDetector.mode(PullDown);
11
12    alarmLed = OFF;
13
14    while ( true ) {
15        if ( gasDetector == ON ) {
16            alarmLed = ON;
17        }
18
19        if ( gasDetector == OFF ) {
20            alarmLed = OFF;
21        }
22    }
23 }
```

Code 1.1 Implementation of Example 1.1.



NOTE: Double straight quotes ("") are used in lines 1 and 2 of Code 1.1. In the program codes that are introduced in this book, double and single straight quotes ('') will gradually be introduced and used for different purposes. It is important for the reader to understand that double and single curly quotes ("", ',', ') are different characters that cannot be used for the same purpose in the program codes. Similarly, a double straight quote cannot be replaced in the code by two consecutive straight single quotes.



NOTE: The colors used by different code editors to highlight reserved words and parts of the code may vary (i.e., some use green for `if` and others use blue). However, it has no relevance for the program code itself.

NOTE: C and C++ are free-form programming languages in which the positioning of characters in the program text is insignificant. However, to clarify the code structure in this book the Kernighan and Ritchie (K&R) indentation style [12] is used. The indentation rules of the K&R style are not discussed for the sake of brevity.

The instructions in line 6 and line 8 are to instruct the microcontroller that the button connected to D2 and LD1 (named LED1 in the code) will be used. The button will be used as an input, while LD1 will be used as an output (as indicated in Table 1.2). Line 6 declares a digital input with the name `gasDetector` and assigns it to button D2. Line 8 declares a digital output with the name `alarmLed` and assigns it to LED1. All these configurations are achieved by the `DigitalIn gasDetector(D2)` and `DigitalOut alarmLed(LED1)` declarations, where the corresponding *objects* are created.



NOTE: In this book, objects' names are stylized using *camel case* convention without capitalization of the first letter (known as *lower camel case* or *dromedary case*), as in `gasDetector` and `alarmLed`. In this way, these names can be easily differentiated from names defined by Mbed OS, like `DigitalIn`, `DigitalOut` or `PullDown`, which use the camel case convention with capitalization of the first letter.

In Line 10, `gasDetector.mode(PullDown)` is used to enable an internal pull-down resistor connected to the `DigitalIn` object `gasDetector`. A typical connection between a button and a microcontroller digital input based on a pull-down resistor is as shown on the left side of Figure 1.10. When the button is pressed, the NUCLEO board reads +3.3 V (high state). When the button is not pressed, the NUCLEO board reads 0 V (low state) because of the pull-down resistor ("R_pulldown"). In order to avoid connecting one resistor for each button, the internal pull-down resistors that are provided by the STM32 microcontroller are used in this example.

The right side of Figure 1.10 shows the simplified diagram of a typical connection between a button and a microcontroller digital input based on a pull-up resistor. In this case, when the button is pressed, the NUCLEO board reads a low state (0 V), and it reads a high state (3.3 V) when the button is not pressed. In the latter part of this book, pull-up resistors will be used so that the reader becomes familiar with them.

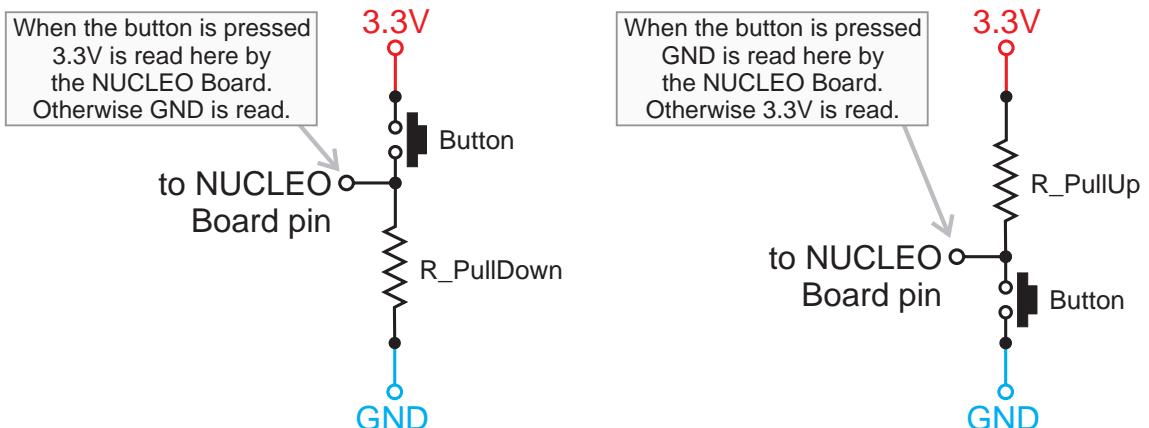


Figure 1.10 Conceptual diagram of how to connect a button to the NUCLEO board using pull-down and pull-up resistors.

In line 12, `alarmLed` is initialized to OFF. This instruction, together with lines 6 to 10, is highlighted in light blue in Code 1.1 and are called *Declarations and Initializations of the main() function*.

In line 14, the `while(true)` statement is used to indicate the beginning of the repetitive block that is repeated forever. In Example 1.3, the concept of a *Boolean variable* will be introduced, and this line will be explained in more detail.

Between lines 16 and 20, the following behavior is implemented: if the `gasDetector` is on (button connected to D2 is pressed) turn on the alarm (turn on LD1). If the `gasDetector` is off (button connected to D2 is not pressed) turn off the alarm (turn off LD1).



NOTE: To implement the comparison inside the `if` statements (line 15 and line 19) a double equals sign is used (i.e., “`==`”) while in order to assign values (line 16 and line 20) a single equals sign is used (i.e., “`=`”).

The reader should note that the braces “{” and “}” play a very important role in the code, because they are used to separate the code into parts.

The reader should also note that semicolons “;” are used to separate different statements, and parentheses “(” and “)” are used to indicate the beginning and ending of the arguments of the statements, as in `if()`, `while()`, and `main()`. Parentheses are also used to indicate which elements of the board are referred to with the objects `alarmLed` and `gasDetector`.

Proposed Exercises

1. What happens if one of the semicolons is replaced by a comma?
2. What happens if one of the `gasDetector` occurrences is replaced by `gasdetector`?

3. What can be changed in the code in order to use LD2 instead of LD1?
4. Is it possible to turn on LD1 and LD2 as the button connected to D2 is pressed?
5. What would happen if the `while (true)` statement was removed?
6. What will happen if the code in Table 1.4 is used?

Table 1.4 New proposed implementation for Example 1.1.

Lines in Code 1.1	New code to be used
<pre> 15 if (gasDetector == ON) { 16 alarmLed = ON; 17 } 18 19 if (gasDetector == OFF) { 20 alarmLed = OFF; 21 } </pre>	<pre> 16 if (gasDetector == ON) { 17 alarmLed = ON; 18 } else { 19 alarmLed = OFF; 20 } 21 </pre>

Answers to the Exercises

1. The Online Compiler will not be able to compile the code and will indicate an error in the line where the semicolon was replaced by a comma.
2. The C/C++ code is case sensitive, so `gasdetector` will be interpreted as a different element than `gasDetector` and, as a consequence, an error will be indicated.
3. Line 8 should be changed to *DigitalOut alarmLed (LD2);*
4. Yes. The lines indicated in Table 1.5 should be changed.
5. The program will work once and will be over in a few microseconds.
6. The program will work in the same way as Code 1.1. The new code has the advantage of being more compact.

Table 1.5 Proposed modifications of the code in order to achieve the new behavior.

Lines in Code 1.1	New code to be used
8 <code>DigitalOut alarmLed(LED1);</code>	8 <code>DigitalOut alarmLed1(LED1);</code> 9 <code>DigitalOut alarmLed2(LED2);</code>
12 <code>alarmLed = OFF;</code>	13 <code>alarmLed1 = OFF;</code> 14 <code>alarmLed2 = OFF;</code>
17 <code>alarmLed = ON;</code>	19 <code>alarmLed1 = ON;</code> 20 <code>alarmLed2 = ON;</code>
20 <code>alarmLed = OFF;</code>	23 <code>alarmLed1 = OFF;</code> 24 <code>alarmLed2 = OFF;</code>

Example 1.2: Activate the Alarm on Gas Presence or Over Temperature

Objective

Review the *if* statement and introduce the OR operator.

Summary of the Expected Behavior

An LED must turn on/off as one or more buttons connected to the NUCLEO board are pressed or released.

Test the Proposed Solution on the Board

Import the project “Example 1.2” using the URL available in [10], build the project, and drag the *.bin* file onto the NUCLEO board. Then press and release the buttons connected to D2 or to D3 and look at the behavior of LD1. LD1 should turn on when any of the buttons connected to D2 and D3 are pressed. LD1 should turn off when both buttons connected to D2 and D3 are released.

Discussion of the Proposed Solution

In Example 1.1, the alarm (LD1) was activated when gas was detected (button connected to D2 was pressed). In this example, the over temperature detector is incorporated into the system and is simulated by the button connected to D3.

In Example 1.1, the *if* statement was used to implement the proposed behavior depending on the state of the button connected to D2. In this case, if the gas detector (D2) or the over temperature detector (D3) is active, the alarm should turn on. Otherwise, the alarm should turn off. The logical OR behavior is indicated by means of the `||` operator in C/C++.

Figure 1.11 shows the proposed parts to implement the solution, following the structure proposed in Figure 1.9. In the *Libraries, Definitions and Global Declarations and Initializations* (Figure 1.11 [a]), the `mbed.h` and `arm_book_lib.h` libraries are included. In the *Declarations and Initializations of the main() function* (Figure 1.11 [b]), the digital inputs `gasDetector` and `overTempDetector` are declared and assigned to D2 and D3, and a digital output named `alarmLed` is declared and assigned to LD1. The repetitive block (Figure 1.11 [c]) turns on `alarmLed` if `gasDetector` or `overTempDetector` is pressed and turns off `alarmLed` if both are not pressed.

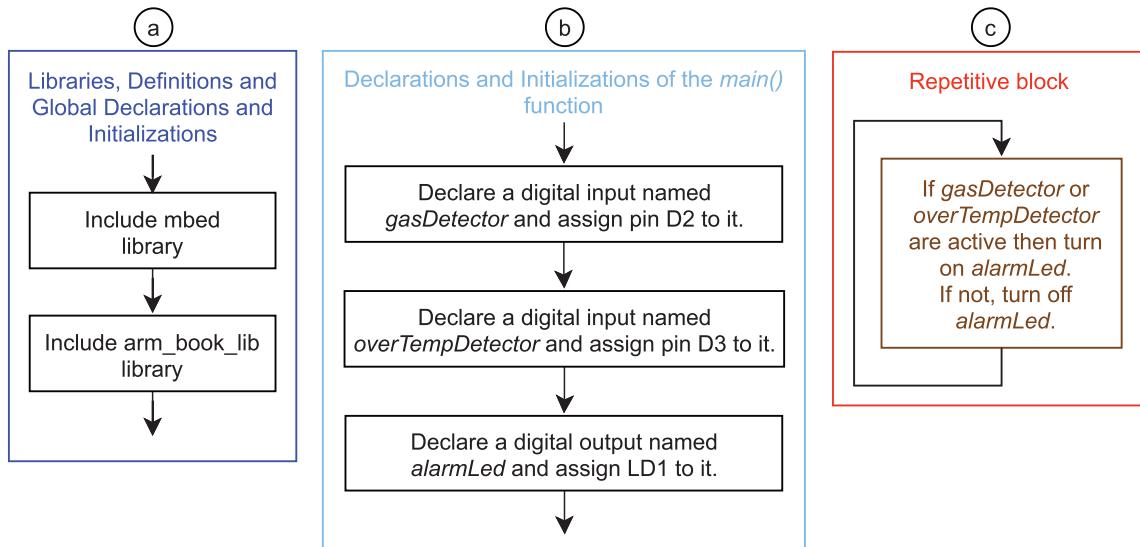


Figure 1.11 Main parts of the program of Example 1.2.

Figure 1.12 shows the details of the repetitive block. This block turns *alarmLed* (LD1) on if *gasDetector* or *overTempDetector* is active (buttons connected to D2 or D3 are pressed).

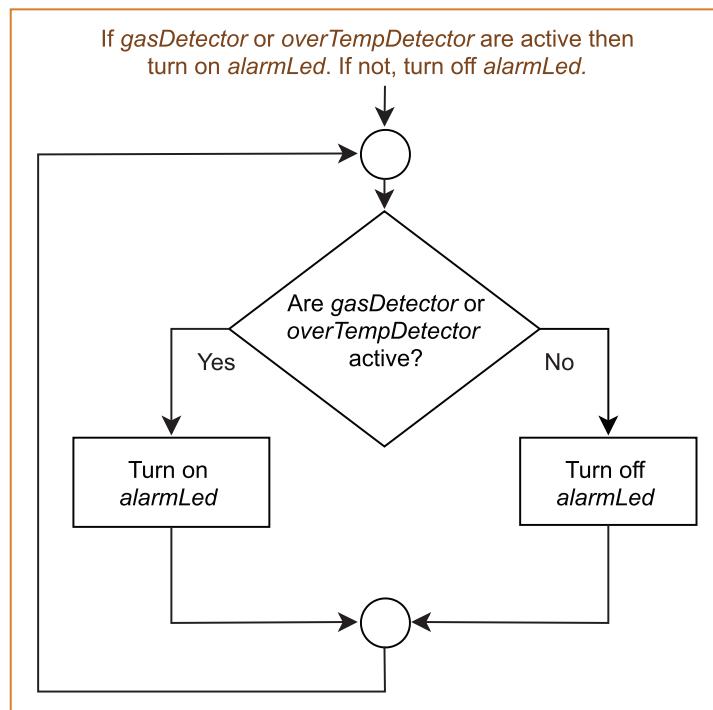


Figure 1.12 Details of the blocks that make up the repetitive block of Example 1.2.

Implementation of the Proposed Solution

In Code 1.2, the implementation of the proposed solution is presented. The parts corresponding to Figure 1.11 and Figure 1.12 are highlighted in color. An explanation of each part is included beside the code.

```

1 #include "mbed.h"
2 #include "arm_book_lib.h"
3
4 int main()
5 {
6     DigitalIn gasDetector(D2);
7     DigitalIn overTempDetector(D3);
8
9     DigitalOut alarmLed(LED1);
10
11    gasDetector.mode(PullDown);
12    overTempDetector.mode(PullDown);
13
14    while ( true ) {
15
16        if ( gasDetector || overTempDetector ) {
17            alarmLed = ON;
18        }
19        alarmLed = OFF;
20    }
21 }
22 }
```

The code is annotated with several callouts:

- A blue callout points to the first two lines: `#include "mbed.h"` and `#include "arm_book_lib.h"`. It states: "Include the libraries "mbed.h" and "arm_book_lib.h""
- A blue callout points to the declarations of `gasDetector` and `overTempDetector`: `DigitalIn gasDetector(D2);` and `DigitalIn overTempDetector(D3);`. It lists:
 - Digital input objects named `gasDetector` and `overTempDetector` are declared and assigned to D2 and D3, respectively.
 - Digital output object named `alarmLed` is declared and assigned to LED1.
 - `gasDetector` and `overTempDetector` are configured with internal pull-down resistors.
- An orange callout points to the `while (true) { ... }` loop. It states: "Do forever loop:" and then lists:
 - If `gasDetector` or `overTempDetector` are active, `alarmLed` is assigned ON.
 - Else, `alarmLed` is assigned OFF.

Code 1.2 Implementation of Example 1.2.

Proposed Exercises

1. What should be modified in order to simulate the alarm by means of LD2?
2. How can it be implemented so that when gas presence is detected or over temperature is detected LD1 and LD2 are both turned on?
3. How can line 16 of Example 1.2 be modified in order to use an explicit formulation of the condition, as in Example 1.1?
4. Is there another way to indicate the OR logical operator beside the `||` symbols used in Example 1.6?

Answers to the Exercises

1. In line 8, `LED1` should be replaced by `LED2`.
2. It can be achieved by means of the changes in Table 1.6.

Table 1.6 Proposed modifications in the code in order to achieve the new behavior.

Lines in Code 1.2	New code to be used
9 DigitalOut alarmLed(LED1);	9 DigitalOut alarmLed1(LED1); 10 DigitalOut alarmLed2(LED2);
17 alarmLed = ON;	18 alarmLed1 = ON; 19 alarmLed2 = ON;
19 alarmLed = OFF;	21 alarmLed1 = OFF; 22 alarmLed2 = OFF;

3. Line 16 can be rewritten as follows:

```
if ( gasDetector == ON || overTempDetector == ON )
```

4. The logical operator OR can be expressed as follows

```
if ( gasDetector == ON or overTempDetector == ON )
```



NOTE: In this book, the notation `||` is used because of historical reasons. However, in Mbed OS, `or` can be used to indicate the logical operator OR; in addition, other reserved words such as `not`, `and`, etc. can be used.

Example 1.3: Keep the Alarm Active After Gas or Over Temperature Were Detected

Objective

Learn how to use a Boolean variable to keep track of the state of a given element.

Summary of the Expected Behavior

The LED is turned on when one or more of the alarm conditions are activated (simulated by buttons connected to the NUCLEO board) and remains on until those alarm conditions are removed and another button is pressed.

Test the Proposed Solution on the Board

Import the project “Example 1.3” using the URL available in [10], build the project, and drag the `.bin` file onto the NUCLEO board. Then press and release the buttons connected to D2 and D3 and look at the behavior of LD1. LD1 should turn on when the buttons connected to D2 and D3 are pressed and turn off when the B1 USER button is pressed.

Discussion of the Proposed Solution

In Example 1.2, the alarm was directly controlled by the buttons connected to D2 and D3 that simulated the gas detector and the over temperature detector, respectively. When any of these buttons were pressed the alarm was turned on, and when both buttons were released the alarm was turned off. To keep the alarm activated until another condition occurs, the alarm state must be stored. This can be done by means of a *Boolean variable*. In this scheme, the activation of the alarm is determined by the state of the variable *alarmState*. This variable is assigned ON when the gas detector or the over temperature detector is activated and is assigned OFF when the B1 USER button is pressed.



NOTE: Strictly speaking, a Boolean variable has only two valid states: *true* or *false*. However, for practical purposes, we will use ON and OFF to refer to the values *true* and *false* of the variable *alarmState*.

It is worth noting that the *while(true)* instruction used to indicate the beginning of the repetitive block is based on the idea that what is inside the braces of the *while* statement should be repeated until the condition inside the parentheses of the *while* becomes false. Given that it is written *while(true)*, it will never be false. Therefore, the instructions inside the braces will be repeated forever.

Figure 1.13 shows the proposed main parts to implement the solution of this example. In the *Libraries, Definitions and Global Declarations and Initializations* (Figure 1.13 [a]), the *mbed.h* and *arm_book.lib.h* libraries are included. In the *Declarations and Initializations of the main()* function (Figure 1.13 [b]), the digital inputs *gasDetector*, *overTempDetector*, and *alarmOffButton* are declared and assigned to buttons D2, D3, and B1 USER, respectively, the digital output *alarmLed* is declared and assigned to LD1, and the variable *alarmState* is declared and initialized to OFF.

The *repetitive block* (Figure 1.13 [c]) is made up of three blocks: a block indicated in brown, which assigns ON to *alarmState* if *gasDetector* or *overTempDetector* is activated; a block indicated in orange, which assigns the value of *alarmState* to *alarmLed*; and a block indicated in violet, which assigns OFF to *alarmState* if the *alarmOffButton* is pressed.



NOTE: In this book names of variables are stylized using the lower camel case format, as in *alarmState*.

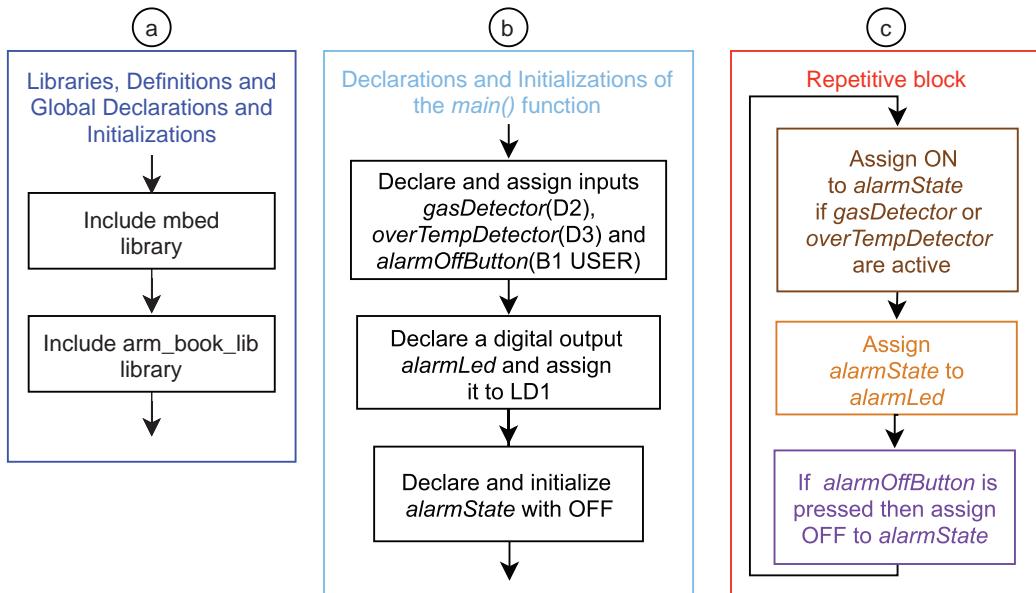


Figure 1.13 Main parts of the program of Example 1.3.

Figure 1.14 shows the details of the blocks that compose the repetitive block. Figure 1.14 (a) shows the block that assigns ON to `alarmState` if `gasDetector` or `overTempDetector` is active. Figure 1.14 (b) shows the details of the block that assigns `alarmState` to `alarmLed`. Figure 1.14 (c) shows the details of the block that assigns OFF to `alarmState` if `alarmOffButton` is pressed.

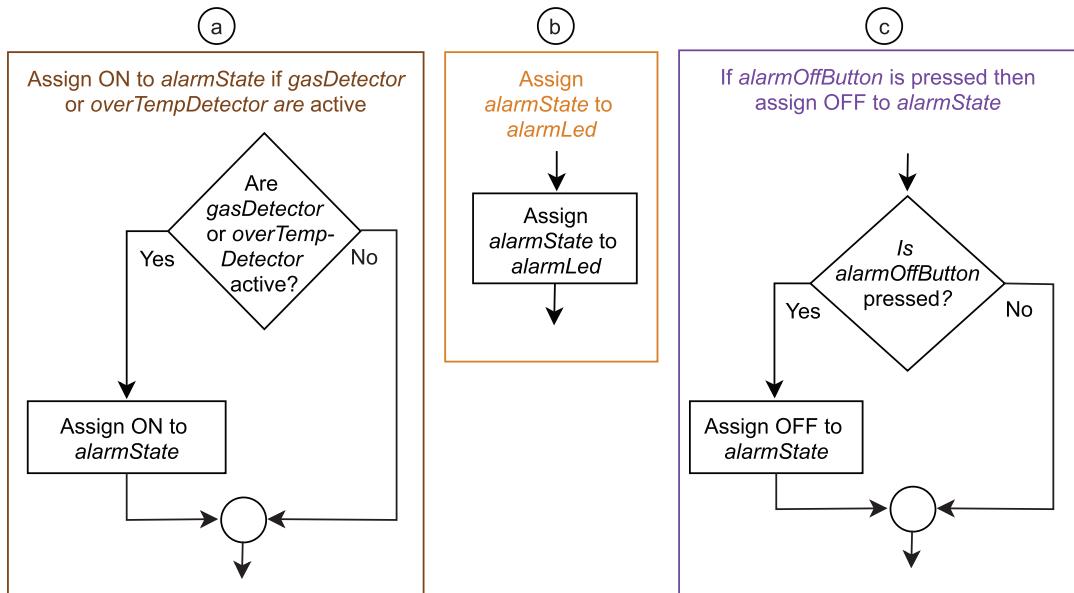


Figure 1.14 Details of the blocks that compose the repetitive block of Example 1.3.

Implementation of the Proposed Solution

In Code 1.3 the implementation of the proposed solution is presented. The parts corresponding to Figure 1.13 and Figure 1.14 are highlighted in color. An explanation of each part is included beside the code.



NOTE: In the code, “BUTTON1” is used to refer to the “B1 USER button” because it is defined this way in the mbed.h library. For the same reason, “LED1” is used to refer to “LD1”, such as in Examples 1.1 and 1.2.

```

1 #include "mbed.h"
2 #include "arm_book_lib.h"
3
4 int main()
5 {
6     DigitalIn gasDetector(D2);
7     DigitalIn overTempDetector(D3);
8     DigitalIn alarmOffButton(BUTTON1);
9
10    DigitalOut alarmLed(LED1);
11
12    gasDetector.mode(PullDown);
13    overTempDetector.mode(PullDown);
14
15    alarmLed = OFF;
16
17    bool alarmState = OFF;
18
19    while ( true ) {
20
21        if ( gasDetector || overTempDetector ) {
22            alarmState = ON;
23        }
24
25        alarmLed = alarmState;
26
27        if ( alarmOffButton ) {
28            alarmState = OFF;
29        }
30    }
31 }
```

Include the libraries "mbed.h" and "arm_book_lib.h"

- Digital input objects named *gasDetector*, *overTempDetector* and *alarmOffButton*, are declared and assigned to D2, D3, and BUTTON1 (B1 USER), respectively.
- Digital output object named *alarmLed* is declared and assigned to LED1.
- *gasDetector* and *overTempDetector* are configured with internal pull-down resistors.
- *alarmLed* is assigned OFF.
- *alarmState* is declared and assigned OFF.

Do forever loop:

- If *gasDetector* or *overTempDetector* are active, assign *alarmState* with ON.
- Assign *alarmLed* with *alarmState*.
- If *alarmOffButton*, assign *alarmState* with OFF.

Code 1.3 Implementation of Example 1.3.



NOTE: It is not necessary to enable an internal pull-down resistor for the B1 User button. This is because this button is already connected to an external pull-down resistor placed in the NUCLEO board, as well as to other elements that are used to reduce the *electrical noise* of the signal that is supplied to the microcontroller.

Proposed Exercises

1. How can the code be changed in such a way that LD1 is turned on at the beginning, is turned off when gas presence or over temperature is detected, and is turned on when the B1 USER button is pressed?
2. What should be modified in order to turn off the alarm by means of the button connected to D4?

Answers to the Exercises

1. It can be achieved by means of the changes shown in Table 1.7.
2. In line 16, BUTTON1 should be replaced by D4.

Table 1.7 Proposed modifications in the code in order to achieve the new behavior.

Lines in Code 1.3	New code to be used
17 bool alarmState = OFF	17 bool alarmState = ON
22 alarmState = ON;	22 alarmState = OFF;
28 alarmState = OFF;	28 alarmState = ON;

Example 1.4: Secure the Alarm Deactivation by Means of a Code

Objective

Introduce the AND and NOT operators.

Summary of the Expected Behavior

An LED must turn on when one or more buttons connected to the NUCLEO board are pressed, and it must be kept on until a combination of certain buttons is entered.

Test the Proposed Solution on the Board

Import the project “Example 1.4” using the URL available in [10], build the project, and drag the .bin file onto the NUCLEO board. Then press and release either of the buttons connected to D2 or D3. Look at the behavior of LD1. LD1 should turn on when any of the buttons connected to D2 and D3 are pressed and turn off only when the buttons connected to D4 and D5 are pressed together at the same time.

Discussion of the Proposed Solution

In general, the behavior of this code is similar to the code in Example 1.3, but in this example the way the alarm is turned off is different. In this case, a combination of certain buttons must be pressed simultaneously. To turn off the alarm, the buttons connected to D4 and D5 should be pressed while at

the same time the buttons connected to D6 and D7 should not be pressed. In C/C++, the logical AND is indicated by means of the `&&` operator and the logical operator NOT is indicated by means of the `!` operator.

Figure 1.15 shows the proposed main parts to implement the solution of this example. In the *Libraries, Definitions and Global Declarations and Initializations* (Figure 1.15 [a]), the `mbed.h` and `arm_book.lib` libraries are included. In the *Declarations and Initializations of the main() function* (Figure 1.15 [b]), the pins connected to D2, D3, D4, D5, D6 and D7 are configured as inputs, the pin connected to LD1 is configured as output, and the variable `alarmState` is declared and initialized to OFF.

The repetitive block (Figure 1.15 [c]) is made up of three blocks: a block indicated in brown, which assigns ON to `alarmState` if `gasDetector` or `overTempDetector` is activated; a block indicated in orange, which assigns the value of `alarmState` to `alarmLed`; and a block indicated in violet, which assigns OFF to `alarmState` if the buttons connected to D4 and D5 are pressed while the buttons connected to D6 and D7 are *not* pressed.

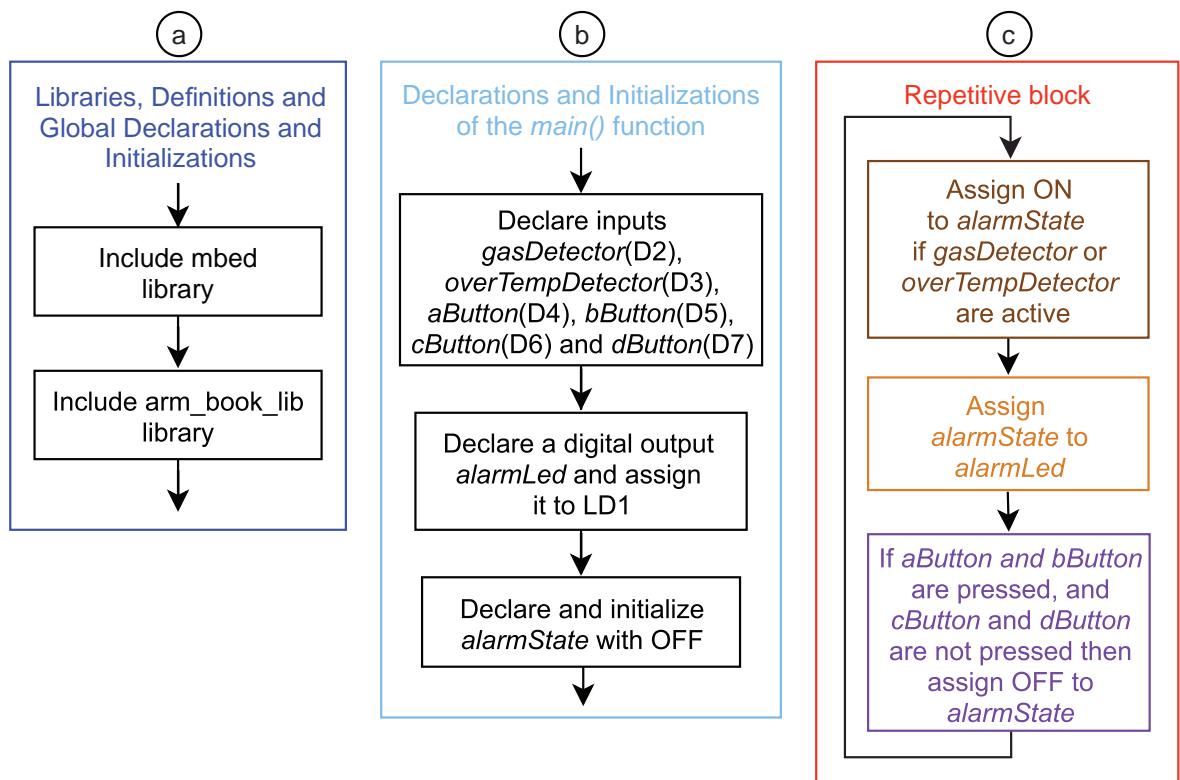


Figure 1.15 Main parts of the program of Example 1.4.

Figure 1.16 shows the details of the blocks that compose the repetitive block. In this case, only the block indicated in violet is included because the brown and orange blocks are the same as in Figure 1.16 (a) and Figure 1.16 (b), respectively. Figure 1.16 (c) shows the block that assigns OFF to *alarmState* if the buttons connected to D4 and D5 are pressed and the buttons connected to D6 and D7 are not pressed.

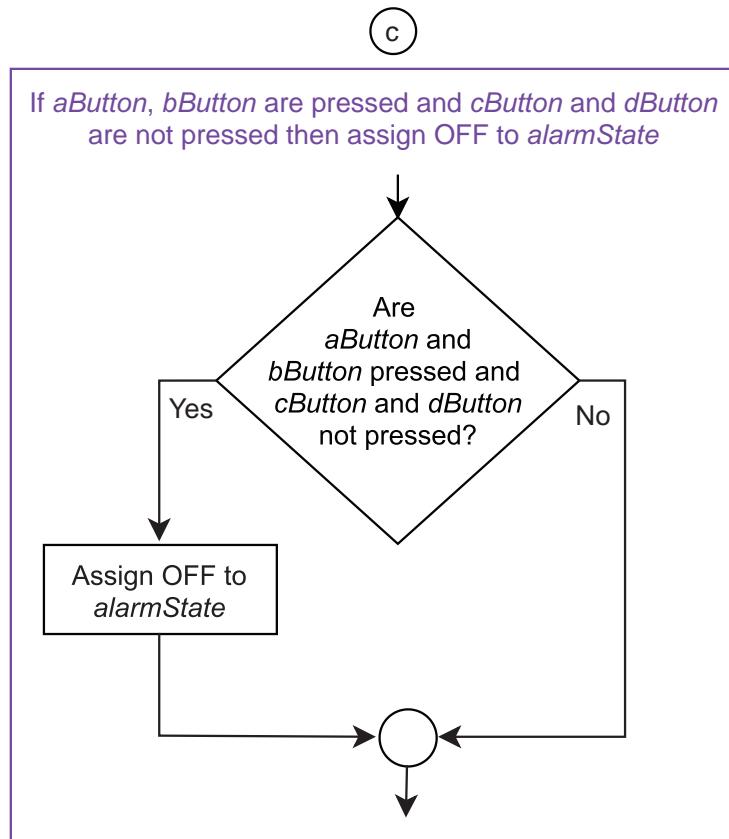


Figure 1.16 Details of the blocks that make up the repetitive block of Example 1.4.

Implementation of the Proposed Solution

In Code 1.4 the implementation of the proposed solution is presented. The parts corresponding to Figure 1.15 and Figure 1.16 are highlighted in different colors. An explanation of each part is also included beside the code.

```

1 #include "mbed.h"
2 #include "arm_book_lib.h"
3
4 int main()
5 {
6     DigitalIn gasDetector(D2);
7     DigitalIn overTempDetector(D3);
8     DigitalIn aButton(D4);
9     DigitalIn bButton(D5);
10    DigitalIn cButton(D6);
11    DigitalIn dButton(D7);
12
13    DigitalOut alarmLed(LED1);
14
15    gasDetector.mode(PullDown);
16    overTempDetector.mode(PullDown);
17    aButton.mode(PullDown);
18    bButton.mode(PullDown);
19    cButton.mode(PullDown);
20    dButton.mode(PullDown);
21
22    alarmLed = OFF;
23
24    bool alarmState = OFF;
25
26    while (true) {
27
28        if ( gasDetector || overTempDetector ) {
29            alarmState = ON;
30        }
31
32        alarmLed = alarmState;
33
34        if ( aButton && bButton && !cButton && !dButton ) {
35            alarmState = OFF;
36        }
37    }
38 }

```

Include the libraries "mbed.h" and "arm_book_lib.h"

- Digital input objects named *gasDetector*, *overTempDetector* and *aButton* to *dButton*, are declared and assigned to D2 to D7, respectively.
- Digital output object named *alarmLed* is declared and assigned to LED1.
- *gasDetector*, *overTempDetector* and *aButton* to *dButton* are configured with internal pull-down resistors.
- *alarmLed* is assigned OFF.
- *alarmState* is declared and assigned OFF.

Do forever loop:

- If *gasDetector* or *overTempDetector* is pressed, assign *alarmState* with ON.
- Assign *alarmLed* with *alarmState*.
- If *aButton* and *bButton* are pressed, and neither *cButton* nor *dButton* are pressed, assign *alarmState* with OFF.

Code 1.4 Implementation of Example 1.4.

Proposed Exercises

1. How can the code be changed in such a way that the alarm is turned off by means of pressing the buttons connected to D6 and D7, while the buttons connected to D4 and D5 are not pressed?
2. How can the code be changed in such a way that the alarm is turned off by means of pressing the buttons connected to D4, D5, and D6, while the button connected to D7 is not pressed?

Answers to the Exercises

1. It can be achieved by means of the changes in Table 1.8.
2. It can be achieved by means of the changes in Table 1.9.

Table 1.8 Proposed modifications in the code in order to achieve the new behavior.

Lines in Code 1.4	New code to be used
34 if (aButton && bButton && !cButton && !dButton)	34 if (!aButton && !bButton && cButton && dButton)

Table 1.9 Proposed modifications in the code in order to achieve the new behavior.

Lines in Code 1.4	New code to be used
34 if (aButton && bButton && !cButton && !dButton)	34 if (aButton && bButton && cButton && !dButton)

Example 1.5: Block the System when Five Incorrect Codes are Entered

Objective

Introduce nested *ifs* and the usage of non-Boolean variables to count the number of iterations.

Summary of the Expected Behavior

If five wrong passwords are introduced, the system is blocked.

Test the Proposed Solution on the Board

Import the project “Example 1.5” using the URL available in [10], build the project, and drag the *.bin* file onto the NUCLEO board. Then press and release the buttons connected to D2 or to D3 and look at the behavior of LD1. LD1 should turn on. If buttons A, B, and Enter (D4, D5, and B1 USER Button) are pressed, LD1 will turn off. Again, press and release the buttons connected to D2 or to D3. LD1 should turn on. If another combination of buttons is pressed (for instance A and B1 USER) the LD3 (Incorrect code) will turn on. A new combination of buttons can be tried after all four buttons connected to D4–D7 are pressed simultaneously. The fifth time that an incorrect combination of buttons is entered, LD2 (System blocked) will turn on, indicating that the system has been blocked. Press the reset button to reset the NUCLEO board and turn off the System blocked LED.

A summary of the buttons that have to be pressed in each case is shown in Table 1.10.

Table 1.10 Summary of the smart home system buttons that should be pressed in each case.

Functionality	Buttons that should be pressed	Corresponding DigitalIn
To test if a given code turns off the Alarm LED	Any of A, B, C, and/or D + Enter	(D4, D5, D6, D7) + BUTTON1
The correct code that turns off the Alarm LED	A + B + Enter	D4 + D5 + BUTTON1
To turn off the Incorrect code LED and enable a new attempt to turn off the Alarm LED	A + B + C + D	D4 + D5 + D6 + D7
To turn off the System Blocked LED (that is turned on after entering five incorrect passwords)	No buttons available (power should be removed or B2 RESET button pressed)	

Discussion of the Proposed Solution

In Example 1.4 there was no limit to the number of incorrect combinations of buttons that could be entered. In this example, the system is blocked when five incorrect combinations are entered. In order to count the number of incorrect combinations entered, *non-Boolean* variables are introduced. Non-Boolean variables can be used to store integer or non-integer numbers depending on the specific type used, as discussed below.

Figure 1.17 shows the proposed main parts to implement the solution of this example. In the *Libraries, Definitions and Global Declarations and Initializations* (Figure 1.17 [a]), the *mbed.h* and *arm_book.lib.h* libraries are included. In the *Declarations and Initializations of the main() function* (Figure 1.17 [b]), several inputs are declared and assigned to D2, D3, D4, D5, D6, D7, and BUTTON1; several outputs are declared and assigned to LD1, LD2, and LD3; the variable *alarmState* is declared and initialized to OFF; and the variable *numberOfIncorrectCodes* is declared and initialized to 0.

The repetitive block (Figure 1.17 [c]) is made up of three blocks: a block indicated in brown, which assigns ON to *alarmState* if *gasDetector* or *overTempDetector* is active; a block indicated in orange, which assigns the value of *alarmState* to *alarmLed*; and a block indicated in violet. In the violet block, if *numberOfIncorrectCodes* has reached five, the system is blocked. If *numberOfIncorrectCodes* has not reached five, then it checks if the user wants to enter a new code to turn off the alarm. If so, it checks if the entered code is correct in order to turn off the alarm LED. The *incorrectCodeLed* (LD2) is turned on if an incorrect code is entered.

Figure 1.18 shows the details of the blocks that comprise the repetitive block. In this case, only the block indicated in violet is included because the brown and orange blocks are the same as in Figure 1.14 (a) and Figure 1.14 (b), respectively.

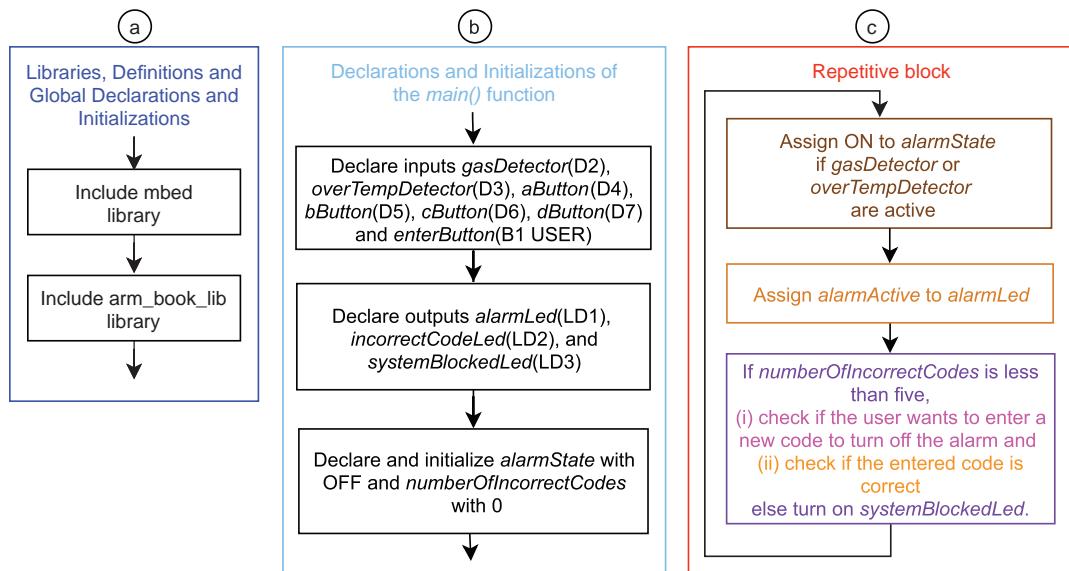


Figure 1.17 Main parts of the program of Example 1.5.

Figure 1.18 shows that the first step is to evaluate if *numberOfIncorrectCodes* is less than five. If five or more incorrect codes have been entered, then the system is blocked, and the System blocked LED is turned on. Buttons are not checked any more, and the only way to unblock the system is to turn off the power supply.

If *numberOfIncorrectCodes* is less than five, a check is made to see if buttons A, B, C, and D are all pressed at the same time, while the Enter button is not pressed, which means that the user wants to enter a code to turn off the alarm. In this case, the Incorrect code LED is turned off.

Next, it is assessed whether the following conditions are all accomplished at the same time: i) the Enter button is being pressed, ii) the Incorrect code LED is OFF, iii) the alarm is ON. In this way, it is determined if the user wants to enter a code (recall Table 1.10), and that a code can be entered (i.e., Incorrect code LED is off) and that a code must be entered (the alarm is ON).

If that is the case, to turn off the Alarm LED, buttons A and B should be pressed, while buttons C and D are not pressed. In this case, OFF is assigned to *alarmState* and *numberOfIncorrectCodes* is set to 0.

If any other combination of buttons A, B, C, and D is entered then the Incorrect code LED is turned on and *numberOfIncorrectCodes* is incremented by one.



NOTE: It is important to mention that this is the first time in this book that an *if* condition is evaluated inside of another *if* condition. This grouping of *if* statements is known as *nested ifs*.



WARNING: Two *if* conditions can also be evaluated one after the other, no matter the result of the first *if* condition. In this case, the structure does not correspond to a nested *if*.

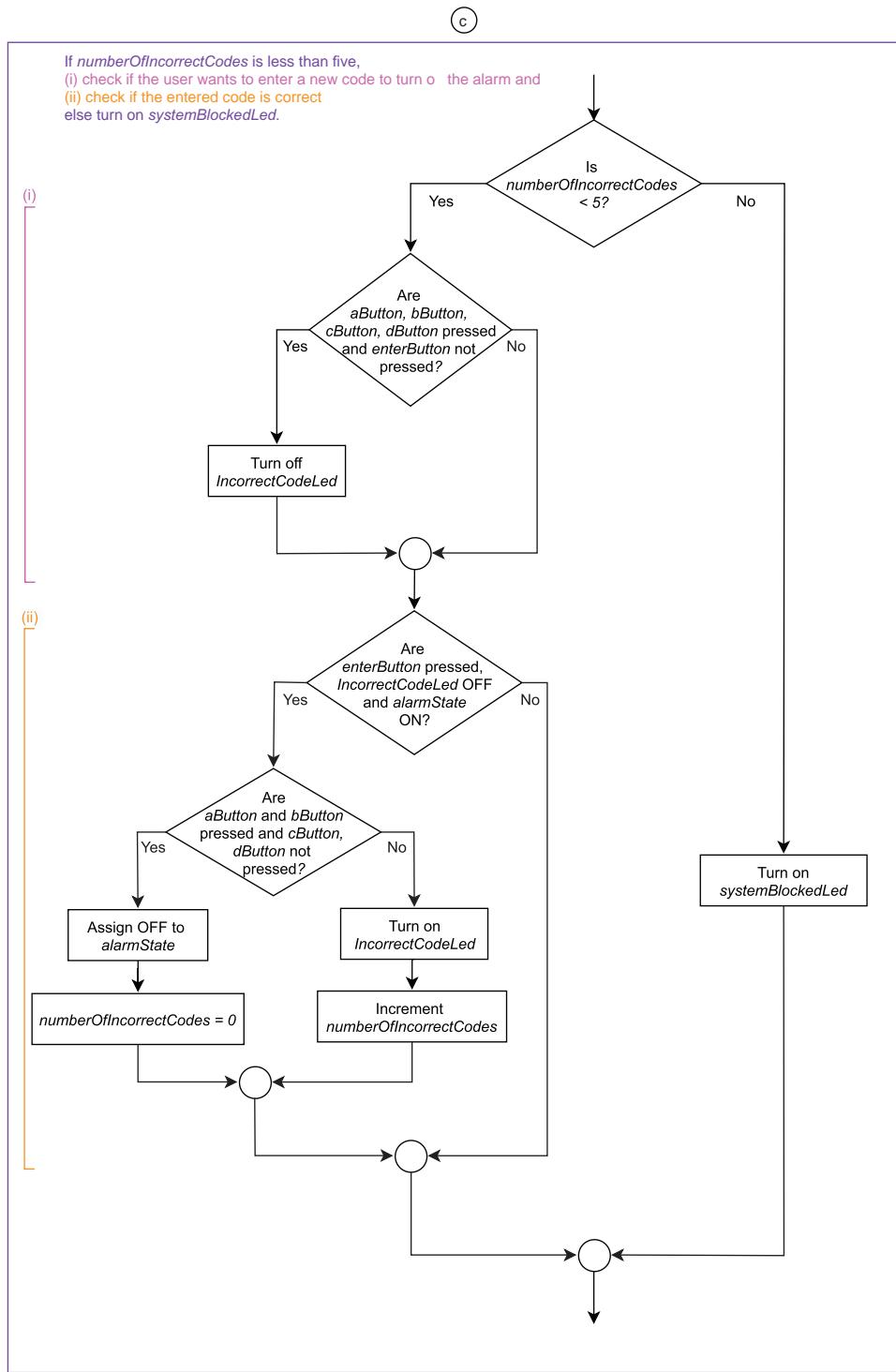


Figure 1.18 Details of the blocks that make up the repetitive block of Example 1.5.

Implementation of the Proposed Solution

In Code 1.5 the implementation of the proposed solution is presented. The parts corresponding to Figure 1.17 and Figure 1.18 are highlighted in different colors. An explanation of each part is also included beside the source code.

It is important to emphasize that all the considerations regarding the nested if discussed above correspond to lines 40 to 58 in Code 1.5. These lines are not discussed here because their behavior has already been discussed in detail in the explanation regarding Figure 1.18.

```

1 #include "mbed.h"
2 #include "arm_book_lib.h" [Include the libraries "mbed.h"
3 and "arm_book_lib.h"]
4 int main()
5 {
6     DigitalIn enterButton(BUTTON1);
7     DigitalIn gasDetector(D2);
8     DigitalIn overTempDetector(D3);
9     DigitalIn aButton(D4);
10    DigitalIn bButton(D5);
11    DigitalIn cButton(D6);
12    DigitalIn dButton(D7);
13
14    DigitalOut alarmLed(LED1);
15    DigitalOut incorrectCodeLed(LED3);
16    DigitalOut systemBlockedLed(LED2);
17
18    gasDetector.mode(PullDown);
19    overTempDetector.mode(PullDown);
20    aButton.mode(PullDown);
21    bButton.mode(PullDown);
22    cButton.mode(PullDown);
23    dButton.mode(PullDown);
24
25    alarmLed = OFF;
26    incorrectCodeLed = OFF;
27    systemBlockedLed = OFF;
28
29    bool alarmState = OFF;
30    int numberOfIncorrectCodes = 0;
31
32    while (true) {
33
34        if ( gasDetector || overTempDetector ) {
35            alarmState = ON;
36        }
37
38        alarmLed = alarmState;
39
40        if ( numberOfIncorrectCodes < 5 ) {
41
42            if ( aButton && bButton && cButton && dButton && !enterButton ) {
43                incorrectCodeLed = OFF;
44            }
45
46            if ( enterButton && !incorrectCodeLed && alarmState ) {
47                if ( aButton && bButton && !cButton && !dButton ) {
48                    alarmState = OFF;
49                    numberOfIncorrectCodes = 0;
50                } else {
51                    incorrectCodeLed = ON;
52                    numberOfIncorrectCodes = numberOfIncorrectCodes + 1;
53                }
54            }
55        } else {
56            systemBlockedLed = ON;
57        }
58    }
59}

```

Include the libraries "mbed.h" and "arm_book_lib.h"

- Digital input objects named `gasDetector`, `overTempDetector` and `aButton` to `dButton`, are declared and assigned to D2 to D7, respectively.
- Digital output objects named `alarmLed`, `incorrectCodeLed`, and `systemBlockedLed` are declared and assigned to LED1, LED2 and LED3, respectively.
- `gasDetector`, `overTempDetector` and `aButton` to `dButton` are configured with internal pull-down resistors.
- `alarmLed`, `incorrectCodeLed`, and `systemBlockedLed` are assigned OFF.
- `alarmState` is declared and assigned OFF.
- `numberOfIncorrectCodes` is declared and assigned 0 (zero).

Do forever loop:

- If `gasDetector` or `overTempDetector` are active, assign `alarmState` with ON.
- Assign `alarmLed` with `alarmState`.
- If `numberOfIncorrectCodes` is less than five, implement the logic (discussed in the text) in order to determine the state of `incorrectCodeLed` and if `alarmState` must be assigned with OFF, and if `numberOfIncorrectCodes` must be set to 0 (zero) or increased by 1.
- If `numberOfIncorrectCodes` is greater than or equal to five, assign `systemBlockedLed` with ON.

Code 1.5 Implementation of Example 1.5.

Proposed Exercise

- How can the code be changed in such a way that the system is blocked after three incorrect codes are entered?

Answer to the Exercise

- It can be achieved by means of the change in Table 1.11.

Table 1.11 Proposed modification in the code in order to achieve the new behavior.

Line in Code 1.5	New code to be used
40 if (numberOfIncorrectCodes < 5)	40 if (numberOfIncorrectCodes < 3)

1.3 Under the Hood

1.3.1 Brief Introduction to the Cortex-M Processor Family and the NUCLEO Board

In this chapter, many programs were developed using the NUCLEO board, provided with the STM32F429ZIT6U microcontroller. This microcontroller is manufactured by STMicroelectronics [13] using a Cortex-M4 processor designed by Arm Ltd. [14]. In order to support some concepts that were introduced through this chapter, as well as concepts that will be presented in the following chapters, in this subsection a brief introduction to Cortex-M processors and some details on the NUCLEO board are provided.

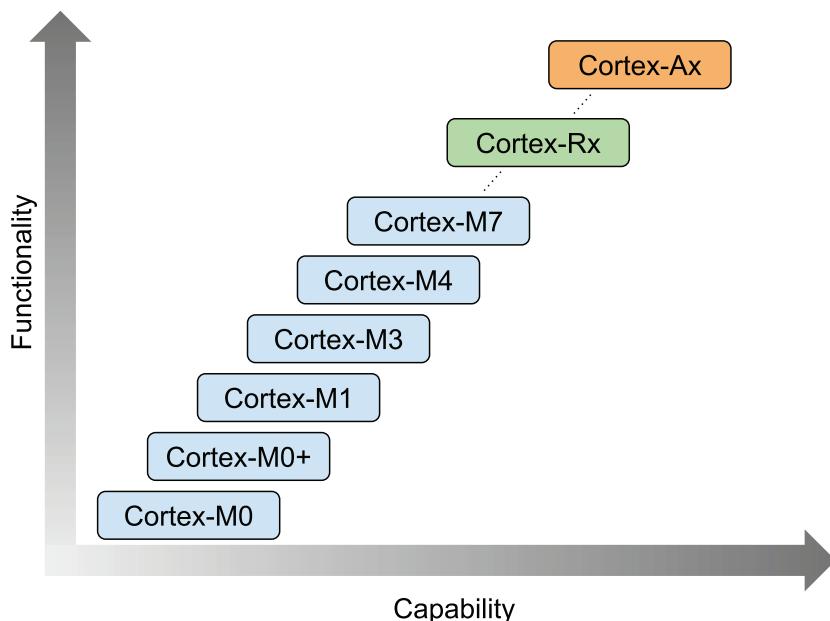


Figure 1.19 Simplified diagram of the Cortex processor family.

In Figure 1.19, a simplified diagram of the Cortex family of processors is shown. The Cortex-M processors are the most energy-efficient embedded devices of the family and have the lowest cost; therefore, they are typically used in microcontrollers and sensors. The Cortex-Rx are robust real-time performance processors, ideal for automotive and safety-critical applications. The Cortex-Ax processors have supreme performance and are used in smartphones, computers, and high-end microprocessors.



NOTE: The Cortex-M family currently has more than ten members (a list is available from [15]), and only six of them are shown in the diagram. In addition, the “Cortex-Rx” and “Cortex-Ax” frames represent whole families, which at the time of writing have half a dozen members and over twenty members, respectively.

In this Under the Hood section, only the Cortex-M0, Cortex-M3, and Cortex-M4 processors are analyzed, because they are more likely to be the processors used in the microcontrollers for elementary embedded systems projects. These processors are primarily focused on delivering highly deterministic behavior in a wide range of power-sensitive applications.

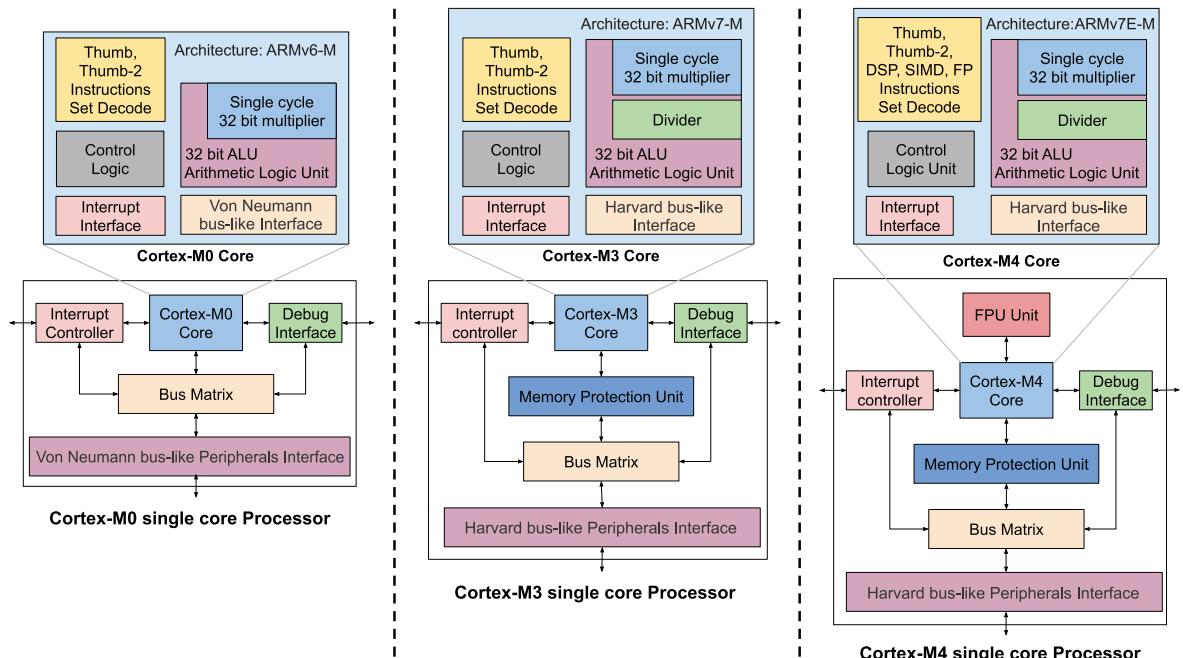


Figure 1.20 Simplified diagram of the Cortex M0, M3, and M4 processors, and details of the corresponding cores.

In Figure 1.20, diagrams of the Cortex-M0, M3, and M4 processors are shown, as well as some details of their corresponding cores. It can be appreciated how there are some similarities among the processors, such as interruption handling (a topic that will be discussed later in this book), and some differences, for example:

- The Cortex-M0 is Von Neumann bus-like (one bus for data/instruction), while the Cortex-M3 and M4 are Harvard bus-like (data and instruction in separate buses), which allows for faster communication.

- The Cortex-M3 and M4 processors have a Memory Protection Unit, which allows safer operation.
- The Cortex-M4 has a larger set of instructions and has a Floating Point Unit (FPU), which speeds up certain calculations because the FPU is used instead of using a multitude of elementary operations.

To indicate the complexity of the different cores, Figure 1.21 shows the set of instructions that each of the cores can process. This is called the *Instruction Set Architecture* (ISA). It can be seen that the Cortex-M0 core implements a reduced number of instructions, while the Cortex-M3 core handles those instructions and incorporates many more instructions, and the Cortex-M4 incorporates even more. Instructions added in the Cortex-M7 are shown in Figure 1.21 to stress that more advanced and powerful cores are available in the Cortex-M family, as shown in [16]. Those cores are not used in this book, hence are not discussed here.

VABS	VADD	VCMP	VCMPE	VCVT	VCVTR	VCVTB	VCVTT	VDIV	VCVTA
PKHBT	PKHTB	QADD	QADD16	QADD8	QASX	QDADD	QDSUB		VCVTN
QSAX	QSUB	QSUB16	QSUB8	SADD16	SADD8	SASX	SEL		VCVTP
SHADD16	SHADD8	SHASX	SHSAX	SHSUB16	SHSUB8	SMLABB	SMLABT		VFMAs
SMLATB	SMLATT	SMLAD	SMLADX	SMLALBB	SMLALBT	SMLALTB	SMLALTT		VFNMA
SMLALD	SMLALDX	SMLAWB	SMLANT	SMLSD	SMLSX	SMLSLD	SMLSIX		VFNMS
ADC	ADD	ADR	AND	ASR	B	SMMLA	SMLLR		VRINTA
BFC	BFI	BIC	CBNZ	CBZ	CDP	SMMLS	SMLLSR		VRINTN
CLREX	CLZ	CMN	CMP	DBG	EOR	SMMUL	SMMULR		VRINTP
LDC	LDC2	LDMIA	LDMDB	LDR	LDRB	SMUAD	SMUADX		VRINTM
LDRBT	LDRD	LDREX	LDREXB	LDREXH	LDRH	SMULBB	SMULBT		VRINTX
LDRHT	LDRSB	LDRSBT	LDRSH	LDRT	LSL	SMULTB	SMULTT		VRINTZ
LSR	MCR	MCR2	MCRR	MCRR2	MLA	SMULWB	SMULWT		VRINTR
MLS	MOV	MOVT	MRC	MRC2	MRRC	SMUSD	SMUSDx		VSEL
MRRC2	MUL	MVN	NOP	ORN	ORR	SSAT16	SSAX		
PLD	PLI	POP	PUSH	RBIT	REV	SSUB16	SSUB8		
REV16	REVSH	ROR	RRX	RSB	SBC	SXTAB	SXTAB16		
SBFX	SDIV	SEV	SMLAL	SMULL	SSAT	SXTAH	UADD16		
ADC	ADD	ADR	AND	ASR	B	STC	STC2	UADD8	VSQRT
BIC	BKPT	BL	BLX	BX		STMDB	STR	UASX	VSTM
CMN	CMP	CPS	DMB	EOR		STRBT	STRD	UHADD16	VSTR
DSB	ISB	LDMIA	LDR			STREXB	STREXH	UHSUB8	VSUB
LDRB	LDRH	LDRSB	LDRSH	LSL		STRHT	STRT	UMAAL	
MOV	MRS	MSR	MUL			SXTB	SXTH	UQADD8	
MVN	NOP	ORR	POP	PUSH		TBH	TEQ	UQASX	
REV16	REVSH	ROR	RSB	SBC		UBFX	UDF	UQSUB8	
STMIA	STR	STRB	STRH	SUB		UMLAL	UMULL	USADA8	
SXTB	SXTH	TST	UDF	UXTB		UXTB	UXTH	USADA8	
WFE	WFI	YIELD	YIELD			WFE		USAT16	
								UXTAB16	
								UXTAH	
								UXTB16	

General data processing I/O control tasks	Advanced data processing bit field manipulations	DSP (SIMD, fast MAC)	Floating Point	Floating Point FPv5
--	---	-------------------------	-------------------	------------------------

Figure 1.21 Arm Cortex M0, M3, and M4 Instruction Set Architecture (ISA).

The instructions shown in Figure 1.21 are in assembly language (also called assembler language), which is not as easy to read and write for programmers as the C/C++ language. Thus, a program called a *compiler* is used to automatically translate C/C++ language code into the corresponding assembly language code. The compiler also verifies that the C/C++ language code complies with the language rules defined in the standards. If there is a syntax issue, the compiler cannot generate the assembly code and an error is indicated to the programmer. Keil Studio Cloud includes all these features.



TIP: Don't worry about needing to learn assembly language, it is not used in most elementary projects.

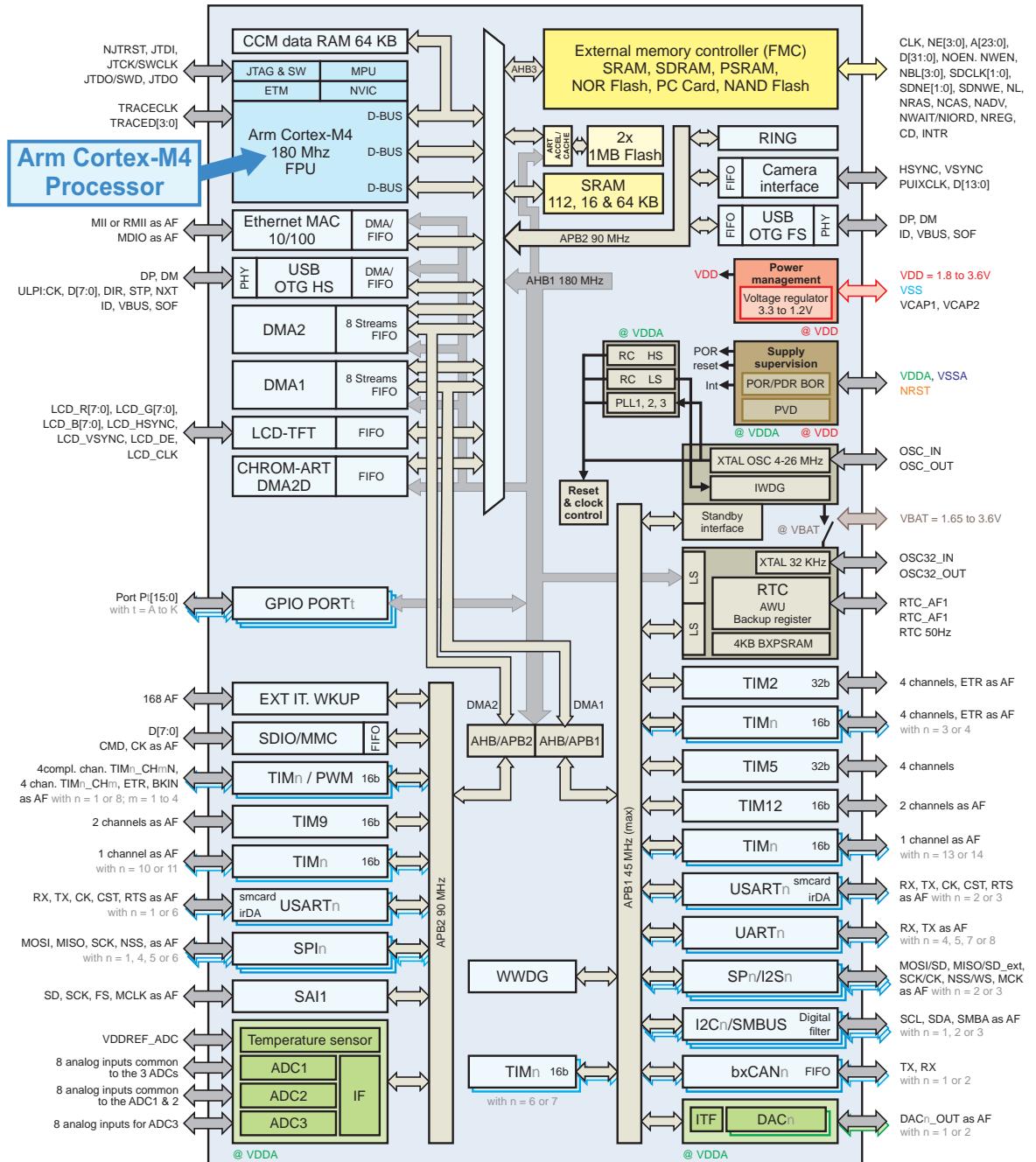


Figure 1.22 STM32F429ZI block diagram made using information available from [9].

The STM32F429ZIT6U microcontroller includes a Cortex-M4 processor, as shown in Figure 1.22. It can be appreciated that, beyond the processor, the microcontroller includes other peripherals such as communication cores (ethernet, USB, UART, etc.), memory, timers, and GPIO (General Purpose Input Output) ports.

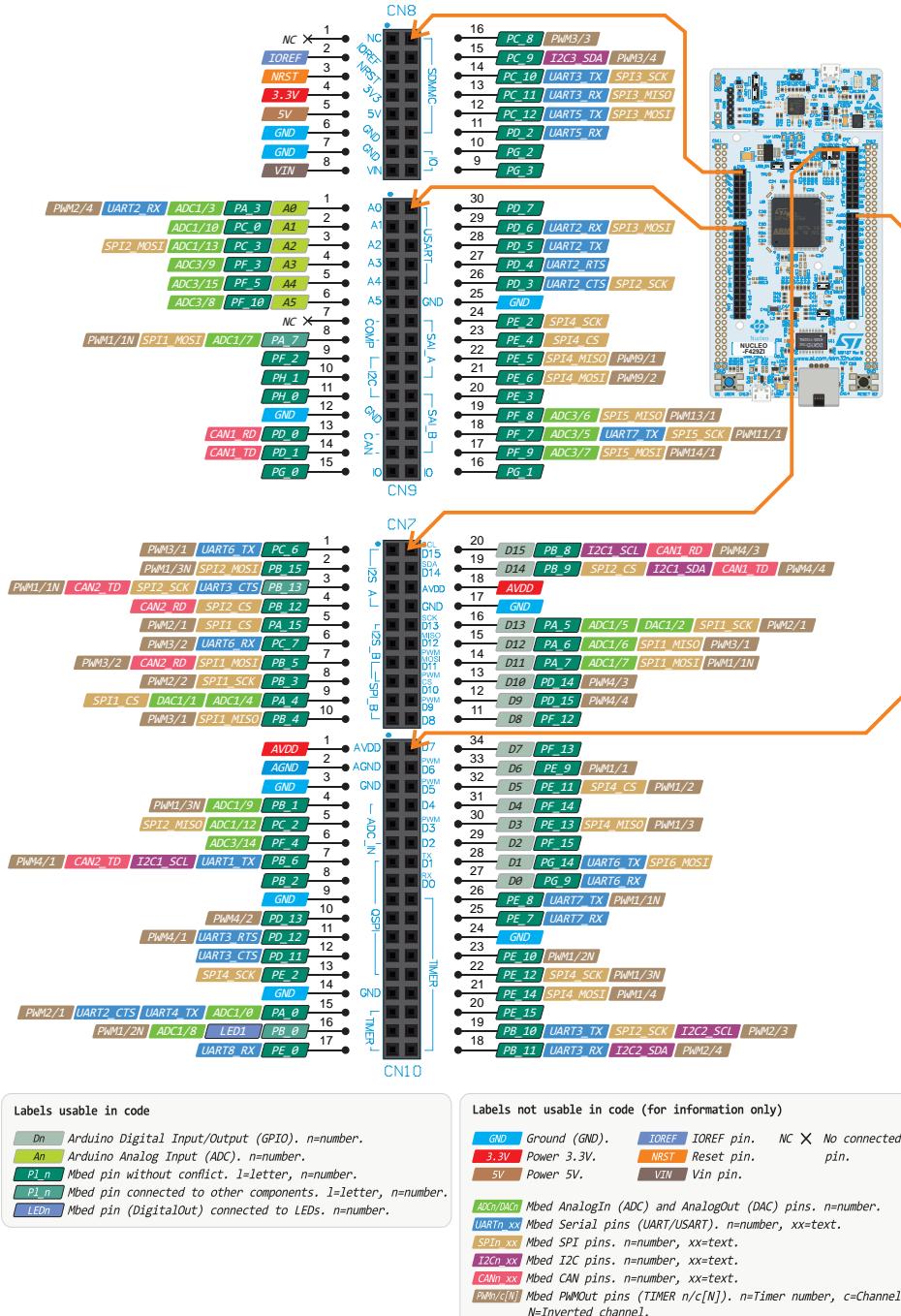


Figure 1.23 ST Zio connectors of the NUCLEO-F429ZI board.

Figure 1.23 shows how different elements of the STM32F429ZIT6U microcontroller are mapped to the Zio and Arduino-compatible headers of the NUCLEO-F429ZI board. Some other elements are mapped to the CN11 and CN12 headers of the NUCLEO-F429ZI board, as will be discussed in upcoming chapters. Further information on these headers is available from [17].

In this chapter, buttons were connected to the NUCLEO board using pins D2 to D7. From Figure 1.23, it can be seen that those digital inputs can also be referred to as PF_15, PE_13, P_14, PE_11, PE_9, and PF_13, respectively. Throughout this book, many pins of the ST Zio connectors will be used, and they will be referred to in the code using the names shown in Figure 1.23.



WARNING: Keil Studio Cloud translates the C/C++ language code into assembly code while considering the available resources of the target board. For this reason, the reader should be very careful to use only pin names that are shown in Figure 1.23.

From the above discussion, it is possible to derive the hierarchy that is represented in Figure 1.24. It should be noted that a given Arm processor can be used by different microcontroller manufacturers, and a given microcontroller can be used in different development boards or embedded systems.

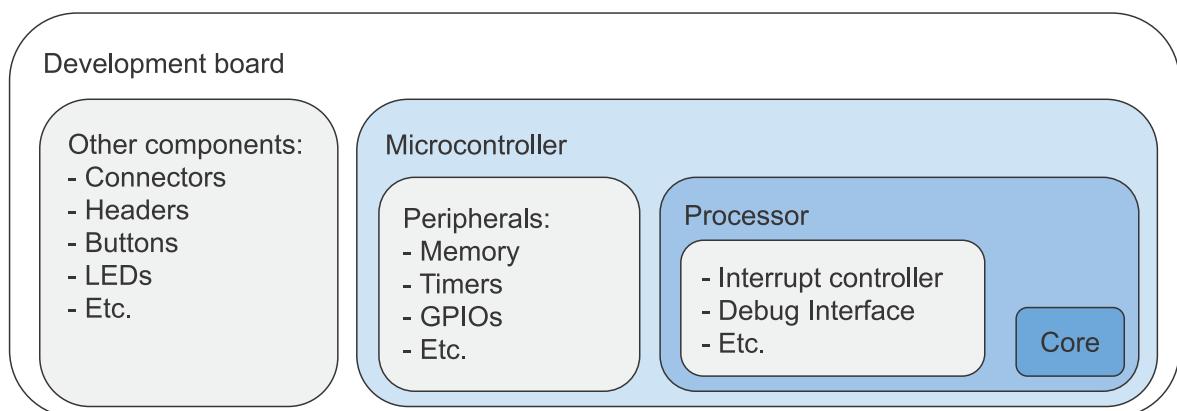


Figure 1.24 Hierarchy of different elements introduced in this chapter.



NOTE: A microcontroller may have one or more processors, while a processor may have one or more cores. A microprocessor consists of the processor, named in this context as the Central Processing Unit (CPU), and uses an external bus to interface with memory and peripherals.

Proposed Exercise

1. How can the code of Example 1.5 be modified in order to use the alternative names of D2 to D7?

Answer to the Exercise

1. It can be achieved by means of the changes shown in Table 1.12.

Table 1.12 Proposed modifications in the code in order to use the alternative names of D2 to D7.

Lines in Code 1.5	New code to be used
2 DigitalIn gasDetector(D2);	2 DigitalIn gasDetector(PF_15);
3 DigitalIn overTempDetector(D3);	3 DigitalIn overTempDetector(PE_13);
4 DigitalIn aButton(D4);	4 DigitalIn aButton(PF_14);
5 DigitalIn bButton(D5);	5 DigitalIn bButton(PE_11);
6 DigitalIn cButton(D6);	6 DigitalIn cButton(PE_9);
7 DigitalIn dButton(D7);	7 DigitalIn dButton(PF_13);

1.4 Case Study

1.4.1 Smart Door Locks

In this chapter, we implemented the functionality to turn on an alarm based on the activation of sensors that were represented by means of buttons. A code was also implemented. A brief view of a commercial “smart door lock” built with Mbed and containing some of these features can be found in [18]. A representation of the system is shown in Figure 1.25.

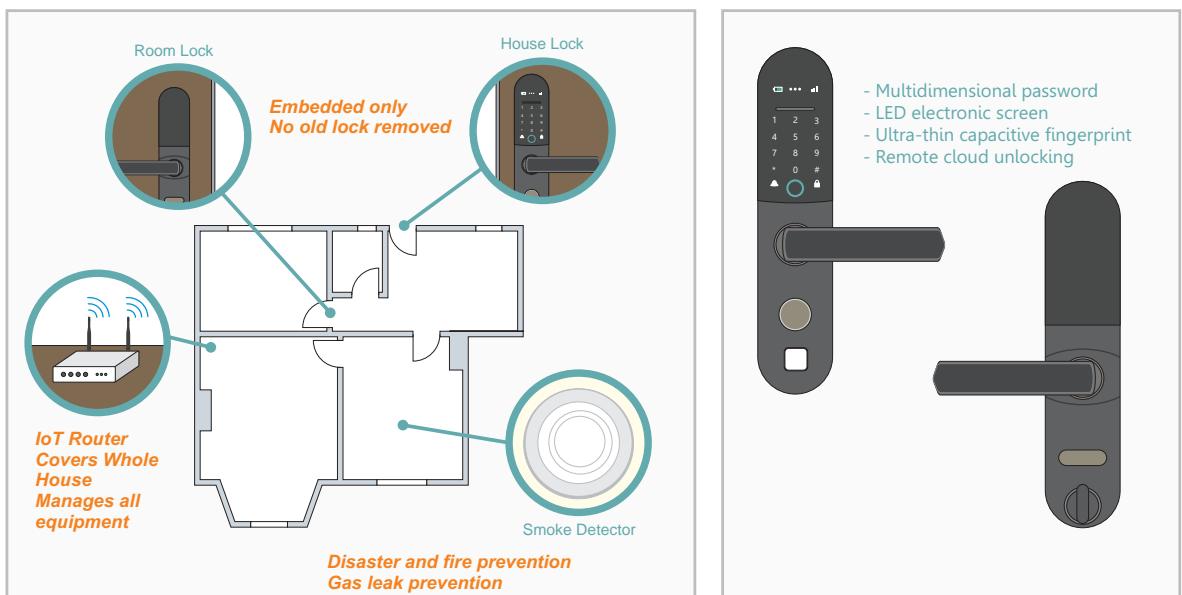


Figure 1.25 “Smart door locks” built with Mbed contains elements introduced in this chapter.

In Figure 1.25 it can be seen that the “smart door lock” has a smoke detector (similar to the gas detector mentioned in this chapter) and a locking system that employs passwords. The system was developed using Mbed OS and is based on a Cortex-M4 core. It is important to highlight that the STM32 microcontroller of the NUCLEO board is also based on a Cortex-M4 core.

Proposed Exercises

1. Are there buttons in Figure 1.25? Where are they located?
2. How is the password entered? Is it like in Example 1.4?

Answers to the Exercises

1. A keypad to enter the password can be seen in Figure 1.25, but there are no switches under the numbers. Instead, ultra-thin capacitive fingerprints are used.
2. The password is entered by pressing the numbers one after the other (not simultaneously like in Example 1.4). Later in this book, it will be explained how to implement a sequential reading of a matrix keypad.

References

- [1] “NUCLEO-F429ZI - STMicroelectronics”. Accessed July 9, 2021.
<https://os.mbed.com/platforms/ST-Nucleo-F429ZI/>
- [2] “Introduction - Mbed OS 6 | Mbed OS 6 documentation”. Accessed July 9, 2021.
<https://os.mbed.com/docs/mbed-os/v6.12/introduction/index.html>
- [3] “Arm Keil | Cloud-based Development Tools for IoT, ML and Embedded”. Accessed July 9, 2021.
<https://www.keil.arm.com/>
- [4] “Log In | Mbed”. Accessed July 9, 2021.
<https://os.mbed.com/account/login/>
- [5] “Keil Studio”. Accessed July 9, 2021.
<https://studio.keil.arm.com/>
- [6] “Documentation - Arm Developer”. Accessed July 9, 2021.
<https://developer.arm.com/documentation>
- [7] “Arm Mbed Studio”. Accessed July 9, 2021.
<https://os.mbed.com/studio/>
- [8] “STM32CubeIDE - Integrated Development Environment”. Accessed July 9, 2021.
<https://www.st.com/en/development-tools/stm32cubeide.html>
- [9] “STM32F429ZI - High-performance advanced line, Arm Cortex-M4”. Accessed July 9, 2021.
<https://www.st.com/en/microcontrollers-microprocessors/stm32f429zi.html>

- [10] "GitHub - armBookCodeExamples/Directory". Accessed July 9, 2021.
<https://github.com/armBookCodeExamples/Directory/>
- [11] "cplusplus.com - The C++ Resources Network". Accessed July 9, 2021.
<https://wwwcplusplus.com/>
- [12] "Indentation style - Wikipedia". Accessed July 9, 2021.
https://en.wikipedia.org/wiki/Indentation_style
- [13] "Home - STMicroelectronics". Accessed July 9, 2021.
<https://www.st.com/>
- [14] "Artificial Intelligence Enhanced Computing - Arm". Accessed July 9, 2021.
<https://www.arm.com/>
- [15] "Microprocessors Cores and Technology - Arm". Accessed July 9, 2021.
<https://www.arm.com/products/silicon-ip-cpu>
- [16] "ARM Cortex-M7: Bringing High Performance to the Cortex-M Processor Series. Accessed July 9, 2021.
http://www.armtechforum.com.cn/2014/bj/B-1_BringingHighPerformancetotheCortex-MProcessorSeries.pdf
- [17] "NUCLEO-F429ZI | Mbed". Accessed July 9, 2021.
<https://os.mbed.com/platforms/ST-Nucleo-F429ZI/#zio-and-arduino-compatible-headers>
- [18] "Smart door locks | Mbed". Accessed July 9, 2021.
<https://os.mbed.com/built-with-mbed/smart-door-locks/>

Chapter 2

Fundamentals of Serial
Communication

2.1 Roadmap

2.1.1 What You Will Learn

After you have studied the material in this chapter, you will be able to:

- Use a serial terminal to communicate between the NUCLEO board and a PC.
- Implement programs to use the UART of the microcontroller to share data between the NUCLEO board and a PC.
- Describe basic concepts about serial communications.

2.1.2 Review of Previous Chapter

In Chapter 1, the basic concepts of embedded systems programming were introduced. The reader was able to load different programs onto the microcontroller of the NUCLEO board, understand how they work, and modify their behavior using the Keil Studio Cloud application.

It was also explained how to expand the functionality of the NUCLEO board using the ST Zio connectors, a breadboard, jumper wires, and buttons. By means of these elements, a representation of a smart home system was implemented and was provided with different features, including a password code to deactivate the alarm once it was activated.

Finally, a brief introduction to the Cortex-M processor family and the NUCLEO board was presented.

2.1.3 Contents of This Chapter

This chapter will explain how to set up the NUCLEO board to communicate with a PC. This will be done by means of the *UART (Universal Asynchronous Receiver Transmitter)* of the STM32 microcontroller of the NUCLEO board, which is accessed via a USB connection with the PC.

The concepts *software maintainability, code modularization, functions, switch statements, for loops, define, and arrays*, among others, will be introduced in the examples for this chapter.

This chapter will also show how to use a *serial terminal* to visualize the information exchange between the PC and the NUCLEO board. By means of the serial terminal it will be possible to visualize different parameters and to configure and operate the smart home system being implemented with the NUCLEO board.

2.2 Serial Communication between a PC and the NUCLEO Board

2.2.1 Connect the Smart Home System to a PC

In this chapter, the smart home system will be connected to a PC, as shown in Figure 2.1. This will be done by means of serial communication using a USB cable. The aim of this setup is to monitor and configure the smart home system from the PC.

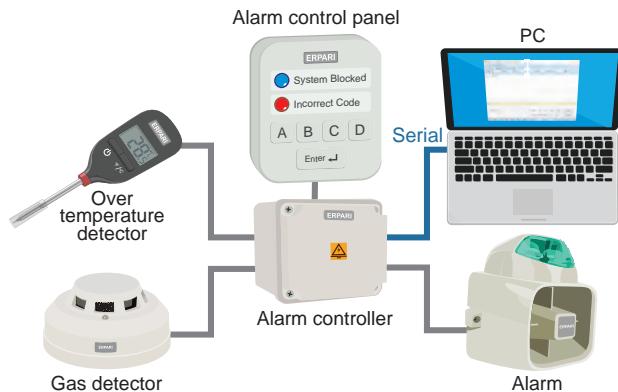


Figure 2.1 The smart home system is now connected to a PC.

It is important to notice that in this chapter nothing new has to be connected to the NUCLEO board in order to implement the smart home system shown in Figure 2.1, because the USB cable has already been connected in the previous chapter to load programs onto the microcontroller of the NUCLEO board, and because buttons are still used in this chapter to simulate the activation of the over temperature detector and the gas detector.



WARNING: to implement the examples in this chapter, and also to test the serial communication with the PC as explained in the following subsection, it is crucial to be sure that everything is connected just as in Figure 1.5.

In order to test the serial communication between the PC and the NUCLEO board, the .bin file of the program “Subsection 2.2.1” should be downloaded from the URL available in [1] and dragged onto the NUCLEO board.

To monitor and manage the serial communication data exchange between the PC and the NUCLEO board, usually a piece of software called a *serial terminal* is used. The reader may choose any serial terminal of their preference from the hundreds of options available on the internet.



NOTE: At the time of writing this book, the serial terminal embedded in Keil Studio Cloud does not support ST devices. When Keil Studio Cloud includes serial terminal support for ST devices, the corresponding instructions will be published in [1]. In that way the reader will be able to use Keil Studio Cloud to test the programs used in this book and will not have to download and install a separate serial terminal.



TIP: Given that the serial terminal will be used only for very basic operations, it is convenient for the reader to choose a serial terminal that is as simple as possible. In this way, just a few configurations will be needed. Tutorials about how to configure and use each serial terminal are available on the internet.



WARNING: It is crucial to ensure that in the serial terminal, the *baud rate* is configured to "115200", data bits is set to "8", parity to "none", stop bits to "1", handshaking to "none", and to send character <CR> (Carriage Return, '\r') when the Enter key is pressed. In the Under the Hood subsection, the meaning of these parameters is explained. It is also important to select in the serial terminal the "COM port" assigned by the operating system to the NUCLEO board.

To test if the serial terminal is working as expected, press the B2 RESET button of the NUCLEO board. A "Hello" message should appear on the serial terminal. In this way it is confirmed that the NUCLEO board is properly sending data and that this data is being received by the PC.



TIP: If the "Hello" message is not displayed on the serial terminal, then select another COM port and press the B2 RESET button. Repeat this operation with all the available COM ports until the "Hello" message appears on the serial terminal.



NOTE: This chapter does not discuss how each of the characters that are transferred between the PC and the NUCLEO board (for example, 'H', 'e', 'l', 'l', and 'o') are codified using the ASCII standard. This topic is addressed in Chapter 6.

To continue with the test, hold the B1 USER button of the NUCLEO board. The NUCLEO board should send the message "Button pressed" to the PC, and it should be displayed on the serial terminal.

To finish the test, release the B1 USER button of the NUCLEO board. The NUCLEO board sends the message "Button released" to the PC, and this message should be displayed on the serial terminal.



NOTE: It is important to note that the aim of this subsection is to test the serial communication between the NUCLEO board and the PC. For that reason, the details of the program being used in the testing and the technical background about serial communications are not presented in this subsection. These topics are addressed in the examples and in the Under the Hood section of this chapter.

NOTE: The text in the serial terminal is not erased until it is closed. Even if the B2 RESET button is pressed on the NUCLEO board, the previous messages will remain visible on the serial terminal.

2.2.2 Modularization of a Program into Functions

In Example 1.5, Code 2.1, shown below, was presented. If even more functionality is included in the smart home system, then the code will become longer, and its behavior will become difficult to understand.

```
1 #include "mbed.h"
2 #include "arm_book_lib.h"
3
4 int main()
5 {
6     DigitalIn enterButton(BUTTON1);
7     DigitalIn gasDetector(D2);
8     DigitalIn overTempDetector(D3);
9     DigitalIn aButton(D4);
10    DigitalIn bButton(D5);
11    DigitalIn cButton(D6);
12    DigitalIn dButton(D7);
13
14    DigitalOut alarmLed(LED1);
15    DigitalOut incorrectCodeLed(LED3);
16    DigitalOut systemBlockedLed(LED2);
17
18    gasDetector.mode(PullDown);
19    overTempDetector.mode(PullDown);
20    aButton.mode(PullDown);
21    bButton.mode(PullDown);
22    cButton.mode(PullDown);
23    dButton.mode(PullDown);
24
25    alarmLed = OFF;
26    incorrectCodeLed = OFF;
27    systemBlockedLed = OFF;
28
29    bool alarmState = OFF;
30    int numberOfIncorrectCodes = 0;
31
32    while (true) {
33
34        if ( gasDetector || overTempDetector ) {
35            alarmState = ON;
36        }
37
38        alarmLed = alarmState;
39
40        if ( numberOfIncorrectCodes < 5 ) {
41
42            if ( aButton && bButton && cButton && dButton && !enterButton ) {
43                incorrectCodeLed = OFF;
44            }
45        }
46    }
47}
```

```
46         if ( enterButton && !incorrectCodeLed && alarmState ) {
47             if ( aButton && bButton && !cButton && !dButton ) {
48                 alarmState = OFF;
49                 numberOfIncorrectCodes = 0;
50             } else {
51                 incorrectCodeLed = ON;
52                 numberOfIncorrectCodes = numberOfIncorrectCodes + 1;
53             }
54         } else {
55             systemBlockedLed = ON;
56         }
57     }
58 }
59 }
```

Code 2.1 Original code used in Example 1.5.

In order to improve the code understandability, the program will be divided and reorganized into *declarations* and *functions*. In this context, the following definitions apply:



DEFINITION: A *declaration* is a section of code where variables or other elements are declared and, sometimes, initialized.

DEFINITION: A *function* is a piece of code that carries out one or more specific tasks and can be used in a given program one or more times.

The use of functions provides two advantages:

- Code modularization: organizing program code into different modules makes it easier to understand a program, which improves its *Maintainability*.
- Code reutilization: the usage of functions avoids the need to write the same piece of code many times. In this way, code size is reduced, and the maintainability of the program is increased.



DEFINITION: Software *Maintainability* is defined as the degree to which it is feasible for other programmers to understand, repair, and enhance program code over time.

In Code 2.2 and Code 2.3, the declarations and functions that were identified in Code 2.1, according to the definitions of *declaration* and *function* presented above, are shown.

In Code 2.2, the declaration and initialization of variables and objects is indicated in magenta, and the code that can be grouped into functions in order to improve the code maintainability is indicated in orange.

```

1 #include "mbed.h"
2 #include "arm_book_lib.h" Libraries
3
4 int main()
5 {
6     DigitalIn enterButton(BUTTON1);
7     DigitalIn gasDetector(D2);
8     DigitalIn overTempDetector(D3);
9     DigitalIn aButton(D4);
10    DigitalIn bButton(D5);
11    DigitalIn cButton(D6);
12    DigitalIn dButton(D7);
13
14    DigitalOut alarmLed(LED1);
15    DigitalOut incorrectCodeLed(LED3);
16    DigitalOut systemBlockedLed(LED2);
17
18    gasDetector.mode(PullDown);
19    overTempDetector.mode(PullDown);
20    aButton.mode(PullDown);
21    bButton.mode(PullDown);
22    cButton.mode(PullDown);
23    dButton.mode(PullDown); Declaration and initialization of local objects
24
25    alarmLed = OFF;
26    incorrectCodeLed = OFF; → Function: inputsInit()
27    systemBlockedLed = OFF; → Function: outputsInit()
28
29    bool alarmState = OFF;
30    int numberofIncorrectCodes = 0; Declaration and initialization of local variables
31

```

Code 2.2 Analysis of the first part of the code of Example 1.5.



NOTE: In this book, the names of functions are stylized using the lower camel case format, as in *inputsInit()*.

In Code 2.3, two different groups of code are identified. One is called *alarmActivationUpdate()* and is used to activate the Alarm LED when gas presence or over temperature is detected. The other is identified as *alarmDeactivationUpdate()* and is responsible for the deactivation of the Alarm LED when the correct code is entered, as well as being responsible for blocking the system if more than five incorrect codes are entered.

```

32     while (true) {
33
34         if ( gasDetector || overTempDetector ) {
35             alarmState = ON;
36         }
37
38         alarmLed = alarmState;
39
40         if ( numberOfIncorrectCodes < 5 ) {
41
42             if ( aButton && bButton && cButton && dButton && !enterButton ) {
43                 incorrectCodeLed = OFF;
44             }
45
46             if ( enterButton && !incorrectCodeLed && alarmState ) {
47                 if ( aButton && bButton && !cButton && !dButton ) {
48                     alarmState = OFF;
49                     numberOfIncorrectCodes = 0;
50                 } else {
51                     incorrectCodeLed = ON;
52                     numberOfIncorrectCodes = numberOfIncorrectCodes + 1;
53                 }
54             }
55         } else {
56             systemBlockedLed = ON;
57         }
58     }
59 }
```

Code 2.3 Analysis of the second part of the code of Example 1.5.

In Code 2.4, the libraries and declarations that had been identified are presented in a more structured way, following the conclusions obtained from the analysis of Code 2.2.

The code identified in Code 2.2 as “declaration and initialization of local objects” was moved outside the *main()* function and is now located, in Code 2.4, in the section “Declaration and initialization of public global objects.” The name given to this section is due to the fact that, by the modification of the location of these objects, they change from being local (only available to the *main()* function) to being global (available to every function in the program).

The same rationale applies to the variables identified in Code 2.2 that are located in Code 2.4 in the section “Declaration and initialization of public global variables.”

A section called “Declarations (prototypes) of public functions” is also introduced in Code 2.4. This section was not previously identified in Code 2.2 but is necessary because in the C/C++ language functions have to be declared before using them for the first time (i.e., *calling* them from another function). This declaration is named *function prototype*.

The keyword `void` used in Code 2.4 specifies that the function does not return a value. In Example 2.5, it will be shown that functions can return a value that results, for example, from a mathematical operation.



NOTE: In Code 2.4, single-line comments (indicated by `//`) are used for the first time in the book. Comments are completely ignored by C/C++ compilers and can be used to increase software maintainability because the purpose of the code can be explained above the code itself. Multiple-line comments can also be used (they begin with `/*` and end with `*/`). Comments will be extensively used throughout this book, mainly to indicate the beginning of code sections. Many programmers also use comments alongside their code.

```

1 //=====[Libraries]=====
2
3 #include "mbed.h"           Libraries
4 #include "arm_book_lib.h"
5
6 //=====[Declaration and initialization of public global objects]=====
7
8 DigitalIn enterButton(BUTTON1);
9 DigitalIn gasDetector(D2);
10 DigitalIn overTempDetector(D3);
11 DigitalIn aButton(D4);
12 DigitalIn bButton(D5);
13 DigitalIn cButton(D6);
14 DigitalIn dButton(D7);
15
16 DigitalOut alarmLed(LED1);
17 DigitalOut incorrectCodeLed(LED3);
18 DigitalOut systemBlockedLed(LED2);
19
20 //=====[Declaration and initialization of public global variables]=====
21
22 bool alarmState = OFF;      Declaration and initialization
23 int numberOfIncorrectCodes = 0; of public global variables
24
25 //=====[Declarations (prototypes) of public functions]=====
26
27 void inputsInit();          Declarations (prototypes)
28 void outputsInit();         of public functions
29
30 void alarmActivationUpdate();
31 void alarmDeactivationUpdate();
32

```

Code 2.4 Libraries and declarations of the modularized version of Example 1.5.

Code 2.5 shows the `main()` function and all the other functions of Code 2.1 after applying the proposed modularization indicated in Code 2.2 and Code 2.3.

The code of the examples that will be introduced in this chapter will be organized as shown in Code 2.4 and Code 2.5.

In this way, all the programs in this chapter will have at least the following parts:

- Libraries
- Declaration and initialization of public global objects
- Declaration and initialization of public global variables
- Declarations (prototypes) of public functions
- Main function
- Implementations of public functions

In the following chapters, some other possibilities for declarations and function prototypes will be also explored.



NOTE: In Chapter 5 this topic will be addressed in greater depth, and it will be explained in more detail how to apply modularization to embedded systems programming.



WARNING: Objects and variables must be declared before being used. The order of the other elements in the program code can be given by conventions and good practices.

```

33 //=====[Main function, the program entry point after power on or reset]=====
34 Implementation of the main() function
35 int main()
36 {
37     inputsInit();
38     outputsInit();
39     while (true) {
40         alarmActivationUpdate();
41         alarmDeactivationUpdate();
42     }
43 }
44
45 //=====[Implementations of public functions]=====
46 Implementation of inputsInit() function
47 void inputsInit()
48 {
49     gasDetector.mode(PullDown);
50     overTempDetector.mode(PullDown);
51     aButton.mode(PullDown);
52     bButton.mode(PullDown);
53     cButton.mode(PullDown);
54     dButton.mode(PullDown);
55 }
56 Implementation of outputsInit() function
57 void outputsInit()
58 {
59     alarmLed = OFF;
60     incorrectCodeLed = OFF;
61     systemBlockedLed = OFF;
62 }
63 Implementation of alarmActivationUpdate() function
64 void alarmActivationUpdate()
65 {
66     if ( gasDetector || overTempDetector ) {
67         alarmState = ON;
68     }
69     alarmLed = alarmState;
70 }
71 Implementation of alarmDeactivationUpdate() function
72 void alarmDeactivationUpdate()
73 {
74     if ( numberOfIncorrectCodes < 5 ) {
75         if ( aButton && bButton && cButton && dButton && !enterButton ) {
76             incorrectCodeLed = OFF;
77         }
78         if ( enterButton && !incorrectCodeLed && alarmState ) {
79             if ( aButton && bButton && !cButton && !dButton ) {
80                 alarmState = OFF;
81                 numberOfIncorrectCodes = 0;
82             } else {
83                 incorrectCodeLed = ON;
84                 numberOfIncorrectCodes = numberOfIncorrectCodes + 1;
85             }
86         }
87     } else {
88         systemBlockedLed = ON;
89     }
90 }

```

Code 2.5 Functions of the modularized version of Example 1.5.

Proposed Exercise

- How can comments be inserted in Code 2.4 and Code 2.5 to document the code above the code itself?

Answer to the Exercise

- Code 2.6 and Code 2.7 show how detailed comments can be included. The comments between lines 1 to 21 and lines 32 to 47 of Code 2.6, and the comments between lines 3 to 5, 10 to 34, and 38 to 44 of Code 2.7, follow the Doxygen standard. This standard is available from [2] and allows the use of a program to generate a website with the documentation, as shown in Figure 2.2.



NOTE: In this book, comments as shown in Code 2.6 and Code 2.7 are not included above the code because all the programs are explained in detail in the text. However, the reader is encouraged to include this type of comment above the code of their own programs.

```
1  /*! @mainpage Example 1.5 Modularized and with doxygen comments
2   * @date Friday, January 29, 2021
3   * @authors Pablo Gomez, Ariel Lutenberg and Eric Pernia
4   * @section genDesc General Description
5   *
6   * This is a preliminary implementation of the smart home system, where the
7   * code has been modularized using functions and documented using Doxygen.
8   * The entry point to the program documentation can be found at
9   * this \ref Example_1_5_Modularized_withDoxygenComments.cpp "link"
10  *
11  * @section genRem General Remarks
12  * [Write here relevant information about the program]
13  *
14  * @section changelog Changelog
15  *
16  * | Date | Description
17  * |-----|-----|
18  * | 29/01/2021 | First version of program |
19  *
20  *
21  */
22
23 /* Example of comment that follows C/C++ format, but not the doxygen standard */
24
25 //=====[Libraries]=====
26
27 #include "mbed.h"
28 #include "arm_book_lib.h"
29
30 //=====[Declaration and initialization of public global objects]=====
31
32 DigitalIn enterButton(BUTTON1); /*< Object associated to
33                                Enter key (B1 User Button) */
34 DigitalIn gasDetector(D2); /*< Object associated to gas detector (D2) */
35 DigitalIn overTempDetector(D3); /*< Object associated to over temperature
36                               detector (D3) */
37 DigitalIn aButton(D4); /*< Object associated to A key (pin D4) */
38 DigitalIn bButton(D5); /*< Object associated to B key (pin D5) */
39 DigitalIn cButton(D6); /*< Object associated to C key (pin D6) */
40 DigitalIn dButton(D7); /*< Object associated to D key (pin D7) */
41
42 DigitalOut alarmLed(LED1); /*< Output associated to alarm
43                            led indicator (LD1)*/
44 DigitalOut incorrectCodeLed(LED3); /*< Output associated to incorrect
45                            code indicator (LD3)*/
46 DigitalOut systemBlockedLed(LED2); /*< Output associated to system blocked
47                            indicator (LD2)*/
```

Code 2.6 Modularized version of Example 1.5 with comments included over the code (Part 1/3).

```

1 //=====[Declaration and initialization of public global variables]=====
2
3 bool alarmState = OFF; /*< Alarm state flag */
4 int numberOfIncorrectCodes = 0; /*< Accounts for the number of incorrect codes
5                                         entered */
6
7 //=====[Declarations (prototypes) of public functions]=====
8
9 void inputsInit();
10 /**<
11     This function configures gasDetector, overTempDetector and aButton to dButton
12     with internal pull-down resistors.
13     @param none
14 */
15
16 void outputsInit();
17 /**<
18     This function initializes the outputs of the system:
19     -# alarmLed = OFF
20     -# incorrectCodeLed = OFF
21     -# systemBlockedLed = OFF
22 */
23
24 void alarmActivationUpdate();
25 /**<
26     This function assigns ON to alarmLed if gasDetector or overTempDetector are
27     active, and assigns alarmLed with alarmState.
28 */
29
30 void alarmDeactivationUpdate();
31 /**<
32     This function assesses the entered code and controls the activation of
33     alarmLed, incorrectCodeLed, and systemBlockedLed.
34 */
35
36 //=====[Main function, the program entry point after power on or reset]=====
37
38 /**
39 * Calls functions to initialize the declared input and output objects, and to
40 * implement the system behavior.
41 * @param none
42 * @return The returned value represents the success
43 * of application.
44 */
45 int main()
46 {
47     inputsInit();
48     outputsInit();
49     while (true) {
50         alarmActivationUpdate();
51         alarmDeactivationUpdate();
52     }
53     return 0;
54 }
```

Code 2.7 Modularized version of Example 1.5 with comments included over the code (Part 2/3).



NOTE: In line 53 of Code 2.7, the return statement has been included because the main function is expected to return an integer value (line 45), as required by most compilers. Line 53 establishes a return value (zero), which is considered to be the return value corresponding to a successful execution of the program. However, notice that the `while` statement in line 49 is executed forever and, therefore, line 53 is never reached. Thus, in most programs of this book, the “return 0” statement is not included at the end of the `main()` function, despite `main()` being declared as `int main()`.

```

1 //=====[Implementations of public functions]=====
2
3 void inputsInit()
4 {
5     gasDetector.mode(PullDown);
6     overTempDetector.mode(PullDown);
7     aButton.mode(PullDown);
8     bButton.mode(PullDown);
9     cButton.mode(PullDown);
10    dButton.mode(PullDown);
11 }
12
13 void outputsInit()
14 {
15     alarmLed = OFF;
16     incorrectCodeLed = OFF;
17     systemBlockedLed = OFF;
18 }
19
20 void alarmActivationUpdate()
21 {
22     if ( gasDetector || overTempDetector ) {
23         alarmState = ON;
24     }
25     alarmLed = alarmState;
26 }
27
28 void alarmDeactivationUpdate()
29 {
30     if ( numberOfIncorrectCodes < 5 ) {
31         if ( aButton && bButton && cButton && dButton && !enterButton ) {
32             incorrectCodeLed = OFF;
33         }
34         if ( enterButton && !incorrectCodeLed && alarmState ) {
35             if ( aButton && bButton && !cButton && !dButton ) {
36                 alarmState = OFF;
37                 numberOfIncorrectCodes = 0;
38             } else {
39                 incorrectCodeLed = ON;
40                 numberOfIncorrectCodes = numberOfIncorrectCodes + 1;
41             }
42         }
43     } else {
44         systemBlockedLed = ON;
45     }
46 }
```

Code 2.8 Modularized version of Example 1.5 with comments included over the code (Part 3/3).

Figure 2.2 shows the website with the documentation of the program that is generated using Doxygen. The website is available as a .zip file from [1]. To navigate to the website, the reader must download the .zip file, uncompress it, and double click on the *index.html* file. From [1] the source files that were used to generate the website using Doxygen are also available; these comprise the file *Example_1_5_Modularized_withDoxygenComments.cpp* and a subset of the Mbed OS files, where for the sake of simplicity and brevity only Mbed OS entities that are used in the example were included.

Figure 2.2 Website with the program documentation generated with Doxygen.

By clicking over the link that is highlighted with a red arrow in Figure 2.2, the web page shown in Figure 2.3 is displayed in the web browser. By means of scrolling down this web page, a detailed description of each function and variable based on the Doxygen formatted comments introduced in Code 2.6 and Code 2.7 can be seen, as shown in Figure 2.4 and Figure 2.5.

Figure 2.3 Detailed description of functions and variables of the program that is available on the website (Part 1/3).

Function Documentation

- **alarmActivationUpdate()**
void alarmActivationUpdate ()

This function assigns ON to alarmLED if gasDetector or overTempDetector are active, and assigns alarmLed with alarmState.
Definition at line 124 of file Example_1_5_Modularized_withDoxygenComments.cpp. 
- **alarmDeactivationUpdate()**
void alarmDeactivationUpdate ()

This function assesses the entered code and controls the activation of alarmLED, incorrectCodeLed, and systemBlockedLed.
Definition at line 132 of file Example_1_5_Modularized_withDoxygenComments.cpp. 
- **inputsInit()**
void inputsInit ()

This function configures gasDetector, overTempDetector and aButton to dButton with internal pull-down resistors.
Parameters

Figure 2.4 Detailed description of functions and variables of the program that is available on the website (Part 2/3).

Variable Documentation

- **aButton**
DigitalIn aButton(D4)

Object associated to A key (pin D4)
- **alarmLed**
DigitalOut alarmLed(LED1)

Output associated to alarm led indicator (LD1)
- **alarmState**
bool alarmState = OFF

Alarm state flag
Definition at line 51 of file Example_1_5_Modularized_withDoxygenComments.cpp. 
- **bButton**

Figure 2.5 Detailed description of functions and variables of the program that is available on the website (Part 3/3).

By clicking on the green arrows shown in Figure 2.3, Figure 2.4, and Figure 2.5, an interactive view of the code is displayed in the web browser, which grants information about the different elements as well as linking to the corresponding documentation that is available in the website, as shown in Figure 2.6.

Finally, it is worth mentioning that in the “Classes” menu can be found a “Class List” item, where reference information on *DigitalIn* and *DigitalOut* can be accessed, as shown in Figure 2.7. This reference information is a subset of the complete information on Mbed OS elements that can be found in [3], as can be seen in Figure 2.8.

```

1  /* Example of comment that follows C/C++ format, but not the doxygen standard */
2  //*****[Libraries]*****
3
4  #include "mbed.h"
5  #include "arm_blink_lib.h"
6
7  //*****[Declaration and initialization of public global objects]*****
8
9  DigitalIn enterButton(BUTTON1);
10 DigitalIn gasDetector(D2);
11 DigitalIn overTempDetector(D3);
12 DigitalIn aButton(D5);
13 DigitalIn bButton(D6);
14 DigitalIn cButton(D6);
15 DigitalIn dButton(D7);
16 DigitalIn eButton(D8);
17 DigitalOut incorrectCodeLed(LED3);
18 DigitalOut systemBlocked(LED2);
19 //*****[Declaration and initialization of public global variables]*****
20
21 bool alarmState = OFF;
22 int numberIncorrectCodes = 0;
23 //*****[Declarations (prototypes) of public functions]*****
24 void inputsInit();
25 void outputsInit();
26 void alarmActivation();
27 void alarmDeactivation();
28 //*****[Main function, the program entry point after power on or reset]*****
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84

```

Figure 2.6 Interactive view of the code.

mbed::DigitalIn Class Reference

Public Member Functions

- `DigitalIn (PinName pin)`
- `DigitalIn (PinName pin, PinMode mode)`
- `int read ()`
- `void mode (PinMode pull)`
- `int is_connected ()`
- `operator int ()`

Protected Attributes

- `gpio_t gpio`

Detailed Description

A digital input, used for reading the state of a pin

Figure 2.7 *DigitalIn* class reference.

The screenshot shows the Mbed website interface. The top navigation bar includes the Mbed logo, a search bar, and links for 'Compiler' and 'Documentation'. A sidebar on the left provides navigation through various Mbed OS components like 'Mbed OS bare metal profile', 'Quick start', 'API references and tutorials' (which is currently selected), and 'Drivers' (which is also selected). Under 'Drivers', there are sub-links for 'DigitalIn', 'DigitalOut', 'AnalogIn', 'AnalogOut', 'BusIn', 'BusOut', and 'BusInOut'. The main content area displays the 'DigitalIn' class reference. It includes a breadcrumb trail: 'Docs > API references and tutorials > Drivers > I/O APIs > DigitalIn'. Below this is the title 'DigitalIn' and a brief description: 'Use the DigitalIn interface to read the value of a digital input pin. The logic level is either 1 or 0. You can use any of the numbered Arm Mbed pins as a DigitalIn.' A 'Feedback' button is located in the bottom right corner of the content area.

Figure 2.8 Website of Mbed with detailed information about DigitalIn and the whole Application Programming Interface.

Example 2.1: Monitor the Alarm State with a PC

Objective

Introduce functions and methods to exchange data between the NUCLEO board and the PC.

Summary of the Expected Behavior

If key “1” is pressed on the PC, the NUCLEO board sends a message to the PC indicating the alarm state, and the message is printed on the serial terminal.

Test the Proposed Solution on the Board

Import the project “Example 2.1” using the URL available in [1], build the project, and drag the

.bin file onto the NUCLEO board. Open the serial terminal. Read the message that appears on the serial terminal summarizing the list of available commands. Press “1” on the PC keyboard and read the message that appears on the serial terminal regarding the state of the alarm. Press the button connected to D2 that represents gas detection. Press “1” again on the PC and read the message that appears on the serial terminal indicating the new state of the alarm. Press “2” (or any other key) on the PC and read the message that appears on the serial terminal indicating that the only valid command is “1”.

Discussion of the Proposed Solution

In this example, the functionality of monitoring the smart home system from a PC using a serial terminal is incorporated. For this purpose, an object that will drive the serial port is declared by means of *UnbufferedSerial uartUsb(USBTX, USBRX, 115200)*, which does not use intermediary buffers to store bytes to transmit to or read from the hardware; thus, the program is responsible for processing each received byte. The parameters USBTX and USBRX indicate that those pins are to be used for transmission and reception of the data of the serial communication, respectively. The parameter 115200 is used to configure the baud rate of the serial communication. In the Under the Hood section of this chapter, the main concepts behind serial communication will be analyzed in more detail.

In the definition of the variables, the data type *char* is used for the variables that are storing characters. This is because Mbed OS uses this data type to store characters. This data type has 8 bits, the same as the data package that is exchanged in each message using the serial communication. This will also be explained in the Under the Hood section of this chapter.

The proposed solution to this example follows the structure that was introduced in Code 2.4 and Code 2.5. Moreover, most of the code used in this proposed solution is the same as the code presented in Code 2.4 and Code 2.5. Therefore, only the differences between those code listings and the code used in this proposed solution will be discussed in the following pages.

A new function called *uartTask()* is used to send and receive information from the PC by means of one of the UARTs of the STM32 microcontroller of the NUCLEO board. The details of the function *uartTask()*, which receives commands from the PC and transmits to the PC the messages that should be displayed on the serial terminal, are presented in Figure 2.9. If there is a new character to be read, it is stored in the variable *receivedChar*. Then, if *receivedChar* is ‘1’, the message reporting the alarm state is sent to the serial terminal. If the received character is not ‘1’, a message containing the list of available commands is sent to the serial terminal. This is shown in Figure 2.9.

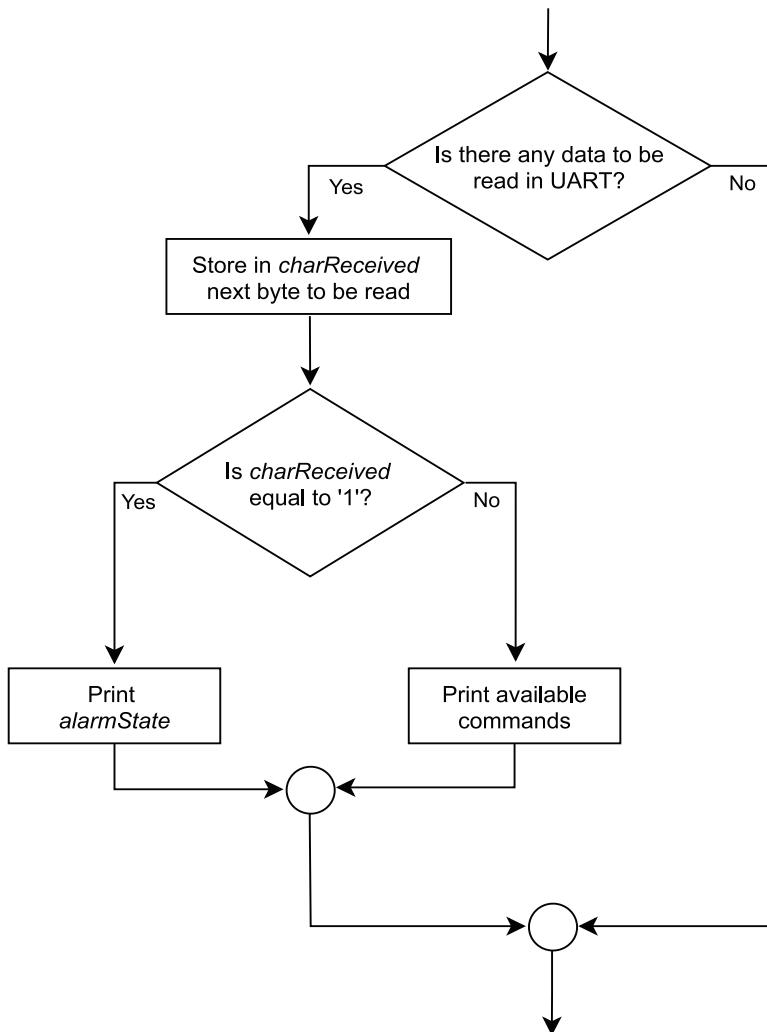


Figure 2.9 Details of the function `uartTask()` used in this proposed solution to Example 2.1.

Implementation of the Proposed Solution

Lines added in specific sections of the program code presented in Code 2.4 and Code 2.5 are shown in Table 2.1. The three functions that were included are discussed below in this example.

Table 2.1 Sections in which lines were added to Code 2.4 and Code 2.5.

Section	Lines that were added to Code 2.4 and Code 2.5.
Declaration and initialization of public global objects	<code>UnbufferedSerial uartUsb(USBTX, USBRX, 115200);</code>
Declarations (prototypes) of public functions	<code>void uartTask();</code> <code>void availableCommands();</code>

In order to periodically check if there is a new character sent by the PC, a call to the function *uartTask()* is included in the *main()* function, as shown in Code 2.9.

```

1 int main()
2 {
3     inputsInit();
4     outputsInit();
5     while (true) {
6         alarmActivationUpdate();
7         alarmDeactivationUpdate();
8         uartTask();
9     }
10 }
```

Code 2.9 New implementation of *main.cpp*.

The implementation of *uartTask()* is shown in Code 2.10. On line 3, a variable of type char named *receivedChar* is declared and set to '\0'. The character '\0' is named the null character and represents the zero-element character (i.e., a '\0' written to *uartUsb* is not printed on the serial terminal). The null character has the ASCII code 0, as will be shown on Chapter 6. For that reason, some programmers use 0 instead of '\0'. Confusion should be avoided with character '0', which is used to print '0' on the serial terminal and has the ASCII code 48, as will be shown in Chapter 6. In this book, '\0' is preferred to indicate the null character.

On line 4, *uartUsb.readable()* is used to determine if there is data available to be read in the UART connected to the USB (see Figure 2.1). If so, *uartUsb.read()* is used in line 5 to get the next available character. It uses "&receivedChar" to indicate where to store the character (the meaning of the & will be discussed in chapter 10) and "1" to indicate that one character must be read. In line 6, the read character is compared with '1', which corresponds to the key used in this example to get the alarm state.

If the key pressed is "1", then the state of the alarm is assessed in line 7. If *alarmState* is true, then in line 8 the message "The alarm is activated\r\n" is sent to the PC using *uartUsb.write()*. The "\r\n" at the end of the message is to indicate that the next message should appear on a new line (\n), at the beginning of the line (\r). The number "24" is the number of characters of "The alarm is activated\r\n" that must be sent (in this case, all the characters, considering that each of "\n" and "\r" count as a single character). If *alarmState* is false, then "The alarm is not activated\r\n" is sent on line 10 (note that this message has 28 characters).

If the received character is not '1', then the available commands are printed in line 13 because it is considered that the user has to be informed about the available commands. The implementation of the function *availableCommands()* is shown in Code 2.11. A specific function is used to print the available commands in order to show how a function can call another function, and also because in the following examples more available commands will be incorporated; therefore, it is convenient to have a specific function to print the list of available commands. Notice that "\r\n\r\n" is used on line 4 in order to print a blank line (a line without text).



NOTE: `readable()`, `read()`, and `write()` are part of the “UnbufferedSerial” API. For more functions of the UnbufferedSerial API, refer to [4].

```
1 void uartTask()
2 {
3     char receivedChar = '\0';
4     if( uartUsb.readable() ) {
5         uartUsb.read( &receivedChar, 1 );
6         if ( receivedChar == '1' ) {
7             if ( alarmState ) {
8                 uartUsb.write( "The alarm is activated\r\n", 24 );
9             } else {
10                 uartUsb.write( "The alarm is not activated\r\n", 28 );
11             }
12         } else {
13             availableCommands();
14         }
15     }
16 }
```

Code 2.10 Details of the function `uartTask()`.

```
1 void availableCommands()
2 {
3     uartUsb.write( "Available command:\r\n", 20 );
4     uartUsb.write( "Press '1' to get the alarm state\r\n\r\n", 36 );
5 }
```

Code 2.11 Details of the function `availableCommands()`.



NOTE: From this chapter onwards, comments alongside the code (as in Chapter 1) will not be included, because it is considered that the reader does not need them anymore.

Proposed Exercises

1. What would happen if “`\r\n`” were removed from lines 7 and 9 of Code 2.10?
2. In section 2.2.1, some configurations such as the number of data bits, the parity, and the number of stop bits were mentioned. What function of the UnbufferedSerial API can be used to configure those parameters?
3. How can a report of the state of the gas detector and the over temperature detector be added to the function `uartTask()` using an `if` and `else if` structure?

Answers to the Exercises

1. All the messages would be printed on the same line.
2. The serial communication can be configured using `uartUsb.format()`, as discussed in [4]. In this chapter, there is no need to make this configuration because the default configuration is used, which is 8 bits, no parity, and one stop bit. All the available configurations are summarized in Table 2.2. To make the configuration used in this example, `uartUsb.format(8, SerialBase::None, 1)` should be used.

Table 2.2 Available configurations of the UnbufferedSerial object.

Configuration	Available values	Default value
Number of bits in a word	5, 6, 7, 8	8
Parity used	SerialBase::None, SerialBase::Odd, SerialBase::Even, SerialBase::Forced1, SerialBase::Forced0	SerialBase::None
Number of stop bits	1, 2	1

3. For this purpose, the `if else` structure shown in Code 2.12 could be used.

```

1 void uartTask()
2 {
3     char receivedChar = '\0';
4     if( uartUsb.readable() ) {
5         uartUsb.read( &receivedChar, 1 );
6         if ( receivedChar == '1' ) {
7             if ( alarmState ) {
8                 uartUsb.write( "The alarm is activated\r\n", 24 );
9             } else {
10                 uartUsb.write( "The alarm is not activated\r\n", 28 );
11             }
12         } else if ( receivedChar == '2' ) {
13             if ( gasDetector ) {
14                 uartUsb.write( "Gas is being detected\r\n", 23 );
15             } else {
16                 uartUsb.write( "Gas is not being detected\r\n", 27 );
17             }
18         } else if ( receivedChar == '3' ) {
19             if ( overTempDetector ) {
20                 uartUsb.write( "Temperature is above the maximum level\r\n", 40 );
21             } else {
22                 uartUsb.write( "Temperature is below the maximum level\r\n", 40 );
23             }
24         } else {
25             availableCommands();
26         }
27     }
28 }
```

Code 2.12 Details of the function `uartTask()` used in the implementation of proposed exercise 3.

Example 2.2: Monitor Over Temperature and Gas Detection with a PC

Objective

Introduce the switch-case statement.

Summary of the Expected Behavior

The expected behavior is similar to Example 2.1, but in this example when key “2” is pressed on the PC, the state of the gas detector is sent to the PC; when key “3” is pressed on the PC, the state of the over temperature detector is sent to the PC. In fact, this is the same behavior as in the second proposed exercise of Example 2.1. The difference is the way in which this behavior will be achieved: not using a group of nested *ifs*, but the switch-case statement.

Test the Proposed Solution on the Board

Import the project “Example 2.2” using the URL available in [1], build the project, and drag the *.bin* file onto the NUCLEO board. Open the serial terminal, press “4” on the PC keyboard and read the message that appears on the serial terminal indicating the list of available commands. Press “2” on the PC keyboard and read the message that appears on the serial terminal, indicating that gas is not being detected. Press and hold the button connected to D2 in order to simulate gas detection. Press “2” again on the PC and read the message that appears on the serial terminal, indicating that gas is being detected. Repeat this operation with key “3” and the button connected to D3 to simulate the detection of over temperature.

Discussion of the Proposed Solution

In Example 2.1, the alarm state was reported to the PC by means of serial communication. The aim of this example is to extend the report functionality that was introduced in proposed exercise 3 of Example 2.1. The difference in this proposed solution is the use of the *switch statement*.

In the *switch* statement, a variable is compared in sequence to a list of values. Each value of the list is called a *case*. In this example, the variable being compared is *receivedChar* and there are three cases: ‘1’, ‘2’ and ‘3’. There is also a *default* case that is executed if none of the cases is equal to the variable being compared.

Even though the behavior is exactly as in proposed exercise 2 of Example 2.1, the reader will be able to see in the code shown below that by using the *switch* statement the code becomes easier to understand.

The flow diagram of the new function *uartTask()* is presented in Figure 2.10. If there is a new character to be read, the corresponding byte is stored in the variable *receivedChar*. Then, *receivedChar* is evaluated by means of the *switch* statement, and the NUCLEO board reports the corresponding value to the PC using the serial communication.

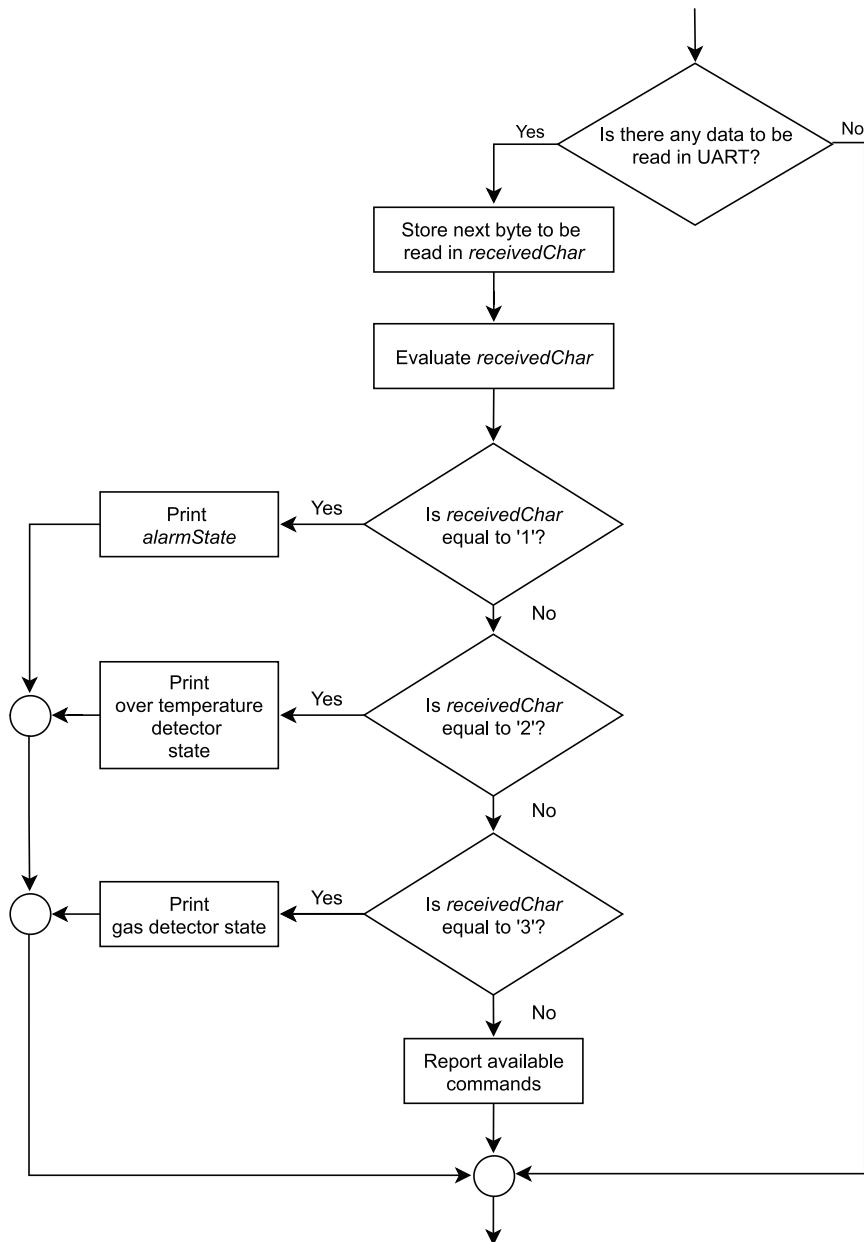


Figure 2.10 Details of the function `uartTask()` used in the proposed solution to Example 2.2.

Implementation of the Proposed Solution

In Code 2.13, the new implementation of `availableCommands()`, where the new commands are included, is shown. Note that in Code 2.11 there were 36 characters in line 4, while in Code 2.13 there are 34 characters in line 4. This is because only one "\r\n" is used in line 4 of Code 2.13 because the last command listed is now "3" (get the over temperature detector state). In Code 2.14, the new

implementation of the function *uartTask()* is presented. The code is very similar to the program code examples discussed earlier, and therefore for the sake of brevity it is not discussed here again.

```
1 void availableCommands()
2 {
3     uartUsb.write( "Available commands:\r\n", 21 );
4     uartUsb.write( "Press '1' to get the alarm state\r\n", 34 );
5     uartUsb.write( "Press '2' to get the gas detector state\r\n", 41 );
6     uartUsb.write( "Press '3' to get the over temperature detector state\r\n\r\n", 56
7 );
8 }
```

Code 2.13 New implementation of *availableCommands()*.

```
1 void uartTask()
2 {
3     char receivedChar = '\0';
4     if( uartUsb.readable() ) {
5         uartUsb.read( &receivedChar, 1 );
6         switch (receivedChar) {
7             case '1':
8                 if ( alarmState ) {
9                     uartUsb.write( "The alarm is activated\r\n", 24 );
10                } else {
11                    uartUsb.write( "The alarm is not activated\r\n", 28 );
12                }
13                break;
14            case '2':
15                if ( gasDetector ) {
16                    uartUsb.write( "Gas is being detected\r\n", 23 );
17                } else {
18                    uartUsb.write( "Gas is not being detected\r\n", 27 );
19                }
20                break;
21            case '3':
22                if ( overTempDetector ) {
23                    uartUsb.write( "Temperature is above the maximum level\r\n", 40 );
24                } else {
25                    uartUsb.write( "Temperature is below the maximum level\r\n", 40 );
26                }
27                break;
28            default:
29                availableCommands();
30                break;
31        }
32    }
33}
```

Code 2.14 Details of the function *uartTask()*.

Proposed Exercises

1. What would happen if the default case were removed?
2. How can a command be implemented using the switch-case statement such that if the “d” key is pressed on the keyboard of the computer, then the serial terminal indicates the state of the gas and

over temperature detectors?

Answers to the Exercises

1. There will be no response when a character that is not listed in the cases is pressed on the PC keyboard.
2. A new case could be added to the *switch* statement of *uartTask()*, as shown in Code 2.15.

```

1  case 'd':
2      if ( gasDetector ) {
3          uartUsb.write( "Gas is being detected\r\n", 23 );
4      } else {
5          uartUsb.write( "Gas is not being detected\r\n", 27 );
6      }
7
8      if ( overTempDetector ) {
9          uartUsb.write( "Temperature is above the maximum level\r\n", 40 );
10     } else {
11         uartUsb.write( "Temperature is below the maximum level\r\n", 40 );
12     }
13     break;

```

*Code 2.15 Details of the new case in function *uartTask()*.*

Example 2.3: Deactivate the Alarm Using the PC

Objective

Develop more complex programs that make use of serial communication.

Summary of the Expected Behavior

The behavior is the same as in Example 2.2, but now the alarm can be deactivated from the PC.

Test the Proposed Solution on the Board

Import the project “Example 2.3” using the URL available in [1], build the project, and drag the *.bin* file onto the NUCLEO board. Open the serial terminal and press and release the button connected to D3 to simulate an over temperature situation. Press “4” on the PC keyboard and look at the message that appears on the serial terminal. Press the code sequence, “1”, then “1”, then “0”, and finally “0”, and look at the message that appears on the serial terminal indicating that the entered code is correct.

Check that the Alarm LED is turned off. Press the button connected to D3. Press “4” on the PC.

Enter an incorrect code sequence, for instance, “1”, then “0”, then “0”, and finally “0”. Check that the Incorrect code LED is turned on. A new code sequence can be tried without the need for turning off the Incorrect code LED, as in example 1.5. After five incorrect attempts, the System blocked LED will turn on.



NOTE: Even when the system is blocked, because five incorrect codes have been entered and codes cannot be entered at the control panel anymore, the proposed implementation allows a code to be entered from the PC to unlock the smart home system without the need to reset the NUCLEO board or turn off the alarm.

The reader is encouraged to activate the alarm, enter five incorrect codes in a row (either from the control panel or from the PC), see how the System blocked LED turns on, and then press “4” on the PC keyboard to enter the correct code in order to unlock the system.

Discussion of the Proposed Solution

In Example 1.4, a code was introduced that allowed the user to turn off the alarm. The code was entered using the buttons connected to D4, D5, D6, and D7. The alarm was turned off only when the right code was entered: the buttons connected to D4 and D5 were both pressed, and at the same time the buttons connected to D6 and D7 were both released. In this example, the functionality to turn off the alarm by means of the PC is implemented. Due to the fact that in this chapter only one PC keyboard key is read at a time, the code should be entered as a sequence.

Implementation of the Proposed Solution

In this example, some lines were added to the program code of Example 2.2, as shown in Table 2.3. In Code 2.16, only the fragment of the code corresponding to case ‘4’ of the switch statement of `uartTask()` is shown because all the rest of the code remains the same as in Example 2.2. An explanation of each part is also included above Code 2.16.

In this example, the NUCLEO board sends messages through the serial port asking for the code (lines 2 to 12 of Code 2.16). The code entered in the PC is called the *code sequence*. In this example, a ‘1’ represents a pressed button and ‘0’ represents a released button. Then, four PC keys are read and stored in four different variables (`receivedAButton` to `receivedDButton`). Finally, this sequence is compared with the code stored in the NUCLEO board (line 14 to line 17) and depending on the result the Alarm LED is turned off or the Incorrect code LED is turned on, in the same way as in Example 1.5, and a message reporting the result is sent through the serial port.

Table 2.3 Lines that were added to the program of Example 2.2.

Section	Lines that were added
Declaration and initialization of public global variables	<pre>char receivedAButton = '\0'; char receivedBButton = '\0'; char receivedCButton = '\0'; char receivedDButton = '\0';</pre>
Function	Lines that were added
availableCommands	<pre>uartUsb.write("Press '4' to enter the code sequence\r\n\x0d", 40);</pre>

```

1 case '4':
2     uartUsb.write( "Please enter the code.\r\n", 24 );
3     uartUsb.write( "Type 1 for button pressed\r\n", 27 );
4     uartUsb.write( "Type 0 for button not pressed\r\n", 31 );
5     uartUsb.write( "Enter the value for 'A' Button\r\n", 32 );
6     uartUsb.read( &receivedAButton, 1 );
7     uartUsb.write( "Enter the value for 'B' Button\r\n", 32 );
8     uartUsb.read( &receivedBButton, 1 );
9     uartUsb.write( "Enter the value for 'C' Button\r\n", 32 );
10    uartUsb.read( &receivedCButton, 1 );
11    uartUsb.write("Enter the value for 'D' Button\r\n\r\n", 34 );
12    uartUsb.read( &receivedDButton, 1 );
13
14    if ( (receivedAButton == '1') &&
15        (receivedBButton == '1') &&
16        (receivedCButton == '0') &&
17        (receivedDButton == '0') ) {
18        uartUsb.write( "The code is correct\r\n\r\n", 23 );
19        alarmState = OFF;
20        incorrectCodeLed = OFF;
21        numberOfIncorrectCodes = 0;
22    } else {
23        uartUsb.write( "The code is incorrect\r\n\r\n", 25 );
24        incorrectCodeLed = ON;
25        numberOfIncorrectCodes = numberOfIncorrectCodes + 1;
26    }
27    break;

```

Code 2.16 Details of the new lines in the function `uartTask()`.

Proposed Exercises

- How can the code be changed in such a way that the order in which the keys are asked for is D, C, B, and finally A?
- How can the code be changed to add a case that shows the state of the Incorrect code LED?
- How can more buttons be incorporated into the smart home system?

Answers to the Exercises

- The order of lines 5 to 12 should be modified.
- In the switch statement, a case similar to the one implemented for the key “1” but reporting the Incorrect code LED instead of the alarm state should be incorporated.
- More variables similar to `receivedAButton` should be declared, and those variables should be incorporated in the `uartTask()` function.

Example 2.4: Improve the Code Maintainability using Arrays

Objective

Introduce the use of a `for` loop, `#define` and arrays.

Summary of the Expected Behavior

The behavior is the same as in Example 2.3, but the program is implemented using a *for loop* and *arrays* in order to improve the code maintainability.

Test the Proposed Solution on the Board

Import the project “Example 2.4” using the URL available in [1], build the project, and drag the *.bin* file onto the NUCLEO board. Perform the same actions as in Example 2.3.

Discussion of the Proposed Solution

In proposed exercise 3 of Example 2.3, the reader was encouraged to think about how to incorporate more buttons into the smart home system. It was seen that under the implementation used in Example 2.3, the complexity of the program is increased as the number of buttons is incremented. This example will show how to tackle this problem using a *for loop*.

The aim of this example is to show a more convenient implementation, based on a *for loop* and *arrays*. An array is a variable that stores a set of values. Each of those values can be individually read and modified by using an *index*. For example, if there is an array called *vector*, its first value can be accessed using *vector[0]*, its second value can be accessed using *vector[1]*, etc.

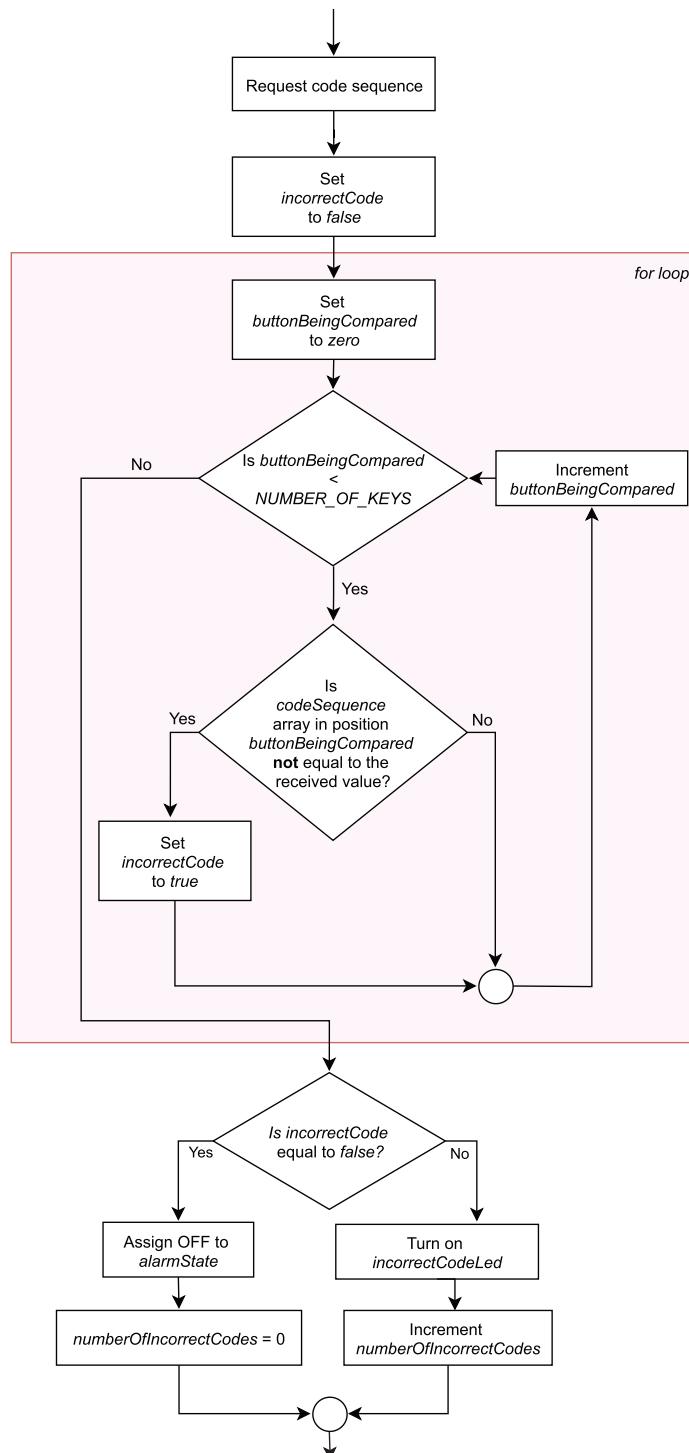


NOTE: It is important to remember that the first position of an array is accessed using the index [0].

Figure 2.11 details the flow diagram corresponding to case ‘4’ of the *switch statement* of *uartTask()*. In case ‘4’, the program uses a *for loop* to compare the PC keys pressed against the code sequence.

The *for loop* indicated in Figure 2.11 is used to check the keys being pressed one after the other. As the keys are entered, they are compared with the given code. If one of the keys does not correspond to the code sequence, then the variable *incorrectCode* is set to true.

Once those four keys are pressed on the PC (i.e., *buttonBeingCompared* is equal to *NUMBER_OF_KEYS*) the *for loop* is concluded. Then, depending on the state of *incorrectCode*, OFF is assigned to *alarmState* and 0 is assigned to *numberOfIncorrectCodes*, or the Incorrect code LED is turned on and the variable *numberOfIncorrectCodes* is incremented by one.

Figure 2.11 Details of the '4' of the function `uartTask()`.

Implementation of the Proposed Solution

In this example, an array called `codeSequence` is declared to store the code sequence of four keys. The size of the array and the type of data that is to be stored in the array need to be declared. A macro called “NUMBER_OF_KEYS” is used to indicate the size of the array. Below the section “Libraries”, a new section named “Definitions” is added, where NUMBER_OF_KEYS is defined. The lines that were added to this new section are shown in Table 2.4. Some other changes made to the code are also indicated in Table 2.4.

Table 2.4 Lines that were added and removed from the code used in Example 2.3.

Section	Lines that were added
Definitions	<code>#define NUMBER_OF_KEYS 4</code>
Declaration and initialization of public global variables	<code>bool incorrectCode = false;</code> <code>int buttonBeingCompared = 0;</code> <code>char codeSequence[NUMBER_OF_KEYS] = {'1','1','0','0'};</code>
Section	Lines that were removed
Declaration and initialization of public global variables	<code>char receivedAButton = 0;</code> <code>char receivedBButton = 0;</code> <code>char receivedCButton = 0;</code> <code>char receivedDButton = 0;</code>

Any time the compiler finds NUMBER_OF_KEYS, it will replace it for the corresponding number 4. In this way, the length of the code can be modified, changing only the definition of NUMBER_OF_KEYS.



NOTE: In this book, names of defines are stylized using the CONSTANT_CASE (also known as MACRO_CASE or SCREAMING_SNAKE_CASE), as in NUMBER_OF_KEYS.

In the section “Declaration and initialization of public global variables,” the array `codeSequence` is declared and initialized with the values '1', '1', '0', and '0'. This means that in the first position of the array the value '1' is stored, in the second position of the array the value '1' is stored, in the third position '0' is stored, and in the last position '0' is stored.

It can be seen in Code 2.17 that in the case corresponding to '4', there is a `for` loop (line 14). The `for` loop allows the programmer to create repetitive blocks that are executed a given number of times. This `for` loop is started with `buttonBeingCompared=0` and is concluded when the condition “`buttonBeingCompared < NUMBER_OF KEYS`” becomes false. In every loop of the `for` loop the variable `buttonBeingCompared` is incremented in order to compare the variable located in the next position of the array `codeSequence`. This is done by means of `buttonBeingCompared++` in the `for` statement.

In this particular example, to assess the code sequence entered by means of the PC keyboard the `for` loop is executed four times. In each of the repetitions, the received character is compared with the corresponding position of the array `codeSequence`. If the key entered is neither “1” or “0”, then `incorrectCode` is set to true (line 30).

Finally, depending on the state of `incorrectCode`, OFF is assigned to `alarmState` and 0 is assigned to `numberOfIncorrectCodes`, or the Incorrect code LED is turned on and the variable `numberOfIncorrectCodes` is incremented by one. Note that in line 42, “`++`” is now used to increment the value of `numberOfIncorrectCodes` by one. This is used to make the code more compact.

```

1 case '4':
2     uartUsb.write( "Please enter the code sequence.\r\n", 33 );
3     uartUsb.write( "First enter 'A', then 'B', then 'C', and ", 41 );
4     uartUsb.write( "finally 'D' button\r\n", 20 );
5     uartUsb.write( "In each case type 1 for pressed or 0 for ", 41 );
6     uartUsb.write( "not pressed\r\n", 13 );
7     uartUsb.write( "For example, for 'A' = pressed, ", 32 );
8     uartUsb.write( "'B' = pressed, 'C' = not pressed, ", 34 );
9     uartUsb.write( "'D' = not pressed, enter '1', ", 40 );
10    uartUsb.write( "then '0', and finally '0'\r\n\r\n", 29 );
11
12    incorrectCode = false;
13
14    for ( buttonBeingCompared = 0;
15          buttonBeingCompared < NUMBER_OF_KEYS;
16          buttonBeingCompared++ ) {
17
18        uartUsb.read( &receivedChar, 1 );
19        uartUsb.write( "*", 1 );
20
21        if ( receivedChar == '1' ) {
22            if ( codeSequence[buttonBeingCompared] != 1 ) {
23                incorrectCode = true;
24            }
25        } else if ( receivedChar == '0' ) {
26            if ( codeSequence[buttonBeingCompared] != 0 ) {
27                incorrectCode = true;
28            }
29        } else {
30            incorrectCode = true;
31        }
32    }
33
34    if ( incorrectCode == false ) {
35        uartUsb.write( "\r\nThe code is correct\r\n\r\n", 25 );
36        alarmState = OFF;
37        incorrectCodeLed = OFF;
38        numberOfIncorrectCodes = 0;
39    } else {
40        uartUsb.write( "\r\nThe code is incorrect\r\n\r\n", 27 );
41        incorrectCodeLed = ON;
42        numberOfIncorrectCodes++;
43    }
44    break;

```

Code 2.17 Details of the new lines in the function `uartTask()`.

Proposed Exercises

- How can the code sequence be changed?

Answers to the Exercises

- The array `codeSequence` in the section “Declaration and initialization of public global variables” should be modified.

Example 2.5: Change the Alarm Turn Off Code Using the PC

Objective

Develop more complex programs using *for* loops and arrays.

Summary of the Expected Behavior

The expected behavior is the same as Example 2.4, but now the code can be changed from the PC.

Test the Proposed Solution on the Board

Import the project “Example 2.5” using the URL available in [1], build the project, and drag the *.bin* file onto the NUCLEO board. Open the serial terminal. Press “5” on the PC keyboard and look at the message that appears on the serial terminal, indicating that a new code sequence can be set. Enter a new code, for instance, “0”, then “0”, then “1”, and finally “1”, and look at the message that appears on the serial terminal indicating that the new code has been created. Press the button connected to D2 that represents gas detection. Press “4” on the PC keyboard. Enter the new code. Check that the Alarm LED is turned off. Press the button connected to D2. To check that the control panel code has also been changed, enter the new code with those buttons by pressing the buttons connected to D6 and D7 and the B1 USER button at the same time. Check that the Alarm LED is turned off.

Discussion of the Proposed Solution

Following the same logic as in Example 2.4, a new case is added to the switch: case ‘5’. The reader will notice that it is very similar to case ‘4’. The new code sequence is stored in the array *codeSequence*. The details of the implementation are discussed below.

Implementation of the Proposed Solution

In this example, some lines were added to the program of Example 2.4, as shown in Table 2.5. Note that *areEqual()* is the first function in this book that returns a value (a Boolean), as is discussed below.

Table 2.5 Lines that were added from the code used in Example 2.4.

Section	Lines that were added
Declaration and initialization of public global variables	char buttonsPressed[NUMBER_OF_KEYS] = {'0','0','0','0'};
Declarations (prototypes) of public functions	bool areEqual();
Function	Lines that were added
availableCommands	uartUsb.write("Press '5' to enter a new code\r\n\r\n", 33);

In this example, *alarmDeactivationUpdate()* is modified to allow the user to change the code. Code 2.18 shows the fragment of the code corresponding to case '5' of the *switch* statement within *uartTask()* that implements this change. The code sequence is loaded into the array *codeSequence* using the *for* loop.

In Code 2.19, the condition in line 7 is the same as in line 78 of Code 2.5. The difference is that lines 8 to 11 are used to assign the value of each button (A to D) to the corresponding positions of the array *buttonsPressed*. Then the function *areEqual()*, shown in Code 2.20, is used to compare each position of *buttonsPressed* with the corresponding position of *codeSequence*.

In Code 2.20, it is shown how *areEqual()* is implemented using a *for* loop indexed by the local variable *i*, where one after the other each of the positions of *codeSequence* and *buttonsPressed* are compared. It is important to note that the return value of the function is implemented in line 7 or line 11, depending on the result of the *if* statement of line 6. Lines 18 to 22 are used to store the number 1 or 0 in the corresponding position of *codeSequence* depending on the value of *receivedChar*, in order to be able to use *areEqual()* to compare *codeSequence[i]* and *buttonsPressed[i]*.



NOTE: If the key pressed is not "1" or "0", then the value stored at *codeSequence[buttonBeingCompared]* is not modified, as can be seen between lines 19 and 23 of Code 2.19.

```

1  case '5':
2      uartUsb.write( "Please enter new code sequence\r\n", 32 );
3      uartUsb.write( "First enter 'A', then 'B', then 'C', and ", 41 );
4      uartUsb.write( "finally 'D' button\r\n", 20 );
5      uartUsb.write( "In each case type 1 for pressed or 0 for not ", 45 );
6      uartUsb.write( "pressed\r\n", 9 );
7      uartUsb.write( "For example, for 'A' = pressed, 'B' = pressed,", 46 );
8      uartUsb.write( " 'C' = not pressed,", 19 );
9      uartUsb.write( "'D' = not pressed, enter '1', then '1', ", 40 );
10     uartUsb.write( "then '0', and finally '0'\r\n\r\n", 29 );
11
12    for ( buttonBeingCompared = 0;
13          buttonBeingCompared < NUMBER_OF_KEYS;
14          buttonBeingCompared++ ) {
15
16        uartUsb.read( &receivedChar, 1 );
17        uartUsb.write( "*", 1 );
18
19        if ( receivedChar == '1' ) {
20            codeSequence[buttonBeingCompared] = 1;
21        } else if ( receivedChar == '0' ) {
22            codeSequence[buttonBeingCompared] = 0;
23        }
24    }
25
26    uartUsb.write( "\r\nNew code generated\r\n\r\n", 24 );
27    break;

```

Code 2.18 Details for case "5" of the switch statement of *uartTask()*.

```
1 void alarmDeactivationUpdate()
2 {
3     if ( numberOfIncorrectCodes < 5 ) {
4         if ( aButton && bButton && cButton && dButton && !enterButton ) {
5             incorrectCodeLed = OFF;
6         }
7         if ( enterButton && !incorrectCodeLed && alarmState ) {
8             buttonsPressed[0] = aButton;
9             buttonsPressed[1] = bButton;
10            buttonsPressed[2] = cButton;
11            buttonsPressed[3] = dButton;
12            if ( areEqual() ) {
13                alarmState = OFF;
14                numberOfIncorrectCodes = 0;
15            } else {
16                incorrectCodeLed = ON;
17                numberOfIncorrectCodes++;
18            }
19        }
20    } else {
21        systemBlockedLed = ON;
22    }
23 }
```

Code 2.19 Details of the function `alarmDeactivationUpdate()`.

```
1 bool areEqual()
2 {
3     int i;
4
5     for ( i = 0; i < NUMBER_OF_KEYS; i++ ) {
6         if ( codeSequence[i] != buttonsPressed[i] ) {
7             return false;
8         }
9     }
10
11    return true;
12 }
```

Code 2.20 Details of the function `areEqual()`.

Proposed Exercises

1. What should be modified in order to implement a five-key code?
2. Is it possible to implement a code of any arbitrary length?

Answers to the Exercises

1. `NUMBER_OF_KEYS` should be changed to 5. The text in the request should be modified. Some other parts of the code should also be modified, for example the reading of the buttons inside the `switch` statement.
2. There are many limitations. For example, the number of buttons that can be connected to the NUCLEO board using the technique explained in Chapter 1.

2.3 Under the Hood

2.3.1 Basic Principles of Serial Communication

In the examples of this chapter, information was exchanged between a PC and the NUCLEO board by means of a USB cable. This section explains how this information exchange is implemented.



WARNING: In this subsection, the fundamental concepts of serial communication are presented. In the communication between the PC and the NUCLEO board, these concepts are applied, but there are some other details that are not covered in this subsection. USB works very differently than UART.

Most of the wired connections used nowadays, like USB, HDMI, and Ethernet, are based on what is called *serial communication*. The details of how serial communication is implemented in each of those cases is quite complex, but for now it is enough to get the basic idea behind UART serial communication.

UART serial communication between two devices, A and B, in its most common setup requires three wires, as shown in Figure 2.12. One wire is used to establish a 0 volts reference (usually called *Ground* or *GND*) between both devices. A second wire is used to transmit the information from A to B (*TxA-RxB*, standing for Transmitter A - Receiver B), and a third wire is used to transmit information from B to A (*TxB-RxA*, standing for Transmitter B - Receiver A).

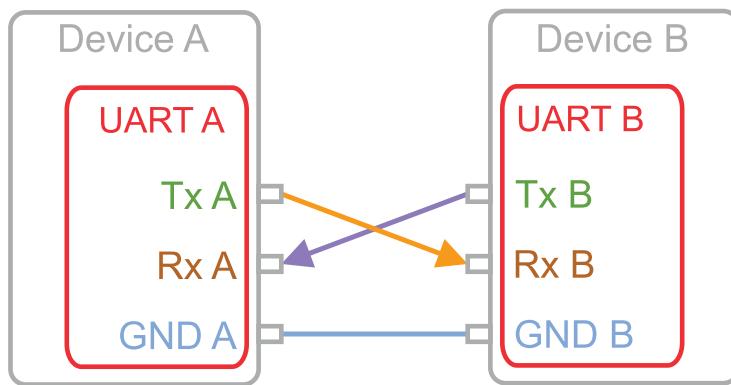


Figure 2.12 Basic setup for a serial communication between two devices.

Initially, the wires that are used to transmit information between the devices are in an *idle state*. This idle state is indicated by means of a previously agreed value, for example 3.3 volts. Then, if, for example, device A wants to start a message transmission to device B, it can indicate this by means of asserting 0 volts in the cable *TxA-RxB*. This notification is called a *start bit* and is shown in Figure 2.13. In this way, device B realizes that device A will send a message using the cable *TxA-RxB*.

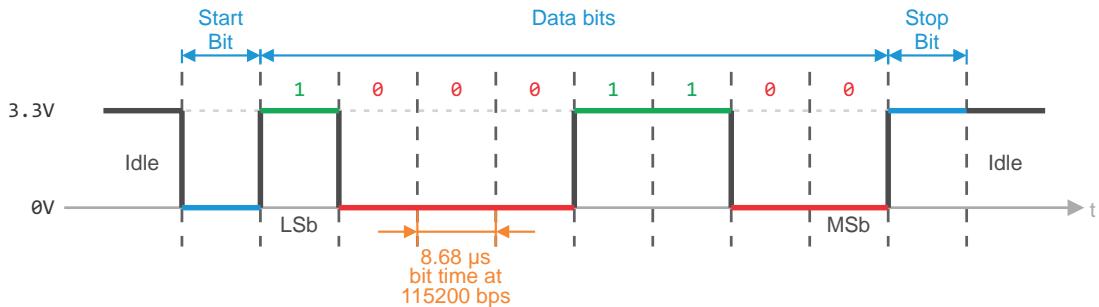


Figure 2.13 Basic sequence of a serial communication between two devices.

After sending the start bit, device A sends the first bit of the character. Frequently, the *little endian* format is used, which implies that the first bit that is sent is the first starting from the right. The first bit starting from the right is called the *Least Significant bit* or LSb.



WARNING: In this book, “LSb” is used to indicate the Least Significant bit. It should be noted that in some books LSB is used to indicate the Least Significant Bit.

For example, if the bits “00110001” represent a given character to be sent, the first bit that is sent is the 1 on the right (the LSb). Device A holds 3.3 volts in the cable TxA-RxB, as indicated by the LSb in Figure 2.13. The next bit to be sent is the 0 that is in the second position starting from the right. For this purpose, device A holds 0 volts in the cable TxA-RxB, as indicated in Figure 2.13.

In a similar way the remaining five bits are sent one after the other from device A to device B. The last bit to be sent is called the *Most Significant bit* or MSb.

Arrays of eight bits, called *bytes*, are sent in each transmission because most microcontrollers and computers internally organize the information in sets of integer multiples of eight bits (i.e., 8, 16, 32, 64, etc.).

There might be a *parity bit* sent after the eight bits if both devices have been previously configured to use this feature. This is not the case in the configuration used in this chapter, so this topic is not explained now.

Finally, device A sends a *stop bit*, to indicate that the transmission is over. This is achieved by means of setting 3.3 volts in the cable TxA-RxB as shown in Figure 2.13. In this way, for every byte of data transmitted, there are actually ten bits being sent: a start bit, eight data bits, and one stop bit.

In the examples of this chapter there was a parameter called “baud rate” that was configured in the serial terminal to “115200”. This means that 115,200 bits are transmitted every second. That is, the time holding each of those bits high (3.3 volts) or low (0 volts) is $1/(115,200 \text{ bps})$ or $8.68 \mu\text{s}$ per bit.

Given that ten bits for every byte of data sent are transmitted, at 115,200 bps, there are 11,520 bytes being sent per second.

The baud rate is a very important parameter because it allows the device that is receiving the bits to know when to read the Rx digital input to get a new bit.



WARNING: In some systems the communication can be implemented using 5 volts or +/- 12 volts signals. Those voltage levels may damage the NUCLEO board.



TIP: Sequences like the one shown in Figure 2.13 can be seen each time the NUCLEO board and the PC exchange messages by means of connecting an oscilloscope or a logic analyzer to the pins Tx and Rx of CN5, as shown in Figure 2.14. It is not explained here how to use oscilloscopes and logic analyzers because that topic is beyond the scope of this book.

Proposed Exercises

1. In the basic sequence of a serial communication between two devices shown in Figure 2.12, how can Device B be sure that the received information has no errors?
2. In the scheme shown in Figure 2.12, how can Device A be sure that Device B received the information with no errors?
3. Assuming 115,200 bps serial communication, how long will it take to send a 1 MB file?

Answers to the Exercises

1. The only way for Device B to be sure that there are no bits with errors is if there is some kind of verification. In the implementation of this verification, both Devices, A and B, must be involved. This will be explained in more advanced chapters of this book.
2. Device B should work with Device A in order to ascertain this. The parity bit can be used for this purpose.
3. 1 MB is equal to 1,000,000 bytes; thus, it will take 86 seconds ($1,000,000 \text{ bytes} / 11,520 \text{ bytes/second}$).



NOTE: In the following chapters, other communications protocols will be explored, which will allow higher transfer rates.

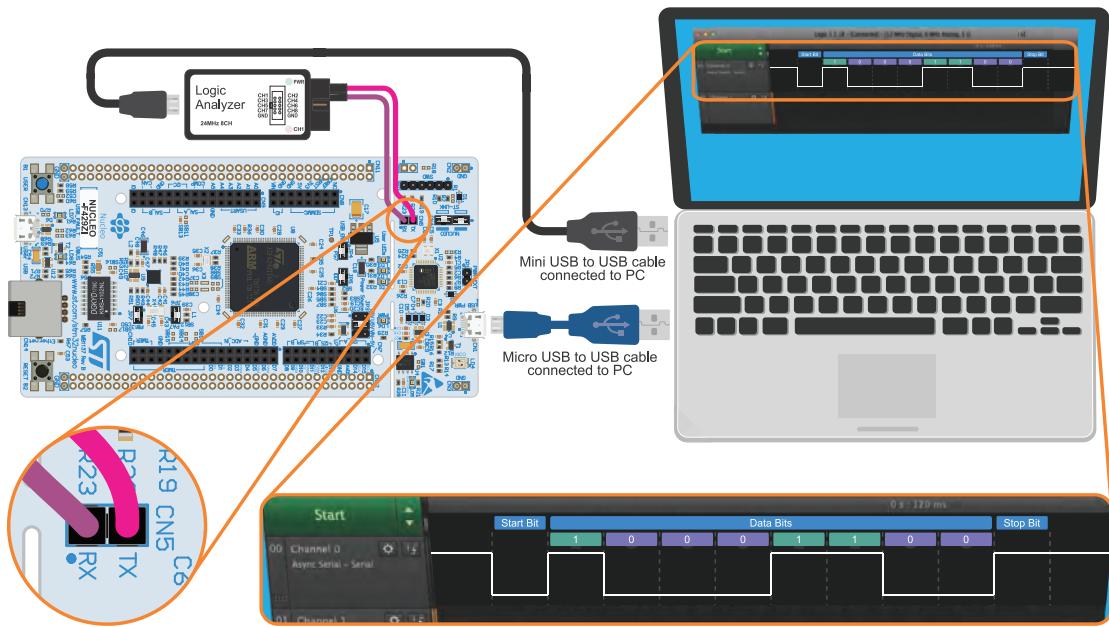


Figure 2.14 The bits transmitted or received by the NUCLEO board UART can be seen by connecting an oscilloscope or a logic analyzer on CN5. These bits do not correspond to the USB protocol.

2.4 Case Study

2.4.1 Industrial Transmitter

In this chapter, the NUCLEO board communicated with a PC by means of the UART of the microcontroller, a USB cable, and a serial terminal, as shown in Figure 2.15. In this way, the state of the gas and over temperature detectors were transmitted to the PC, the alarm could be turned off from the PC, and the password could be changed from the PC.

A brief of a commercial “Industrial transmitter” built with Mbed containing some similar features can be found in [5] and is shown in Figure 2.15. It can be seen that the “Petasense Motes” send the data to a server indicated by the “Petasense Cloud.” Then, a program running on a PC gets the data from the “Petasense Cloud” and shares it with a software application called “Pi Vision.”

By comparing both implementations shown in Figure 2.15, it can be seen that there is a device sending data and a PC used to visualize those data. In the case of the smart home system, the data is transmitted directly from the device to the PC by means of a serial communication based on a UART. In the case of the “Petasense Motes,” there is a microcontroller (“MCU”) that gets the values of vibration, ultrasound, current, and temperature and transmits this information using serial communication to a Wi-Fi module. The Wi-Fi module sends the information to the Petasense Cloud using the internet. The last chapters of this book will explain how to send information to a PC using a Wi-Fi module.

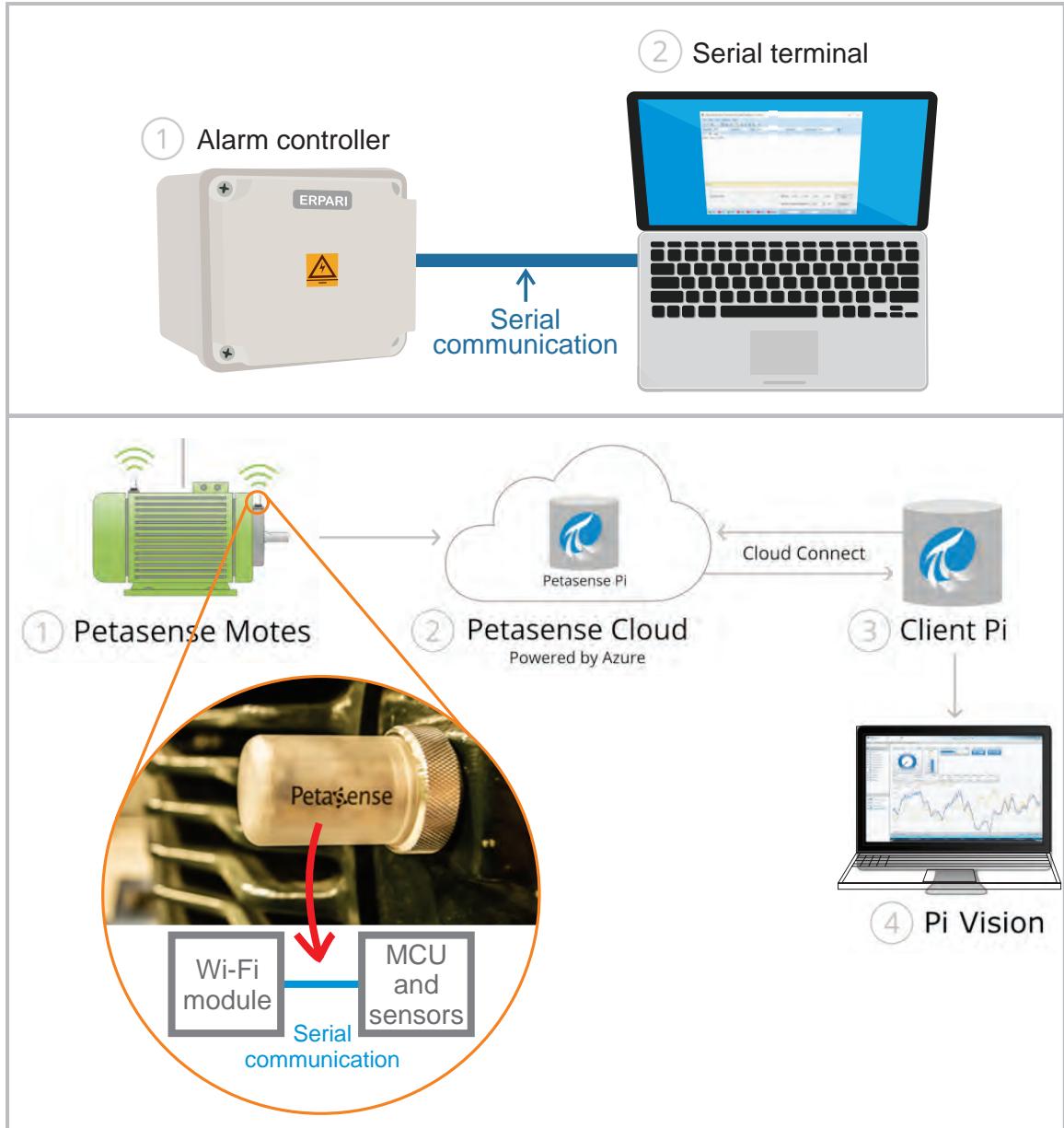


Figure 2.15 Top, the smart home system. Bottom, a diagram of the "Industrial transmitter."

Proposed Exercises

1. How is the information transmitted from the Petasense Motes to the Petasense Cloud?
2. Is a serial terminal being used on the PC?

Answers to the Exercises

1. By carefully looking in Figure 2.15 at the Petasense Motes, it is possible to see the Wi-Fi symbol. This is used to indicate that Wi-Fi is being used to transmit the information from the Petasense Motes to the Petasense Cloud. Moreover, in [5] it can be seen that the connectivity being used is “Ethernet/Wi-Fi.”
2. No, there is no serial terminal involved. This is because the microcontroller that gets the values of vibration, ultrasound, current, and temperature transmits this information using UART to a Wi-Fi module.

References

- [1] “GitHub - armBookCodeExamples/Directory”. Accessed July 9, 2021.
<https://github.com/armBookCodeExamples/Directory/>
- [2] “Doxygen: Doxygen”. Accessed July 9, 2021.
<https://www.doxygen.nl/index.html>
- [3] “DigitalIn - API references and tutorials | Mbed OS 6 Documentation”. Accessed July 9, 2021.
<https://os.mbed.com/docs/mbed-os/v6.12/apis/digitalin.html>
- [4] “UnbufferedSerial - API references and tutorials | Mbed OS 6 Documentation”. Accessed July 9, 2021.
<https://os.mbed.com/docs/mbed-os/v6.12/apis/unbufferedserial.html>
- [5] “Industrial Transmitter | Mbed”. Accessed July 9, 2021.
<https://os.mbed.com/built-with-mbed/industrial-transmitter/>

Chapter 3

Time Management and
Analog Signals

3.1 Roadmap

3.1.1 What You Will Learn

After you have studied the material in this chapter, you will be able to:

- Describe how to connect sensors to the NUCLEO board using an analog signal interface.
- Develop programs to get and manage analog signals with the NUCLEO board.
- Develop programs that use parameter passing in C/C++ functions.
- Summarize the fundamentals of analog to digital conversion.
- Introduce time management in microcontroller programs.
- Implement basic character string management.

3.1.2 Review of Previous Chapters

In previous chapters, the reader learned how to communicate between the NUCLEO board and a PC using UART serial communication. A broad variety of functions were implemented, and much of that functionality relayed the state of the gas detector and the over temperature detector. These were not sensors, but representations by means of buttons.

Also, in the implementation within previous chapters, the Alarm LED was activated because of gas detection, or over temperature detection, or both being detected at the same time. The user had no information from looking at the Alarm LED about why the LED was active.

3.1.3 Contents of This Chapter

This chapter introduces a way to indicate to the user whether an alarm is caused by gas detection, over temperature detection, or simultaneous gas and over temperature detection. This is based on controlling the blinking rate of the Alarm LED. It will be explained how to utilize time with the NUCLEO board, and the usage of delays is introduced. In Example 3.1 and Example 3.2, two different ways of implementing a given delay are shown in order to compare the responsiveness of both techniques.

It will also be explained how to measure analog signals with the NUCLEO board using one of the *analog to digital converters* (ADCs) included in the STM32 microcontroller. By means of a potentiometer, an LM35 temperature sensor, and an MQ-2 gas sensor module, the concepts of analog to digital converters are explored. The over temperature detection is done using a temperature sensor and the gas detection is implemented using a gas sensor. It will also be shown how to activate a 5 V buzzer using one of the 3.3 V digital outputs of the NUCLEO board. Finally, the basic principles of analog to digital conversion are explained and a case study related to temperature measurement is shown in the Under the Hood and Case Study sections, respectively.

3.2 Analog Signals Measurement with the NUCLEO Board

3.2.1 Connect Sensors, a Potentiometer, and a Buzzer to the Smart Home System

In this chapter, an LM35 temperature sensor [1], an MQ-2 gas sensor module [2], a potentiometer, and a buzzer are connected to the smart home system, as shown in Figure 3.1. The aim of this setup is to introduce the reading of analog signals.

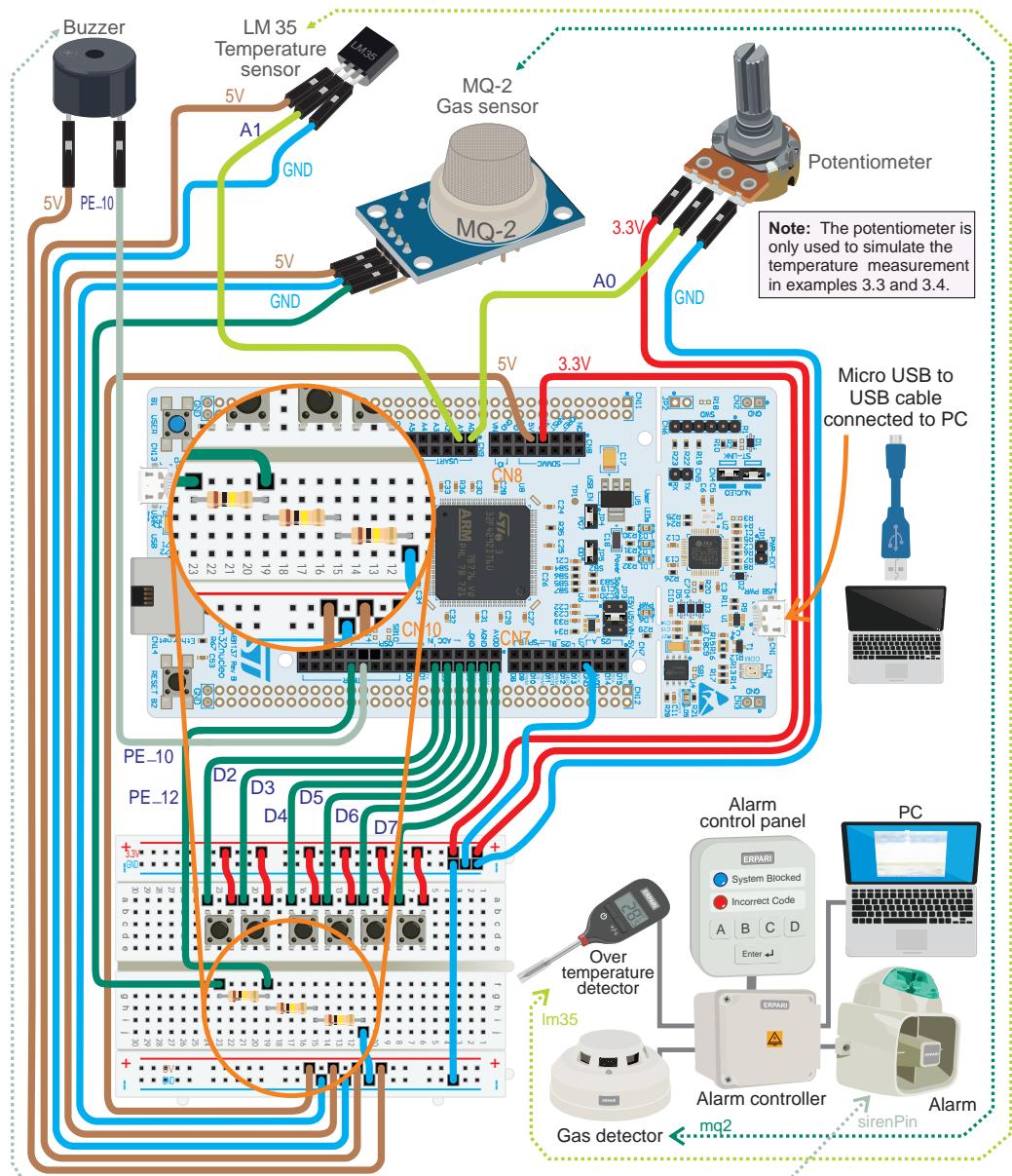


Figure 3.1 The smart home system is now connected to a temperature sensor, a gas detector, a potentiometer, and a buzzer.



NOTE: The LM35 temperature sensor, the MQ-2 gas sensor, and the buzzer must be connected to 5 V, as indicated in Figure 3.1.



WARNING: To connect modules and elements, unplug the USB power supply from the NUCLEO board, and prior to reconnecting the USB power supply check that the connections are made properly and safely.

WARNING: Some MQ-2 modules have a different pinout. Follow the VCC, GND, and DO labels of the module when making the connections to 5 V, GND, and PE_12, respectively.

Figure 3.2 shows a potentiometer. This component allows the user vary the resistance between terminal 2 and terminals 1 and 3, depending on the angular position of its knob. It will be used in this chapter to explore some concepts beyond analog to digital conversion.

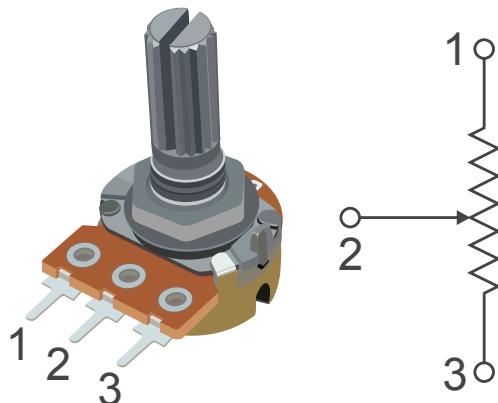


Figure 3.2 A typical potentiometer and its corresponding electrical diagram.

The most common way to connect a potentiometer to a development board is as shown in Figure 3.3. It can be seen that terminal (1) is connected to a 3.3 Volt supply voltage, terminal (3) is connected to GND, and terminal (2) is connected to an analog input pin, which in this case is the A0 pin.



NOTE: It is recommended to use a potentiometer with a full-scale resistance of 10 k Ω . If the resistance is too small, the NUCLEO board could be damaged, and if it is too big, the measurement could be unstable.

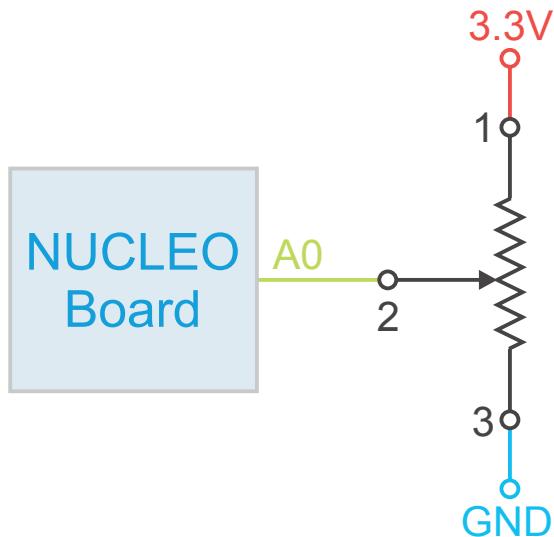


Figure 3.3 Diagram of the connection of the potentiometer to the NUCLEO board.

The LM35 temperature sensor is an integrated circuit that delivers an output voltage linearly proportional to its temperature. It has three terminals, which are identified with the names GND, +VS, and VOUT, as shown in Figure 3.4. These names stand for Ground, Voltage Supply, and Voltage Out, respectively.



Figure 3.4 The LM35 temperature sensor in a TO-92 package.



TIP: If necessary, the LM35 in the TO-220 package described in [3] can be used. In that case, the code remains the same, but the reader must properly identify the terminals GND, +VS, and VOUT [1]. An LM35 from any brand can be used in any of its different order number options (e.g., LM35CZ, LM35DZ, etc.).

The most basic setup for the LM35 temperature sensor is as shown in Figure 3.5. This setup is the one used in Figure 3.1 and provides an output signal in VOUT that increases at a rate of 10 mV/°C (millivolts per Celsius degree) in the range of 2 to 150°C, as indicated in [1].

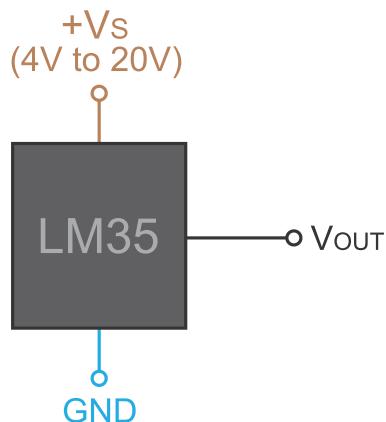


Figure 3.5 Basic setup for the LM35 temperature sensor.

In Table 3.1, some examples of the voltage at V_{OUT} are shown for different temperatures. For the convenience of readers from the United States, temperatures are also expressed in degrees Fahrenheit. To convert a temperature expressed in °C into degrees Fahrenheit, it must be multiplied by 9/5 and 32 must be added.

Table 3.1 Examples of the voltage at V_{OUT} using the connection shown in Figure 3.5.

Temperature		Voltage at V _{OUT}
[°C]	[°F]	[mV]
2	35.6	20
3	37.4	30
10	50.0	100
30	86.0	300
150	302	1500

Figure 3.6 shows a diagram of the connection of the LM35 temperature sensor to the NUCLEO board.

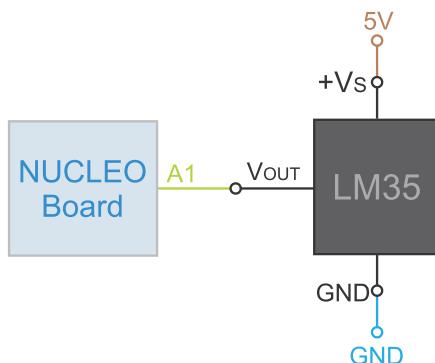


Figure 3.6 Diagram of the connection of the LM35 to the NUCLEO board.



NOTE: In this particular case, the sensor is connected directly to the NUCLEO board to simplify the circuitry. In a real application, conditioning circuits are often used. Those circuits allow the user, for example, to make use of the entire signal range of the analog input (0 to 3.3 V).

In Figure 3.7, the MQ-2 gas sensor is connected to the NUCLEO board. This sensor is supplied with 5 V and detects LPG, i-butane, propane, methane, alcohol, hydrogen, and smoke. Its AOUT (Analog Output) pin is left unconnected, and the DOUT (Digital Output) pin is used. The DOUT pin provides 0 V when gas presence over a certain concentration is detected and 5 V when the concentration is below a certain level.



WARNING: The maximum voltage that can be applied to most of the NUCLEO board pins without damage is 4 V, while the DOUT pin provides 5 V. Therefore, the NUCLEO board can be damaged if DOUT is connected without a voltage limitation. The resistors R1, R2, and R3 shown in Figure 3.7 are used to attenuate DOUT by a factor of 2/3, which produces 3.3 V when DOUT is 5 V and 0 V when DOUT is 0 V.



NOTE: In this particular case, pin PE_12 is a 5 V tolerant I/O, so strictly speaking the resistor divider is not needed. However, it is included to show how to proceed when a given input pin is not 5 V tolerant.

NOTE: In Figure 3.7, 100 k Ω resistors are shown, but if they are not available they can be replaced with three resistors of similar resistance values, provided they are in the range of 47 to 150 k Ω .

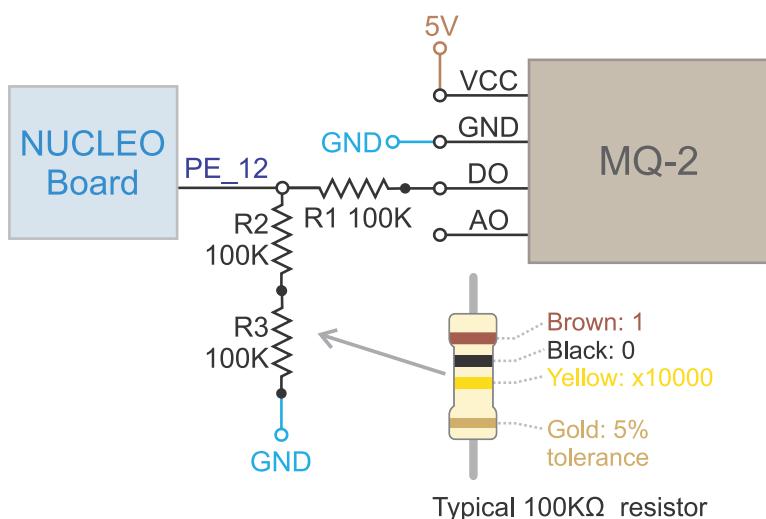


Figure 3.7 Diagram of the connection of the MQ-2 to the NUCLEO board.



NOTE: In the next chapters, different techniques to adapt voltage levels are introduced as they are needed. In this way the reader will be able to compare the techniques in terms of cost, performance, complexity, etc.

In Figure 3.8, the buzzer connection is summarized. The buzzer represents a siren that is activated when there is an alarm situation, such as gas or over temperature detection.

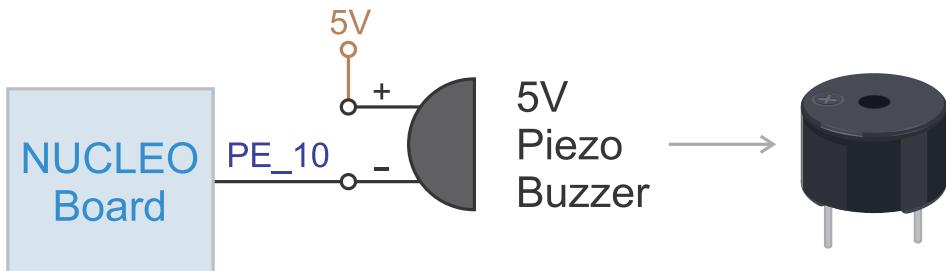


Figure 3.8 Diagram of the connection of the buzzer to the NUCLEO board.



NOTE: With the connection shown in Figure 3.8, the buzzer is activated when LOW is assigned to the PE_10 pin. In this way, a 5 V device such as the buzzer can be activated using a 3.3 V digital output of the NUCLEO board. To turn off the buzzer, the PE_10 pin is configured as *open drain output*, in order to not assert any voltage to the buzzer “-” pin, causing the buzzer to turn off. Open drain means that the output is the drain terminal of a MOSFET transistor [4] without a pull-up resistor. It can establish 0 V if the MOSFET is activated and behaves like an open circuit otherwise. This is implemented in Example 3.5.

3.2.2 Test the Operation of the Sensors, the Potentiometer, and the Buzzer

This subsection will explain how to load a program onto the STM32 microcontroller in order to test if the components that were connected are working properly. This will be achieved by using the serial terminal to show the reading of the LM35 temperature sensor, the potentiometer, and the status of the DOUT signal of the MQ-2 sensor. The buzzer will be activated or deactivated depending on the readings. For this purpose, the .bin file of the program “Subsection 3.2.2” should be downloaded from the URL available in [5] and the .bin file dragged onto the NUCLEO board.

In Table 3.2, the available commands for the program that will be used in this subsection are shown. If the key “a” is pressed on the PC keyboard, then the reading of the analog input A0 of the NUCLEO board is continuously displayed on the serial terminal. In this case, because of the Mbed OS function being used, this is a value between 0.0 and 1.0, depending on the angular position of the knob. Rotate the knob of the potentiometer from one side to the other in order to test if the potentiometer is working correctly and is well connected to the NUCLEO board. After verifying that the potentiometer is working correctly, press “q” to quit this verification and continue with the next step.

Table 3.2 Available commands of the program used to test the LM35 temperature sensor and the potentiometer.

Key pressed	Information that is displayed in the serial terminal
a	Reading from the analog input A0 of the NUCLEO board (a value between 0.0 and 1.0)
b	Reading from the analog input A1 of the NUCLEO board (a value between 0.0 and 1.0)
c	Reading of the temperature measured by the LM35 expressed in °C
d	Reading of the temperature measured by the LM35 expressed in °F
e	Reading of the temperature measured by the LM35 expressed in °C and reading of the potentiometer scaled by the same factor
f	Reading of the temperature measured by the LM35 expressed in °F and reading of the potentiometer scaled by the same factor
g	Reading of the DOUT signal of the MQ-2 gas sensor
q	Quit the last entered command

Press the “b” key to get the reading at the analog input A1 of the NUCLEO board. This input is connected to the LM35 temperature sensor and will be a value between 0 and 1. Press “q” to continue with the next step.

When the “c” key is pressed, the reading of the temperature measured by the LM35, which is connected to A1, is displayed on the serial terminal and expressed in °C. The formula used to convert the analog reading to temperature expressed in degrees Celsius is:

$$\text{Temperature } [{}^{\circ}\text{C}] = \frac{\text{Analog Reading} \times 3.3\text{ V}}{0.01\text{ V}/{}^{\circ}\text{C}} \quad (1)$$

This formula indicates that the analog reading, which is a non-integer value between 0 and 1, is first multiplied by 3.3 V in order to get the corresponding voltage. This is due to the fact that the analog to digital converter of the NUCLEO board provides approximately 0.0 for a 0 V input and gives its maximum value (in this case 1.0) for a 3.3 V input. The analog to digital conversion used in the NUCLEO board will be explained in detail in the Under the Hood section. The result is then divided by 0.01 V/°C because the output signal VOUT of the LM35 increases by 10 mV/°C in the range of 2 to 150 °C.

The reader is encouraged to hold the LM35 between two fingers and to observe that the temperature displayed on the serial terminal becomes about 32 °C.

If the “d” key is pressed, then the temperature measured by the LM35 that is connected to A1 is displayed on the serial terminal expressed in °F. The conversion formula from Celsius to Fahrenheit is:

$$\text{Temperature } [{}^{\circ}\text{F}] = \frac{\text{Temperature } [{}^{\circ}\text{C}] \times 9}{5} + 32 \quad (2)$$

The commands that are activated by the keys “e” and “f” are provided only to compare the noise in the readings of A0 (connected to the potentiometer) and A1 (connected to the LM35 temperature sensor). In order to make the comparison more straightforward in these two commands, formula (1)

is applied to the readings from analog input A0, and the following formula is applied to A1 in order to scale its reading between 2 and 150:

$$\text{Value expressed in } [{}^{\circ}\text{C}] = \text{Analog Reading} \times 148 + 2 \quad (3)$$



NOTE: It should be clear that formula (1) has a physical meaning only when an LM35 temperature sensor is connected to the analog input. The result of formula (3), applied to the analog signal coming from the potentiometer, is only valid for the purpose of comparing the noise in the readings of A0 and A1 (i.e., there is no variation in temperature when the knob of the potentiometer is rotated).

Press the “e” key to get the readings at the analog inputs A0 and A1 of the NUCLEO board at the same time, as discussed above. Rotate the knob of the potentiometer in order to get a reading for the potentiometer similar to the reading obtained for the LM35.

Press the “f” key to get the readings at the analog inputs A0 and A1 of the NUCLEO board at the same time, as discussed above. This result is obtained by first applying formula (1) or formula (3) and then applying formula (2). The combination of formulas (1) and (2) will be used in some of the proposed exercises in this chapter.



NOTE: The examples in this chapter will explore how to reduce the noise in the measurements by means of averaging a set of consecutive readings.

Finally, press the “g” key to continuously print the reading of the DOUT signal from the MQ-2 gas sensor. The reading should be consistent with the state of the DO-LED shown in Figure 3.9: if gas is not detected, the DO-LED should be off and the message on the serial terminal should be “Gas is not being detected.” If gas is detected, the DO-LED should be on and the message on the serial terminal should be “Gas is being detected.” When gas is detected, the buzzer should sound.

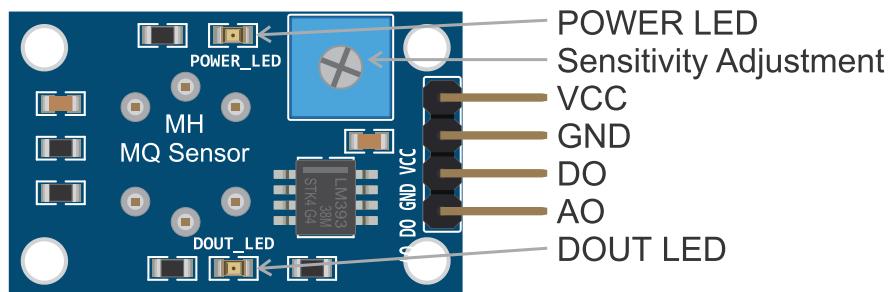


Figure 3.9 Diagram of the connection of the MQ-2 gas sensor module.

Modify the Sensitivity Adjustment of the MQ-2 gas sensor module (Figure 3.7) in one direction and then in the other. The DO-LED should turn on and off, as should the buzzer, and the message on the serial terminal should change between “Gas detected” and “Gas not detected.”

To adjust the Sensitivity Adjustment to an appropriate level, use a lighter. Press the button to open the gas valve without rotating the spark wheel. In this way, a small amount of gas will be released without producing a flame on the top of the lighter, and the sensitivity of the MQ-2 gas sensor module can be easily adjusted.



WARNING: Be careful not to rotate the spark wheel of the lighter in order to avoid lighting the flame.

Example 3.1: Indicate which Sensor has Triggered the Alarm

Objective

Introduce time management by means of delays.



NOTE: In this example, the sensors are still activated by means of the buttons connected to D2 and D3.

Summary of the Expected Behavior

If the alarm has been triggered by the gas detector, then the Alarm LED (LD1) should blink at a rate of one second (1,000 milliseconds) on and one second off. If the alarm has been triggered by the over temperature detector, the Alarm LED should blink at a rate of 500 ms on and 500 ms off. If gas is being detected and the temperature is above the maximum level, then LD1 should blink at a rate of 100 ms on and 100 ms off.

Test the Proposed Solution on the Board

Import the project “Example 3.1” using the URL available in [5], build the project, and drag the *.bin* file onto the NUCLEO board. Press the button connected to D2 in order to activate the alarm for gas detection. The Alarm LED (LD1) should start blinking, at a rate of one second on and one second off. Deactivate the alarm by simultaneously pressing buttons A + B + Enter on the control panel (i.e., D4 + D5 + B1 USER). Note that there is a slight delay in the response because of the way in which the program is implemented, as discussed below. Press the button connected to D3 in order to activate the alarm for over temperature detection. The Alarm LED should start blinking at a rate of half of a second on and half of a second off. Press the button connected to D2 in order to add a gas detection state. LD1 should blink faster, being one tenth of a second on and one tenth of a second off.

Discussion of the Proposed Solution

The proposed solution is based on the *delay()* function defined in the *arm_book_lib.h* library. This function is based on the Mbed OS *thread_sleep_for()* function and pauses the execution of the program, causing a delay. Depending on the situation, a delay with a different length can be used when turning the Alarm LED on and off. In this way, it can be indicated if the Alarm LED is active because of gas detection, over temperature detection, or the simultaneous detection of gas and over temperature.

Implementation of the Proposed Solution

The definitions and variables that were added to the program of Example 2.5 in order to implement the new functionality are shown in Table 3.3. It can be seen that three constants are defined that will be used in the delays. In the case of BLINKING_TIME_GAS_ALARM, the delay will be one second (1000 ms); in the case of BLINKING_TIME_OVER_TEMP_ALARM, the delay will be half of one second (500 ms); and in the case of BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM, the delay will be one tenth of a second (100 ms). There are also two public Boolean variables, *gasDetectorState* and *overTempDetectorState*. These variables will be used to store the state of the gas detector and the over temperature detector.

In Code 3.1, the new implementation of the function *alarmActivationUpdate()* is shown. On lines 3 to 6, it can be seen that if *gasDetector* is active, then *gasDetectorState* and *alarmState* are both set to ON. A similar piece of code is used from lines 7 to 10 regarding *overTempDetector*, *overTempDetectorState*, and *alarmState*. On line 11, *alarmState* is checked to see if it is ON. If it is ON, the state of the Alarm LED is toggled on line 12. Next, a delay is introduced in lines 13 to 19, whose length depends on the state of *gasDetectorState* and *overTempDetectorState*. If *alarmState* is OFF, then on line 21 the Alarm LED is turned off and the variables *gasDetectorState* and *overTempDetectorState* are set to OFF.

Table 3.3 Sections in which lines were added to Example 2.5.

Section	Lines that were added
Defines	#define BLINKING_TIME_GAS_ALARM 1000 #define BLINKING_TIME_OVER_TEMP_ALARM 500 #define BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM 100
Declaration and initialization of public global variables	bool gasDetectorState = OFF; bool overTempDetectorState = OFF;

```

1 void alarmActivationUpdate()
2 {
3     if( gasDetector ) {
4         gasDetectorState = ON;
5         alarmState = ON;
6     }
7     if( overTempDetector ) {
8         overTempDetectorState = ON;
9         alarmState = ON;
10    }
11    if( alarmState ) {
12        alarmLed = !alarmLed;
13        if( gasDetectorState && overTempDetectorState ) {
14            delay( BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM );
15        } else if ( gasDetectorState ) {
16            delay( BLINKING_TIME_GAS_ALARM );
17        } else if ( overTempDetectorState ) {
18            delay( BLINKING_TIME_OVER_TEMP_ALARM );
19        }
20    } else{
21        alarmLed = OFF;
22        gasDetectorState = OFF;
23        overTempDetectorState = OFF;
24    }
25 }
```

Code 3.1 Details of the new implementation of the function *alarmActivationUpdate()*.

Proposed Exercise

1. Is there any consequence for the program responsiveness if the delays are increased by a factor of ten?

Answer to the Exercise

1. Table 3.4 shows the values to be used to increase the delays by a factor of ten. The reader should repeat the steps detailed in the subsection “Implement the Proposed Solution on the Board” of Example 3.1 using these new values for BLINKING_TIME_GAS_ALARM, BLINKING_TIME_OVER_TEMP_ALARM, and BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM. It can be seen that the responsiveness of the program is severely affected by these long delays. Example 3.2 will show how to implement a change in the code in order to overcome this problem.

Table 3.4 Lines that were modified from Example 3.1.

Section	Lines that were added	
Definitions	#define BLINKING_TIME_GAS_ALARM	10000
	#define BLINKING_TIME_OVER_TEMP_ALARM	5000
	#define BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM	1000

Example 3.2: Increase the Responsiveness of the Program

Objective

Introduce a technique to avoid non-responsive behavior when long times are used in the delays.



NOTE: In this example, the sensors are still activated by means of the buttons connected to D2 and D3.

Summary of the Expected Behavior

The expected behavior is the same as in Example 3.1, but the new method of implementing delays leads to a more responsive behavior, even if long delays are used.

Test the Proposed Solution on the Board

Import the project “Example 3.2” using the URL available in [5], build the project, and drag the .bin file onto the NUCLEO board. Repeat the same steps that were described in the subsection “Test the Proposed Solution on the Board” of Example 3.1. The behavior should be exactly the same but exhibit a more responsive behavior.

Discussion of the Proposed Solution

The proposed solution is based on the idea of using a given number of short delays in order to achieve a longer delay. For instance, in this example ten 10 ms delays are consecutively used to achieve a

100 ms delay, or fifty 10 ms delays are consecutively used to achieve a 500 ms delay. In this way, every 10 ms, the buttons can be read in order to see if the user is asking for a given response. The result is that the program becomes much more responsive.



NOTE: This technique of counting small delays is the first approach that is used in this book in order to manage a program with pauses in a “non-blocking” way.

Implementation of the Proposed Solution

The new implementation of the *main()* function is shown in Code 3.2. It can be seen that a delay of *TIME_INCREMENT_MS* is included on line 9.

```
1 int main()
2 {
3     inputsInit();
4     outputsInit();
5     while (true) {
6         alarmActivationUpdate();
7         alarmDeactivationUpdate();
8         uartTask();
9         delay(TIME_INCREMENT_MS);
10    }
11 }
```

Code 3.2 Details of the new implementation of the main() function.

The lines that were added to Example 3.1 are shown in Table 3.5. It can be seen that *TIME_INCREMENT_MS* is defined as 10, and the integer variable *accumulatedTime* is declared and initialized to 0. As was discussed in “Discussion of the Proposed Solution,” 10 ms delays are accumulated to implement longer delays.

Table 3.5 Section where a line was added to Example 3.1.

Section	Lines that were added
Definitions	#define TIME_INCREMENT_MS 10
Declaration and initialization of public global variables	int accumulatedTime = 0;

Code 3.3 shows the new implementation used in *alarmActivationUpdate()*. Lines 1 to 10 are the same as the previous implementation used in Example 3.1. The difference is the way the delays are implemented inside the *if* statement in line 11 when *alarmState* is active (lines 11 to 35). In line 12, *accumulatedTime* is increased by *TIME_INCREMENT_MS*, accounting for the 10 ms delay introduced in the *main()* function.

Line 14 checks if *alarmState* is active due to the gas detector and the over temperature detector both being active. If so, *accumulatedTime* is checked to see if it has reached the time established by

BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM. If that is the case, *accumulatedTime* is set to 0 for the next iteration and the state of *alarmLed* is toggled.

The same behavior is implemented in lines 19 to 23 for the case in which *alarmState* is active only because of the gas detector. In the case of *alarmState* being active because of the over temperature detector alone, the executed statements are those between lines 24 and 28.

If *alarmState* is not true, then *alarmLed*, *gasDetectorState*, and *overTempDetector State* are set to OFF in lines 31 to 33.

In this way, the delays are always divided into pieces of 10 ms. As a consequence, the smart home system is never blocked for a long time waiting for the elapse of a long delay (i.e., 1000 ms or 500 ms).



NOTE: The 10 ms delay is not perceptible to human beings but is a long time for a microcontroller. More advanced time management techniques will be introduced later on in this book.

```

1 void alarmActivationUpdate()
2 {
3     if( gasDetector ) {
4         gasDetectorState = ON;
5         alarmState = ON;
6     }
7     if( overTempDetector ) {
8         overTempDetectorState = ON;
9         alarmState = ON;
10    }
11    if( alarmState ) {
12        accumulatedTimeAlarm = accumulatedTimeAlarm + TIME_INCREMENT_MS;
13
14        if( gasDetectorState && overTempDetectorState ) {
15            if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM ) {
16                accumulatedTimeAlarm = 0;
17                alarmLed = !alarmLed;
18            }
19        } else if( gasDetectorState ) {
20            if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_ALARM ) {
21                accumulatedTimeAlarm = 0;
22                alarmLed = !alarmLed;
23            }
24        } else if( overTempDetectorState ) {
25            if( accumulatedTimeAlarm >= BLINKING_TIME_OVER_TEMP_ALARM ) {
26                accumulatedTimeAlarm = 0;
27                alarmLed = !alarmLed;
28            }
29        }
30    } else{
31        alarmLed = OFF;
32        gasDetectorState = OFF;
33        overTempDetectorState = OFF;
34    }
35 }
```

Code 3.3 Details of the new implementation of the function *alarmActivationUpdate()*.

Proposed Exercise

- Given the new implementation used in Code 3.3, does the program become non-responsive if the delays are increased by a factor of ten?

Answer to the Exercise

- Table 3.6 shows the values to be used to result in an increase in the delays by a factor of ten. The reader should repeat the steps detailed in the subsection “Implement the Proposed Solution on the Board” of Example 4.1 using these new values for the delays. It can be seen that the responsiveness of the program is no longer affected by these long delays.

Table 3.6 Lines that were modified from Example 3.2.

Section	Lines that were added	
Defines	#define BLINKING_TIME_GAS_ALARM #define BLINKING_TIME_OVER_TEMP_ALARM #define BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM	10000 5000 1000

Example 3.3: Activate the Over Temperature Alarm by Means of the Potentiometer

Objective

Introduce the measurement of analog signals, the use of float variables, and the use of strings.

Summary of the Expected Behavior

The alarm should be activated when the knob of the potentiometer is rotated beyond half of its rotational travel. The corresponding reading should be shown on the serial terminal when the “p” key is pressed on the PC keyboard.

Test the Proposed Solution on the Board

Import the project “Example 3.3” using the URL available in [5], build the project, and drag the .bin file onto the NUCLEO board. Open the serial terminal. Press “p” on the PC keyboard and read the message that appears on the serial terminal. Rotate the knob in both directions and see how the values displayed on the serial terminal change in the range of 0 to 1. Rotate the knob until a reading of about 0.2 is obtained and deactivate the alarm by simultaneously pressing the A + B + Enter buttons on the control panel (i.e., D4 + D5 + B1 USER). Then, slowly rotate the knob until a reading above 0.5 is obtained. The alarm should turn on.

Discussion of the Proposed Solution

The proposed solution is based on the reading of the analog signal that is provided by the central terminal of the potentiometer. This signal, which is proportional to the rotation of the knob, is connected to the analog input 0 (A0) of the NUCLEO board. If the reading is below 0.5, then *overTempDetector* is set to OFF, and if it is above 0.5, *overTempDetector* is set to ON.

Implementation of the Proposed Solution

The object and variables that were added to the program of Example 3.2 in order to implement the new functionality are shown in Table 3.7. It can be seen that POTENTIOMETER_OVER_TEMP_LEVEL has been defined as 0.5. It is declared as an analog input object called *potentiometer* and assigned to the analog input 0 (A0) of the NUCLEO board. Finally, a Boolean global variable *overTempDetector* and a global variable *potentiometerReading* of type float are declared. *overTempDetector* is used to keep track of the current state of the over temperature detector, which is implemented by the potentiometer. In this example, the float variable is used to store the values in the range of 0.0 to 1.0 that are obtained when the analog A0 is read. A float variable can store a value in the range of $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$.

Table 3.8 and Table 3.9 show some lines that were removed from Example 3.2. Code 3.4 shows the code used to get the potentiometer reading. This code is included in the *alarmActivationUpdate()* function. As can be seen, in line 3 the reading is obtained by *potentiometer.read()*. In line 5, the potentiometer reading is compared with POTENTIOMETER_OVER_TEMP_LEVEL. Depending on the result, the state of *overTempDetector* is set to ON or OFF in lines 6 or 8. The remaining lines (10 to 44) were not changed.

Table 3.7 Sections in which lines were added to Example 3.2.

Section	Lines that were added
Definitions	#define POTENTIOMETER_OVER_TEMP_LEVEL 0.5
Declaration and initialization of public global objects	AnalogIn potentiometer(A0);
Declaration and initialization of public global variables	bool overTempDetector = OFF; float potentiometerReading = 0.0;

Table 3.8 Sections in which lines were removed from Example 3.2.

Section	Lines that were removed
Declaration and initialization of public global objects	DigitalIn overTempDetector(D3);

Table 3.9 Functions in which lines were removed from Example 3.2.

Function	Lines that were removed
void inputsInit()	overTempDetector.mode(PullDown);

```

1 void alarmActivationUpdate()
2 {
3     potentiometerReading = potentiometer.read();
4
5     if ( potentiometerReading > POTENTIOMETER_OVER_TEMP_LEVEL ) {
6         overTempDetector = ON;
7     } else {
8         overTempDetector = OFF;
9     }
10
11    if( gasDetector) {
12        gasDetectorState = ON;
13        alarmState = ON;
14    }
15    if( overTempDetector ) {
16        overTempDetectorState = ON;
17        alarmState = ON;
18    }
19    if( alarmState ) {
20        accumulatedTimeAlarm = accumulatedTimeAlarm + TIME_INCREMENT_MS;
21
22        if( gasDetectorState && overTempDetectorState ) {
23            if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM ) {
24                accumulatedTimeAlarm = 0;
25                alarmLed = !alarmLed;
26            }
27        } else if( gasDetectorState ) {
28            if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_ALARM ) {
29                accumulatedTimeAlarm = 0;
30                alarmLed = !alarmLed;
31            }
32        } else if ( overTempDetectorState ) {
33            if( accumulatedTimeAlarm >= BLINKING_TIME_OVER_TEMP_ALARM ) {
34                accumulatedTimeAlarm = 0;
35                alarmLed = !alarmLed;
36            }
37        }
38    } else{
39        alarmLed = OFF;
40        gasDetectorState = OFF;
41        overTempDetectorState = OFF;
42    }
43 }

```

Code 3.4 New implementation of the function `alarmActivationUpdate()`.

Table 3.10 shows the lines that were added to `availableCommands()` to inform how to get the potentiometer reading.

Table 3.10 Lines added to the function `availableCommands()`.

Function	Lines that were added
<code>void availableCommands()</code>	<code>uartUsb.write("Press 'P' or 'p' to get potentiometer reading\r\n\r\n", 49);</code>

Code 3.5 shows the lines that were added to *uartTask()* to inform the user of the reading of the potentiometer. In the event of “p” or “P” being pressed, the potentiometer reading is checked (line 3) and a message is sent to the PC by means of *uartUsb.write* (line 6). In order to prepare the message, a *string* is conformed in line 4, as discussed below.

```

1 case 'p':
2 case 'P':
3     potentiometerReading = potentiometer.read();
4     sprintf ( str, "Potentiometer: %.2f\r\n", potentiometerReading );
5     stringLength = strlen(str);
6     uartUsb.write( str, stringLength );
7     break;

```

Code 3.5 Lines that were added into the function *uartTask()*.

A *string* is an array of *char* terminated with the null character ('\0') which is usually used to process and store words or sentences in C/C++. As was explained in Chapter 2, in C/C++ the character '\0' represents the zero-element character (i.e., a '\0' written to *uartUsb* is not printed on the serial terminal), which is different from the character '0' which is used to indicate the number zero. In Chapter 6, the ASCII standard that is used in most computers and microcontrollers to encode characters will be introduced, and it will be seen that the character '0' is encoded with the number 48, while the null character '\0' is encoded with the number 0.

The null character is used in C/C++ to indicate the ending of strings because it is usual to deal with messages whose length may vary depending on certain circumstances. For instance, in Example 3.4 it will be seen that the number of characters in a message that will be used to indicate the temperature will vary depending on the temperature value. For example, if the temperature reading is equal to or above ten degrees, then one more character will be included in the message that will be used to report the reading (i.e., “Temperature: 12.2 °C” has one more character than “Temperature: 9.8 °C”). Therefore, the null character is appended at the end of the message to indicate where the message ends (i.e., “Temperature: 12.2 °C\0” and “Temperature: 9.8 °C\0”). In this way, a function can be used in order to print the message characters that are stored in an array until the null character is found.

This idea about *strings* is used in lines 4 and 5 of Code 3.5. The array is declared in the function *uartTask()*, as can be seen in Table 3.11. It is declared having 100 positions in order to stress that only the characters delimited by the null character will be printed in the message sent to the PC. There is also declared an *int* type variable named *stringLength* that will be used to store the length of the string (which might be different to the length of the array, as was discussed above).

Table 3.11 New variables that are declared in the function *uartTask()*.

Function	Lines that were added
void <i>uartTask()</i>	char str[100]; int stringLength;



NOTE: In order to save memory, it is good practice to use arrays of chars that are as small as possible to store the strings. In this example, 100 positions were used just to stress that not all the array positions are printed.

On line 4 of Code 3.5, the function `sprintf` provided by Mbed OS is used to write formatted data into the string. The string is composed of "Potentiometer:", followed by the value of `potentiometerReading` displayed with a precision of two decimal digits (%.*2f* format stands for a float variable with two decimal digits; for more information on how to set the format, please refer to [6]). On line 4 it can be seen that "\r\n" is also appended to the content of the string. After the content is complete (as indicated by the last " symbol on line 4), a terminating null character is automatically appended to the string by the function `sprintf`.



NOTE: The size of the buffer where the string is composed (in this case the array of char named `string`) should be large enough to contain the entire resulting string. Otherwise, problems may occur, as discussed later in this book.

On line 5 of Code 3.5, the function `strlen()` provided by Mbed OS is used to get the length of the string, which is stored in the variable `stringLength` that was introduced in Table 3.11. On line 6, `uartUsb.write()` is used to send the string having a length of `stringLength` to the PC.



NOTE: In order to enable the %.*2f* format that is used in Code 3.5, the file `mbed_app.json` was added to the "Example 3.3" project. For more information, please refer to [7].

NOTE: The C language provides four basic types: `char`, `int`, `float`, and `double`, and the modifiers `signed`, `unsigned`, `short`, and `long`. Table 3.12 lists the most common data types. The C99 standard (ISO/IEC 9899:1999) added the Boolean type. C99 includes definitions of new integer types to enhance the portability of programs, as the size of previous integer types may vary across different systems. The new types are especially useful in embedded systems, and the most common ones are listed in Table 3.13.

Table 3.12 C language basic arithmetic type specifiers.

Type	Details	Size (bits)	Format specifier
<code>char</code>	Smallest addressable integer type. Can contain the basic character set.	8	%c
<code>signed char</code>	Of the same size as <code>char</code> , but capable of containing the range [-127 to +127].	8	%c
<code>unsigned char</code>	Of the same size as <code>char</code> , but capable of containing the range [0 to 255].	8	%c
<code>int</code>	Basic signed integer type. Capable of containing the range [-32,767 to +32,767].	16	%i or %d
<code>unsigned int</code>	Basic unsigned integer type. Capable of containing the range [0 to 65,535].	16	%u

Type	Details	Size (bits)	Format specifier
long	Long signed integer type. Capable of containing the range [-2,147,483,647 to +2,147,483,647].	32	%li or %ld
float	Real floating-point type. Actual properties unspecified, but typically contains the range [$\pm 1.2\text{E-}38$ to $\pm 3.4\text{E+}38$].	32	%f
double	Real floating-point type. Actual properties unspecified, but typically contains the range [$\pm 2.3\text{E-}308$ to $\pm 1.7\text{E+}308$].	64	%lf



NOTE: The sizes of int, unsigned int, and long types are not specified by C99 and are platform-dependent. For more information on floating-point types, see the IEEE Standard for Floating-Point Arithmetic (IEEE 754) established by the Institute of Electrical and Electronics Engineers (IEEE).

Table 3.13 C99 standard definitions of new integer types.

Type	Details	Size (bits)
uint8_t	Capable of containing the range [0 to 255].	8
uint16_t	Capable of containing the range [0 to 65,535].	16
uint32_t	Capable of containing the range [0 to 4,294,967,295].	32
int8_t	Capable of containing the range [-128 to +127].	8
int16_t	Capable of containing the range [-32,768 to +32,767].	16
int32_t	Capable of containing the range [-2,147,483,648 to +2,147,483,647].	32

Proposed Exercises

- How can the code be modified in order to implement the over temperature alarm activation when the knob of the potentiometer is at 30% of its rotational travel?
- Are A0 to A5 the only analog inputs that can be used in the NUCLEO board?

Answers to the Exercises

- The line "#define POTENTIOMETER_OVER_TEMP_LEVEL 0.5" should be modified to: "#define POTENTIOMETER_OVER_TEMP_LEVEL 0.3".
- In Figure 1.23, it can be seen that A0 corresponds to PA_3 and ADC1/3 (i.e., Channel 3 of ADC 1). It implies that "AnalogIn potentiometer(A0);;" is the same as "AnalogIn potentiometer(PA_3);". In the same way, any of the ADCs that are shown in Figure 1.23 can be used. For example, if it is declared as "AnalogIn analogInput(PF_4)", then pin 7 of CN10 will be used as an analog input.

Example 3.4: Usage of Functions to Compute the Temperature Value

Objective

Introduce parameter passing in C/C++ functions.

Summary of the Expected Behavior

When “c” is pressed on the PC keyboard, formula (1), introduced in section 3.2.2, is applied to the reading of the potentiometer:

$$\text{Temperature } [{}^{\circ}\text{C}] = \frac{\text{Analog Reading} \times 3.3 \text{ V}}{0.01 \text{ V}/{}^{\circ}\text{C}} \quad (1)$$

The result of this formula is shown on the serial terminal with a legend indicating “Temperature: xx.xx °C”.

When “f” is pressed on the PC keyboard, the result of (1) is processed using formula (2), introduced in section 3.2.2:

$$\text{Temperature } [{}^{\circ}\text{F}] = \frac{\text{Temperature } [{}^{\circ}\text{C}] \times 9}{5} + 32 \text{ } {}^{\circ}\text{C} \quad (2)$$

The result of this formula is shown on the serial terminal with a legend indicating “Temperature: xx.xx °F”.



NOTE: In this example, the LM35 temperature sensor is not read, but its reading is simulated by means of the potentiometer reading. The LM35 temperature sensor will be read in Example 3.5, and the formulae introduced in this example will be applied to its reading.

Test the Proposed Solution on the Board

Import the project “Example 3.4” using the URL available in [5], build the project, and drag the .bin file onto the NUCLEO board. Open the serial terminal. Press “p” on the PC keyboard and read the message that appears on the serial terminal. Rotate the knob in both directions and see how the values displayed on the serial terminal change in the range of 0 to 1. Press “c” on the PC keyboard and read the message that appears on the serial terminal. Rotate the knob in both directions and see how the values displayed on the serial terminal change. Press “f” on the PC keyboard and read the message that appears on the serial terminal. Rotate the knob in both directions and see how the values displayed on the serial terminal change.

Discussion of the Proposed Solution

The proposed solution is based on two functions, one to implement formula (1), *analogReadingScaledWithTheLM35Formula()*, and another to implement formula (2), *celsiusToFahrenheit()*. Each of these functions receives one value, known in C/C++ as the function parameter. These values are, respectively, the value of “Analog Reading” in formula (1) and the value of “Temperature [°C]” in formula (2). After making the corresponding calculation, the function *analogReadingScaledWithTheLM35Formula()* returns the value indicated as “Temperature [°C]” in formula (1), while the function *celsiusToFahrenheit()* implements the calculation shown in formula (2) and returns the value of “Temperature [°F]”.



NOTE: In this example, the return value of `analogReadingScaledWithTheLM35Formula()` can be in the range of 0 to 330, given that the reading of the potentiometer is in the range of 0.0 to 1.0. Consequently, in this example the return value of `celsiusToFahrenheit()` can be in the range of 32 to 626. In Example 3.5, the return value of these two functions will correspond to the measurement range of the LM35 sensor.

Implementation of the Proposed Solution

The lines shown in Table 3.14 were modified and added to the code used in Example 3.3. The value of `POTENTIOMETER_OVER_TEMP_LEVEL` has been changed to 50, because the range of the value is no longer between 0.0 and 1.0, as discussed above. In addition, a new variable named `potentiometerReadingScaled` of type float has been declared, and two new functions have been declared: `analogReadingScaledWithTheLM35Formula()` and `celsiusToFahrenheit()`. In Code 3.6, the implementation of these two functions is shown. The first (lines 1 to 4) corresponds to formula (1) and the second (lines 6 to 9) corresponds to formula 2. Some lines were added as shown in Table 3.15.

Table 3.14 Sections in which lines were modified and added to Example 3.3.

Section	Lines that were modified
Definitions	#define POTENTIOMETER_OVER_TEMP_LEVEL 50
Section	Lines that were added
Declaration and initialization of public global variables	float potentiometerReadingScaled;
Declarations (prototypes) of public functions	float analogReadingScaledWithTheLM35Formula(float analogReading) float celsiusToFahrenheit(float tempInCelsiusDegrees)

Table 3.15 Functions in which lines were added to Example 3.3.

Function	Lines that were modified and added
void availableCommands()	uartUsb.write("Press 'f' or 'F' to get potentiometer reading in Fahrenheit\r\n", 61); uartUsb.write("Press 'c' or 'C' to get potentiometer reading in Celsius\r\n\r\n", 60);

```

1 float analogReadingScaledWithTheLM35Formula( float analogReading )
2 {
3     return ( analogReading * 3.3 / 0.01 );
4 }
5
6 float celsiusToFahrenheit( float tempInCelsiusDegrees )
7 {
8     return ( tempInCelsiusDegrees * 9.0 / 5.0 + 32.0 );
9 }
```

Code 3.6 Details of the new functions added to the code introduced in Example 3.3.

In Code 3.7, the modifications introduced in the function `alarmActivationUpdate()` are shown. In lines 3 and 4 it can be seen that the reading of the potentiometer is processed by the function

analogReadingScaledWithTheLM35Formula() and then stored in *potentiometerReadingScaled*. This variable is used in the *if* statement on line 6. The remaining lines of Code 3.7 are the previous existing implementation of *alarmActivationUpdate()*.

The lines added to *uartTask()* to get the temperature in Celsius and Fahrenheit are shown in Code 3.8. On line 4, it can be seen that the function *analogReadingScaledWithTheLM35Formula()* is used to obtain the temperature in °C. \xB0 is used to print the ° symbol in the string. Line 13 of Code 3.8 shows that the function *celsiusToFahrenheit()* is used to convert the result obtained on line 14 into °F.

```

1 void alarmActivationUpdate()
2 {
3     potentiometerReadingScaled =
4         analogReadingScaledWithTheLM35Formula (potentiometer.read() );
5
6     if ( potentiometerReadingScaled > POTENTIOMETER_OVER_TEMP_LEVEL ) {
7         overTempDetector = ON;
8     } else {
9         overTempDetector = OFF;
10    }
11
12    if( gasDetector ) {
13        gasDetectorState = ON;
14        alarmState = ON;
15    }
16    if( overTempDetector ) {
17        overTempDetectorState = ON;
18        alarmState = ON;
19    }
20    if( alarmState ) {
21        accumulatedTimeAlarm = accumulatedTimeAlarm + TIME_INCREMENT_MS;
22
23        if( gasDetectorState && overTempDetectorState ) {
24            if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM ) {
25                accumulatedTimeAlarm = 0;
26                alarmLed = !alarmLed;
27            }
28        } else if( gasDetectorState ) {
29            if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_ALARM ) {
30                accumulatedTimeAlarm = 0;
31                alarmLed = !alarmLed;
32            }
33        } else if ( overTempDetectorState ) {
34            if( accumulatedTimeAlarm >= BLINKING_TIME_OVER_TEMP_ALARM ) {
35                accumulatedTimeAlarm = 0;
36                alarmLed = !alarmLed;
37            }
38        }
39    } else{
40        alarmLed = OFF;
41        gasDetectorState = OFF;
42        overTempDetectorState = OFF;
43    }
44 }
```

Code 3.7 Modifications introduced to the function *alarmActivationUpdate()*.

```

1 case 'c':
2 case 'C':
3     sprintf ( str, "Temperature: %.2f \xB0 C\r\n",
4             analogReadingScaledWithTheLM35Formula (
5                 potentiometer.read() ) );
6     stringLength = strlen(str);
7     uartUsb.write( str, stringLength );
8     break;
9
10 case 'f':
11 case 'F':
12     sprintf ( str, "Temperature: %.2f \xB0 F\r\n",
13             celsiusToFahrenheit(
14                 analogReadingScaledWithTheLM35Formula (
15                     potentiometer.read() ) );
16     stringLength = strlen(str);
17     uartUsb.write( str, stringLength );
18     break;

```

Code 3.8 Lines that were added to the function `uartTask()`.

Proposed Exercise

- How can a C/C++ function be implemented to compute a temperature expressed in degrees Celsius from a temperature expressed in degrees Fahrenheit?

Answer to the Exercise

- A possible implementation is shown in Code 3.9.

```

1 float fahrenheitToCelsius( float tempInFahrenheitDegrees )
2 {
3     return ( (tempInFahrenheitDegrees - 32.0) * 5.0 / 9.0 );
4 }

```

Code 3.9 Implementation of `fahrenheitToCelsius()`.

Example 3.5: Measure Temperature and Detect Gas using the Sensors

Objective

Review the measurement of analog signals and introduce mathematical operations with arrays.

Summary of the Expected Behavior

The temperature measured by the LM35 temperature sensor should be displayed on the serial terminal in degrees Celsius when the “c” key is pressed on the computer keyboard, or in degrees Fahrenheit when the “f” key is pressed. In addition, the alarm should be activated when the temperature measured by the LM35 is above 50 °C (122 °F). When the “g” key is pressed, the serial terminal should indicate if gas is being detected or not by the MQ-2 gas sensor module. The potentiometer is not used in this example.



NOTE: Hereinafter, the buttons connected to D2 and D3 are not used to simulate gas detection and over temperature detection. Gas and temperature are sensed using the corresponding sensors. The button connected to D2 is used as the Alarm test button, in order to test if the alarm system is working without the need for gas or over temperature. When this button is pressed, the siren (implemented by a buzzer) sounds, and the Alarm LED turns on. The button connected to D3 has no functionality in this example.

The buzzer should sound continuously if any of the alarm conditions described in the previous examples occur. The Alarm LED should blink at the same rates described in previous examples. When the Alarm test button is pressed, the condition of simultaneous gas and over temperature detection is simulated.

Test the Proposed Solution on the Board

Import the project “Example 3.5” using the URL available in [5], build the project, and drag the `.bin` file onto the NUCLEO board. Open the serial terminal. Press “c” on the PC keyboard and read the message that appears on the serial terminal, indicating in degrees Celsius the temperature measured by the LM35. Press “f” and read the message indicating the measured temperature in degrees Fahrenheit. By means of a hairdryer or any similar method, increase the temperature of the LM35. When the temperature exceeds 50 °C, the Alarm LED should turn on with a blink period of 1000 ms, and the buzzer should sound.

Once the temperature of the LM35 sensor has reduced, use a lighter as described in section 3.2.2 to trigger the gas sensor. The Alarm LED should turn on with a blink of 500 ms and the buzzer should sound.

Finally, press the Alarm test button (the button connected to D2). The Alarm LED should turn on with a blink period of 100 ms (representing gas and over temperature detection) and the buzzer should sound.

Discussion of the Proposed Solution

The proposed solution is based on the reading of the analog signal that is provided by the LM35 temperature sensor. This signal increases by 10 mV/°C and is connected to the analog input 1 (A1) of the NUCLEO board. In subsection 3.2.2, it was shown that the reading of this signal is quite noisy. In order to overcome this problem, consecutive readings are averaged. If this average indicates that the temperature is below 50 °C, `overTempDetector` is set to OFF, and if it indicates that it is above 50 °C, `overTempDetector` is set to ON.

The proposed solution is also based on the DOUT signal of the MQ-2 gas sensor module, which was introduced in subsection 3.2.2. It also makes use of the buzzer, which is activated by means of setting the PE_10 pin of the NUCLEO board to GND. In this example, the button connected to D2 is used as the Alarm test button, as discussed above.

Implementation of the Proposed Solution

The objects and variables that were added to the program of Example 3.4 in order to implement the new functionality are shown in Table 3.16. NUMBER_OF_AVG_SAMPLES has been defined as 10 and OVER_TEMP_LEVEL as 50. The definition of POTENTIOMETER_OVER_TEMP_LEVEL was deleted because it is not used anymore. A DigitalIn object named *alarmTestButton* has been declared and assigned to D2, while a DigitalIn object named *mq2* has been declared and assigned to PE_12.

A DigitalInOut object named *sirenPin* has been declared and assigned to PE_10. This object is used to control the buzzer, as is explained below. An analog input object called *lm35* has also been declared and assigned to the analog input 1 (A1) of the NUCLEO board.

Finally, five public global variables are declared. The variable *lm35SampleIndex* will be used for the index of the array *lm35ReadingsArray*. The variable *lm35ReadingsArray* is an array of floats where 10 (i.e., NUMBER_OF_AVG_SAMPLES) consecutive readings will be stored. The variable *lm35ReadingsSum* will be used to store the sum of the ten positions of *lm35ReadingsArray*. *lm35ReadingsAverage* will be used to store the average of all the positions of *lm35ReadingsArray*, and finally, *lm35TempC* will be used to store the value of *lm35ReadingsAverage*, expressed in degrees Celsius. The variable *potentiometerReadingScaled* was deleted because it is not used anymore.

Table 3.16 Sections in which lines were added or modified in Example 3.4.

Section	Lines that were added	
Definitions	#define NUMBER_OF_AVG_SAMPLES #define OVER_TEMP_LEVEL	10 50
Declaration and initialization of public global objects	DigitalIn alarmTestButton(D2); DigitalIn mq2(PE_12); DigitalInOut sirenPin(PE_10); AnalogIn lm35(A1);	
Declaration and initialization of public global variables	int lm35SampleIndex = 0; float lm35ReadingsArray [NUMBER_OF_AVG_SAMPLES]; float lm35ReadingsSum = 0.0; float lm35ReadingsAverage = 0.0; float lm35TempC = 0.0;	

Code 3.10 shows the new implementation of *inputsInit()*. On line 3, the *alarmTestButton* is configured with a pull-down resistor. Lines 4 to 7 remain just as in the previous examples. On line 8, *sirenPin* is configured as “OpenDrain”, and on line 9 it is configured as an input. In this way, *sirenPin* is not energized, which turns off the buzzer.

```

1 void inputsInit()
2 {
3     alarmTestButton.mode(PullDown);
4     aButton.mode(PullDown);
5     bButton.mode(PullDown);
6     cButton.mode(PullDown);
7     dButton.mode(PullDown);
8     sirenPin.mode(OpenDrain);
9     sirenPin.input();
10 }
```

Code 3.10 New implementation of *inputsInit()*.

In Code 3.11, the new implementation of *alarmActivationUpdate()* is shown, with the following modified lines:

- The lines regarding the potentiometer were all removed (lines 3 to 10 of Code 3.7).
- Lines 6 to 17: the calculation of the temperature is implemented. The analog input is read and stored in the current position of the *lm35ReadingsArray* on line 6. The index is incremented (line 7), and it is set to 0 if it is beyond the last position of the array (lines 8 and 9). Then, all the array positions are summed (lines 12 to 15), the average value is computed (line 16), and the value of *lm35TempC* is obtained (line 17).
- Line 25: the digital input *mq2* is used to assess gas detection (it is active in low state).
- Lines 33 to 37: the Alarm test button functionality is implemented (if it is pressed, then *overTempDetectorState*, *gasDetectorState*, and *alarmState* are set to ON).
- Lines 40 and 41: *sirenPin* is configured as output, and its value is set to LOW to activate the buzzer.
- Line 63: *sirenPin* is configured as input, which turns off the buzzer.

```

1 void alarmActivationUpdate()
2 {
3     static int lm35SampleIndex = 0;
4     int i = 0;
5
6     lm35ReadingsArray[lm35SampleIndex] = lm35.read();
7     lm35SampleIndex++;
8     if ( lm35SampleIndex >= NUMBER_OF_AVG_SAMPLES ) {
9         lm35SampleIndex = 0;
10    }
11
12    lm35ReadingsSum = 0.0;
13    for ( i = 0; i < NUMBER_OF_AVG_SAMPLES; i++ ) {
14        lm35ReadingsSum = lm35ReadingsSum + lm35ReadingsArray[i];
15    }
16    lm35ReadingsAverage = lm35ReadingsSum / NUMBER_OF_AVG_SAMPLES;
17    lm35TempC = analogReadingScaledWithTheLM35Formula ( lm35ReadingsAverage );
18 }
```

```

19     if ( lm35TempC > OVER_TEMP_LEVEL ) {
20         overTempDetector = ON;
21     } else {
22         overTempDetector = OFF;
23     }
24
25     if( !mq2 ) {
26         gasDetectorState = ON;
27         alarmState = ON;
28     }
29     if( overTempDetector ) {
30         overTempDetectorState = ON;
31         alarmState = ON;
32     }
33     if( alarmTestButton ) {
34         overTempDetectorState = ON;
35         gasDetectorState = ON;
36         alarmState = ON;
37     }
38     if( alarmState ) {
39         accumulatedTimeAlarm = accumulatedTimeAlarm + TIME_INCREMENT_MS;
40         sirenPin.output();
41         sirenPin = LOW;
42
43         if( gasDetectorState && overTempDetectorState ) {
44             if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_AND_OVER_TEMP_ALARM ) {
45                 accumulatedTimeAlarm = 0;
46                 alarmLed = !alarmLed;
47             }
48         } else if( gasDetectorState ) {
49             if( accumulatedTimeAlarm >= BLINKING_TIME_GAS_ALARM ) {
50                 accumulatedTimeAlarm = 0;
51                 alarmLed = !alarmLed;
52             }
53         } else if ( overTempDetectorState ) {
54             if( accumulatedTimeAlarm >= BLINKING_TIME_OVER_TEMP_ALARM ) {
55                 accumulatedTimeAlarm = 0;
56                 alarmLed = !alarmLed;
57             }
58         }
59     } else{
60         alarmLed = OFF;
61         gasDetectorState = OFF;
62         overTempDetectorState = OFF;
63         sirenPin.input();
64     }
65 }
```

Code 3.11 New implementation of `alarmActivationUpdate()`.

NOTE: After a reset, the NUMBER_OF_AVG_SAMPLES positions of `lm35ReadingsArray` are not initialized. Therefore, until all of the positions of `m35ReadingsArray` are written at least once, the implementation shown in Code 3.11 may lead to wrong values in `lm35TempC`. This issue lasts for only one second and is addressed in the Proposed Exercise of this example.

The lines that were modified in `uartTask()` are shown in Code 3.12. It can be seen that in case '2', the digital input connected to the DOUT pin of the MQ-2 gas sensor is checked. If it has a low state, then it implies that gas is being detected. In case 'c' or 'C', the temperature expressed in degrees Celsius is shown on the serial terminal, while in case 'f' or 'F', the temperature expressed in degrees Fahrenheit is shown.

```

1 case '2':
2     if ( !mq2 ) {
3         uartUsb.write( "Gas is being detected\r\n", 22 );
4     } else {
5         uartUsb.write( "Gas is not being detected\r\n", 27 );
6     }
7     break;
8
9 case 'c':
10 case 'C':
11     sprintf ( str, "Temperature: %.2f \xB0 C\r\n", lm35TempC );
12     stringLength = strlen(str);
13     uartUsb.write( str, stringLength );
14     break;
15
16 case 'f':
17 case 'F':
18     sprintf ( str, "Temperature: %.2f \xB0 F\r\n",
19             celsiusToFahrenheit( lm35TempC ) );
20     stringLength = strlen(str);
21     uartUsb.write( str, stringLength );
22     break;

```

Code 3.12 Lines that were modified in the function `uartTask()`.

Proposed Exercise

1. It has been mentioned that until all the positions of `lm35ReadingsArray` are written (i.e., during the first second), the value of `lm35TempC` is not correct because many positions of this array were not initialized. How can this problem be solved?

Answer to the Exercise

1. The function shown in Code 3.13 can be executed once after power on in order to initialize all the positions of `lm35ReadingsArray` to zero. In this way, at the beginning the value of `lm35TempC` will not be correct but will not trigger the over temperature alarm anyway, because the resulting average temperature will be very low (due to the fact that many zero values will be used in the average calculation). After one second, all the positions of `lm35ReadingsArray` will have correct values and, therefore, the value of `lm35TempC` will be correct. Be aware that if an under temperature condition is being checked, then this solution must be adapted.

```

1 void lm35ReadingsArrayInit()
2 {
3     int i;
4     for( i=0; i<NUMBER_OF_AVG_SAMPLES ; i++ ) {
5         lm35ReadingsArray[i] = 0;
6     }
7 }

```

Code 3.13 Implementation of the proposed function `lm35ReadingsArrayInit()`.

3.3 Under the Hood

3.3.1 Basic Principles of Analog to Digital Conversion

In this chapter, the analog signal provided by the LM35 was digitized by the NUCLEO board by means of an analog to digital converter (ADC) included in the STM32 microcontroller. The aim of this subsection is to explain how the analog to digital converter works.

The STM32 microcontroller of the NUCLEO board includes a Successive Approximation Register (SAR) ADC. A simplified diagram of an SAR ADC is shown in Figure 3.10. The analog input of the ADC is indicated at the top of the figure, and the digital output value that is obtained as a result of the conversion process is indicated at the right side. The SAR ADC consists of three main elements: (1) an *Analog Comparator*, (2) a *Digital to Analog Converter (DAC)*, and (3) an *Iterative Conversion Controller (ICC)*. The analog signals and elements are indicated in light green, while the digital signals are indicated in dark green. A color gradient is used to indicate that a given element has inputs of one type and outputs of another type (i.e., the DAC has digital inputs and an analog output, while the analog converter has analog inputs and a digital output).

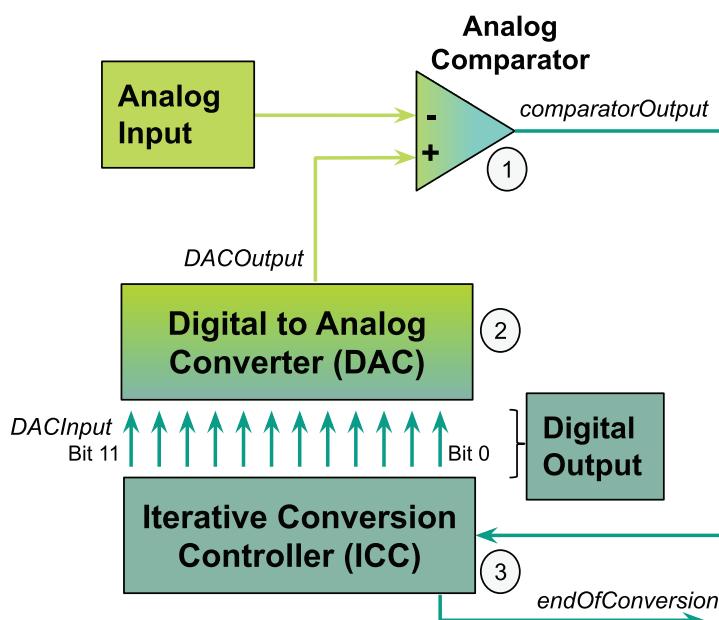


Figure 3.10 Simplified diagram of a Successive Approximation Register analog to digital converter.

As its name indicates, the SAR ADC is based on an iterative process. At the beginning of the process, the ICC sets Bit 11 at the *DACInput* to 1 (i.e., *DACInput* = 1000 0000 0000). In this way, a value equal to half of its full range is obtained at *DACOutput*. Then, the ICC analyzes the value of the *comparatorOutput*. If the *comparatorOutput* indicates that the voltage of the Analog Input is bigger than the *DACOutput*, the value at the *DACInput* is increased to 1100 0000 0000. In this way, a value equal to three quarters (75%) of the full range is obtained at the *DACOutput*. Conversely, if the value

of the *comparatorOutput* indicates that the Analog Input is smaller than the *DACOutput*, then the ICC decreases the value at the *DACInput* to 0100 0000 0000. As a consequence, a value equal to one quarter (25%) of the *DACOutput* full range is obtained.

This process continues until the values of the twelve bits of *DACInput* are determined. At that point, the *End of Conversion* is reached, the signal *endOfConversion* is set to the active state, and the *DACOutput* value is the best possible digitalization of the analog input voltage. Then, the twelve bits that the ICC has established at *DACInput* are shared by the SAR ADC as the Digital Output result of the conversion. In the particular case of the functionality provided for the *AnalogIn* objects by the *mbed.h* library used in this chapter, the result of the conversion is scaled in the range of 0 to 1 (i.e., it returns 1.0 if 1111 1111 1111 is obtained, and 0.0 if 0000 0000 0000 is obtained).



TIP: For more information on SAR ADCs, see Maxim's application note 1080, available from [8].

Proposed Exercise

1. How can a successive approximation register (SAR) ADC be implemented on the NUCLEO board?

Answer to the Exercise

1. The proposed solution is available from [5] with the name "Under the Hood Chapter 3." The reader is encouraged to load the proposed solution onto the NUCLEO board and follow the prompts on the serial terminal to see how the implemented SAR ADC works.

In the proposed solution, the libraries *mbed.h* and *arm_book_lib.h* are used, as shown in Code 3.14. There are also two #defines, one to indicate the number of bits, *NUMBER_OF_BITS*, defined as 12, and another one, *MAX_RESOLUTION*, that is used to normalize the output into the 0 to 1 range, defined as 4095 (i.e., $2^{12} - 1$).

```
1 //=====[Libraries]=====
2
3 #include "mbed.h"
4 #include "arm_book_lib.h"
5
6 //=====[Defines]=====
7
8 #define NUMBER_OF_BITS 12
9 #define MAX_RESOLUTION 4095.0
```

Code 3.14 Libraries and defines used in the implementation of the proposed solution.

The public global objects that are used are shown in Code 3.15. The B1 USER button of the NUCLEO board is declared as the *nextStepButton* (line 3) and is used to advance one step in the iterative conversion process. The analog input A0 (line 5) is used earlier to get a reading of the analog signal at terminal (2) of the potentiometer, in the same way as it was used in this chapter. The LD1 of the

NUCLEO board is used to indicate the start of a new conversion (line 7), LD2 is used to indicate a new step in the conversion process (line 8), and LD3 is used to announce the end of the conversion (line 9). Finally, on line 11, the UnbufferedSerial object *uartUsb* is declared and will be used to send messages to the PC to show the current state of the conversion process.

```

1 //=====[Declaration and initialization of public global objects]=====
2
3 DigitalIn nextStepButton(BUTTON1);
4
5 AnalogIn potentiometer(A0);
6
7 DigitalOut startOfConversionLed(LED1);
8 DigitalOut stepOfConversionLed(LED2);
9 DigitalOut endOfConversionLed(LED3);
10
11 UnbufferedSerial uartUsb(USBTX, USBRX, 115200);

```

Code 3.15 Public global objects used in the implementation of the proposed solution.

The public global variables that are used are shown in Code 3.16. On line 3, the variable *comparatorOutput* is declared and initialized to 0 and is used to implement the output of the Analog Comparator, as illustrated in Figure 3.10. An array called *DACInput* of type bool, of size *NUMBER_OF_BITS*, is declared on line 4. This is used to implement the input to the Digital to Analog Converter (see Figure 3.10). On line 5, an array called *digitalOutput* of type bool, of size *NUMBER_OF_BITS*, is declared. This is used to provide the digital output, as shown in Figure 3.10.

On line 7 of Code 3.16, an integer type variable called *conversionStep* is declared, which is used to keep track of the current conversion step. On line 9, the variable *analogInput* of type float is declared. This variable is used to get a reading of terminal 2 of the potentiometer. On line 10, a float type variable (*digitalOutputScaledInRange0to1*) is declared that is used once the conversion is finished to provide an output normalized in the range 0 to 1. Finally, on line 11, a float type variable *DACOutput* is defined that is used to implement the output of the Digital to Analog Converter (see Figure 3.10).

```

1 //=====[Declaration and initialization of public global variables]=====
2
3 bool comparatorOutput = 0;
4 bool DACInput[NUMBER_OF_BITS];
5 bool digitalOutput[NUMBER_OF_BITS];
6
7 int conversionStep = 0;
8
9 float analogInput;
10 float digitalOutputScaledInRange0to1 = 0;
11 float DACOutput = 0;

```

Code 3.16 Public global variables used in the implementation of the proposed solution.

In Code 3.17, the public functions used in the implementation of the proposed solution are shown. Line 3 declares the function *inputsInit()* that is used to initialize the inputs, and on line 4, the function *outputsInit()* that is used to initialize the outputs is declared. On line 6, the function *startOfConversion()* is declared and, on line 12, the function *endOfConversion()*. These two functions are used to turn on

and off LD1 and LD3, respectively, and to send messages to the user by means of *uartUsb*. On line 7, the function *analogComparator()*, which is used to implement the Analog Comparator, is declared (see Figure 3.10). On line 8, the function that is used to implement one step of the Iterative Conversion Controller, *iterativeConversionControllerStep()*, is declared. The function used to implement the Digital to Analog Converter (DAC) is declared on line 9. On line 10, a function is declared that is used to reset the Iterative Conversion Controller. The function that is used to show the conversion status is declared on line 11. This function will send messages to the PC by means of *uartUsb* and will also turn on and off the LD2 of the NUCLEO board. Finally, the function that will show the result, *endOfConversion()*, is declared on line 12.

```

1 //=====[Declarations (prototypes) of public functions]=====
2
3 void inputsInit();
4 void outputsInit();
5
6 void startOfConversion();
7 bool analogComparator();
8 bool iterativeConversionControllerStep();
9 float digitalToAnalogConverter();
10 void resetIterativeConversionController();
11 void showConversionStatus();
12 void endOfConversion();
```

Code 3.17 Public functions used in the implementation of the proposed solution.

In Code 3.18, the *main()* function is shown. On line 5, the function used to initialize the inputs is called (i.e., *inputsInit()*), and on line 6 the function used to initialize the outputs is called (*outputsInit()*). On line 8, the function *startOfConversion()* is used to send a message by means of *uartUsb* and to turn on LD1 for one second. On line 9, *resetIterativeConversionController()* is called in order to reset some specific variables.

The *for* loop between lines 10 and 22 is used to implement the SAR ADC following the diagram illustrated in Figure 3.10. From lines 10 to 12, it can be seen that the number of loops that are executed is equal to *NUMBER_OF_BITS* and that the current conversion step is tracked by *conversionStep*. On line 14, the variable *DACOutput* is assigned the return value of the function *digitalToAnalogConverter()*. On line 16, the variable *comparatorOutput* is assigned the return value of the function *analogComparator()*. The assignment of the Boolean state at the position “*NUMBER_OF_BITS - conversionStep*” of the array *DACInput* is made on line 18, using the return value of the function *iterativeConversionControllerStep()*. Line 21 is used to send the status of the conversion via *uartUsb* and LD2 by means of the function *showConversionStatus()*. Finally, on line 24, the function *endOfConversion()* is used to send the corresponding message by means of *uartUsb* and to turn on LD3 for one second.

```

1 //=====[Main function, the program entry point after power on or reset]=====
2
3 int main()
4 {
5     inputsInit();
6     outputsInit();
7     while (true) {
8         startOfConversion();
9         resetIterativeConversionController();
10        for (conversionStep = 1;
11            conversionStep <= NUMBER_OF_BITS;
12            conversionStep++) {
13
14            DACOutput = digitalToAnalogConverter( );
15
16            comparatorOutput = analogComparator( );
17
18            DACInput[NUMBER_OF_BITS - conversionStep] =
19                iterativeConversionControllerStep( );
20
21            showConversionStatus( );
22        }
23
24        endOfConversion( );
25    }
26 }
```

Code 3.18 Public functions used in the implementation of the proposed solution.

The implementation of the first two public functions is shown in Code 3.19. It is important to note that the function *inputsInit()* is empty and that the only reason this function is kept is to follow the organization of the programs established in previous chapters. On line 7, the implementation of *outputsInit()* is shown. It can be seen that the three LEDs are turned off.

```

1 //=====[Implementations of public functions]=====
2
3 void inputsInit()
4 {
5 }
6
7 void outputsInit()
8 {
9     startOfConversionLed = OFF;
10    stepOfConversionLed = OFF;
11    endOfConversionLed = OFF;
12 }
```

Code 3.19 Implementation of the functions *inputsInit()* and *outputsInit()*.

The implementation of the function *startOfConversion()* is shown in Code 3.20. In line 3, the message “Please press Next Step Button (B1 USER)” is sent over *uartUsb*. Line 4 is used to wait until the *nextStepButton* is pressed. In line 5, a message is sent to indicate that the conversion has started. On line 6, the reading of terminal 2 of the potentiometer is stored in the variable *analogInput*. Finally, lines 7 to 9 turn on *startOfConversionLed* for one second.

```

1 void startOfConversion()
2 {
3     uartUsb.write( "Please press Next Step Button (B1 USER)\r\n\r\n" , 43 );
4     while ( !nextStepButton ) {};
5     uartUsb.write( "Conversion started\r\n\r\n" , 22 );
6     analogInput = potentiometer.read();
7     startOfConversionLed = ON;
8     delay(1000);
9     startOfConversionLed = OFF;
10 }
```

Code 3.20 Implementation of the function startOfConversion().

In Code 3.21, the implementation of the function *resetIterativeConversionController()* is shown. On line 3, a local integer variable named *i* is declared. This variable is used from line 4 to 6 in order to store a 0 in each of the NUMBER_OF_BITS positions of *DACInput*. Finally, in line 7, a message indicating that *DACinput* has been reset is sent by *uartUsb*.

```

1 void resetIterativeConversionController( )
2 {
3     int i;
4     for ( i = 0; i < NUMBER_OF_BITS; i++ ) {
5         DACInput[i]=0;
6     }
7     uartUsb.write( "DACinput reseted\r\n\r\n" , 20 );
8 }
```

Code 3.21 Implementation of the function resetIterativeConversionController().

Code 3.22 shows the implementation of the function *iterativeConversionControllerStep()*. It can be seen in line 3 that if *comparatorOutput* is equal to 1, then 1 is returned (line 4), and if *comparatorOutput* is not equal to 1, then 0 is returned (line 6).

```

1 bool iterativeConversionControllerStep( )
2 {
3     if (comparatorOutput == 1) {
4         return 1;
5     } else {
6         return 0;
7     }
8 }
```

Code 3.22 Implementation of the function iterativeConversionControllerStep().

The implementation of the function *digitalToAnalogConverter()* is shown in Code 3.23. Three variables are declared in lines 3 to 5: a float called *output*, which is initialized to 0 and is used to compute the output value; an integer variable called *power*, which is initialized to 1 and is used to implement some mathematical operations; and an integer auxiliary variable called *i*. The *for* loop on line 7 is used to implement the operation between lines 8 and 14 NUMBER_OF_BITS times. The condition inside the

if statement on line 8 is true only when the variable *i* has the same value as “NUMBER_OF_BITS - conversionStep”. For example, in the first conversion step the variable *conversionStep* is equal to 1. Then, considering that NUMBER_OF_BITS is defined as 12, “NUMBER_OF_BITS - conversionStep” is equal to 11. So the condition “*i == NUMBER_OF_BITS - conversionStep*” is only valid for *i* being 11. In this way, considering that at the beginning all the positions of *DACInput* are equal to 0, after 11 executions of line 11 with *i* varying from 0 to 10, *output* will be equal to 0 and *power* will be equal to 2^{11} . Then, with *i* being 11, 2^{11} (2048) will be added to *output* and, therefore, the return value (line 16) will be 0.5001 (2048/4095).

```

1 float digitalToAnalogConverter( )
2 {
3     float output = 0;
4     int power = 1;
5     int i;
6
7     for (i=0; i<NUMBER_OF_BITS; i++) {
8         if (i == ( NUMBER_OF_BITS - conversionStep ) ) {
9             output += 1*power;
10        } else {
11            output += DACInput[i]*power;
12        }
13        power *= 2;
14    }
15
16    return output / MAX_RESOLUTION;
17 }
```

Code 3.23 Implementation of the function *digitalToAnalogConverter()*.

In Code 3.24, the implementation of the *analogComparator()* function is shown. It returns 1 if “*analogInput* >= *DACOutput*” and returns 0 otherwise.

```

1 bool analogComparator( )
2 {
3     if ( analogInput >= DACOutput ) {
4         return 1;
5     } else {
6         return 0;
7     }
8 }
```

Code 3.24 Implementation of the function *analogComparator()*.

The implementation of the function *showConversionStatus()* is shown in Code 3.25. Lines 5 to 10 are used to send the current value of *conversionStep*, *analogInput*, and *DACOutput* over *uartUsb*. On line 11, the message “DAC Input:” and then one after the other “1” and “0” are sent, depending on the content of each of the positions of the array *DACInput* (lines 13 to 19). Line 20 is used to send a message that contains only a vertical separation. A delay of one second is introduced in line 21, then in line 22 *stepOfConversion* is turned ON and waits until the *nextStepButton* is pressed (line 23) to turn it OFF (line 24).

```

1 void showConversionStatus( )
2 {
3     int i;
4     char str[30];
5     sprintf ( str, "Conversion step: %i\r\n", conversionStep );
6     uartUsb.write( str, strlen(str) );
7     sprintf ( str, "Analog Input: %.3f\r\n", analogInput );
8     uartUsb.write( str, strlen(str) );
9     sprintf ( str, "DAC Output: %.3f\r\n", DACOutput );
10    uartUsb.write( str, strlen(str) );
11    uartUsb.write( "DAC Input: " , 11);
12
13    for (i=1; i<=NUMBER_OF_BITS; i++) {
14        if (DACInput[NODE_OF_BITS-i] == 1) {
15            uartUsb.write( "1" , 1 );
16        } else {
17            uartUsb.write( "0" , 1 );
18        }
19    }
20    uartUsb.write( "\r\n\r\n\r\n" , 4 );
21    delay(1000);
22    stepOfConversionLed = ON;
23    while (!nextStepButton);
24    stepOfConversionLed = OFF;
25 }
```

Code 3.25 Implementation of the function showConversionStatus().

Finally, in Code 3.26, the implementation of the function `endOfConversion()` is shown. It is quite similar to the implementation of the function `showConversionStatus()`. The differences are that there is now an “End of conversion message” on line 4 and `endOfConversionLed` is turned on instead of `stepOfConversionLed`. Additionally, `DACInput` is not shown because it is an internal value that has no meaning once the conversion is finished.

```

1 void endOfConversion( )
2 {
3     char str[30];
4     sprintf ( str, "End of conversion\r\n\r\n" );
5     uartUsb.write( str, strlen(str) );
6     sprintf ( str, "Analog Input: %.3f\r\n", analogInput );
7     uartUsb.write( str, strlen(str) );
8     sprintf ( str, "DAC Output: %.3f\r\n", DACOutput );
9     uartUsb.write( str, strlen(str) );
10
11    endOfConversionLed = ON;
12    delay(1000);
13    endOfConversionLed = OFF;
14 }
```

Code 3.26 Implementation of the function endOfConversion().

3.4 Case Study

3.4.1 Vineyard Frost Prevention

In this chapter, a temperature sensor was connected to the NUCLEO board and the measured temperature was sent to a PC using serial communication. In this way, the alarm was activated if over temperature was detected, and using a serial terminal, it was possible to read the temperature on the PC. A vineyard frost prevention system, where low temperatures are detected, built with Mbed and containing some similar features, can be found in [9]. Figure 3.11 shows a representation of the system.



Figure 3.11 “Vineyard frost prevention” built with Mbed contains elements introduced in this chapter.

The localized sensors of the vineyard frost prevention system are designed to be mounted over the vines, and their measurements are transmitted via a wireless LoRa network. Therefore, specific vines requiring protection from frost can be detected, which improves the yield of grapes. Better performance is obtained in comparison with centralized sensing systems.

It can be appreciated that the functionality shown in Figure 3.11 is very similar to the functionality implemented in this chapter (i.e., temperature measurement plus results displayed on a PC). The following chapters will explain how to implement a wireless connection to the NUCLEO board and will also introduce the knowledge required to understand the use and utility of some other technologies mentioned in [9], such as LoRa networking.

Proposed Exercises

1. Does the vineyard frost prevention system measure any other variables besides temperature?
2. Is the core of the microcontroller used by the vineyard frost prevention system the same as the core of the STM32 microcontroller of the NUCLEO Board?

Answers to the Exercises

1. The vineyard frost prevention system is also provided with humidity sensors.
2. The vineyard frost prevention system is based on an Arm Cortex-M3 running at 32 MHz, while the NUCLEO board has an Arm Cortex-M4 that can run at 180 MHz. This implies that the NUCLEO board is provided with a more powerful processor that runs up to six times faster.

References

- [1] "LM35 data sheet, product information and support | TI.com". Accessed July 9, 2021.
<https://www.ti.com/product/LM35>
- [2] "MQ2 Gas Sensor Pinout, Features, Equivalents & Datasheet". Accessed July 9, 2021.
<https://components101.com/sensors/mq2-gas-sensor>
- [3] "List of integrated circuit packaging types - Wikipedia". Accessed July 9, 2021.
https://en.wikipedia.org/wiki/List_of_integrated_circuit_packaging_types
- [4] "What is MOSFET: Symbol, Working, Types & Different Packages". Accessed July 9, 2021.
<https://components101.com/articles/mosfet-symbol-working-operation-types-and-applications>
- [5] "GitHub - armBookCodeExamples/Directory". Accessed July 9, 2021.
<https://github.com/armBookCodeExamples/Directory/>
- [6] "printf - C++ Reference". Accessed July 9, 2021.
<https://www.cplusplus.com/reference/cstdio/printf/>
- [7] "mbed-os/README.md at master · ARMmbed/mbed-os · GitHub". Accessed July 9, 2021.
<https://github.com/ARMmbed/mbed-os/blob/master/platform/source/minimal-printf/README.md#usage>
- [8] "Understanding SAR ADCs: Their Architecture and Comparison with Other ADCs". Accessed July 9, 2021.
<https://pdfserv.maximintegrated.com/en/an/AN1080.pdf>
- [9] "Vineyard frost prevention | Mbed". Accessed July 9, 2021.
<https://os.mbed.com/built-with-mbed/vineyard-frost-prevention/>

Chapter 4

Finite-State Machines and
the Real-Time Clock

4.1 Roadmap

4.1.1 What You Will Learn

After you have studied the material in this chapter, you will be able to:

- Describe how to connect matrix keypads to the NUCLEO board.
- Summarize the fundamentals of programs based on finite-state machines (FSMs).
- Develop programs that implement FSMs with the NUCLEO board.
- Implement programs that make use of the real-time clock (RTC).
- Use pointers to manage character strings.

4.1.2 Review of Previous Chapters

In previous chapters, many features were added to the smart home system. Those features were controlled by a set of buttons, which in some cases had to be pressed in a particular order and in other cases had to be pressed at the same time (e.g., A + B + Enter or A + B + C + D). Pressing multiple buttons at the same time can be both difficult and impractical. It is, therefore, desirable to find a more convenient way to control the system.

An alarm was also included in the system and could be activated due to gas detection, over temperature detection, or the simultaneous occurrence of both. In the current implementation, the system has no way to record which alarms have been triggered or which events triggered those alarms.

4.1.3 Contents of This Chapter

As the complexity of the system increases, it becomes necessary to introduce more powerful techniques in order to sustain software maintainability and increase flexibility. In this chapter, the concept of a *finite-state machine* (FSM) is introduced and its support in organizing programs will be described.

FSMs are introduced by means of a new feature that is added to the system: the “double-press” functionality on the Enter button. Using this, the Enter button can be pressed twice consecutively to turn off the Incorrect code LED. This improvement will be accompanied by the use of a matrix keypad instead of buttons connected to the breadboard. This replacement will be completed gradually through the chapter, as new concepts must be developed to fully incorporate the usage of the matrix keypad.

Finally, the *real-time clock* (RTC) of the STM32 microcontroller will be used to incorporate a time stamp into the events that are detected by the smart home system. In this way, it will be possible, for example, to register or log the time and date of the alarm activations and access this information from the PC.



WARNING: The program code of the examples introduced in this chapter gets quite large. For example, the program code in Example 4.4 has almost 600 lines. The reader will see that it is not easy to follow a program with so many lines in a single file. This problem will be tackled in the next chapter, where modularization applied to embedded systems programming is introduced in order to reorganize the program code into smaller files. In this way, the learn-by-doing principle is followed.

4.2 Matrix Keypad Reading with the NUCLEO Board

4.2.1 Connect a Matrix Keypad and a Power Supply to the Smart Home System

In this chapter, more buttons will be added to the alarm control panel of the smart home system, as shown in Figure 4.1. The aim is to improve the functionality and to allow the use of numeric codes as is common in this type of system.

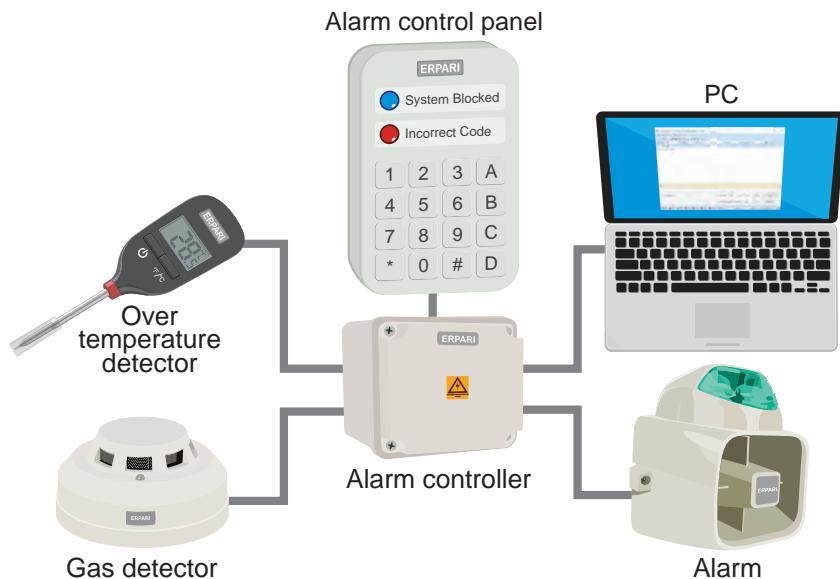


Figure 4.1 The smart home system is now connected to a matrix keypad.

It will also be explained through the examples how an FSM can be used to improve the way in which the numeric code is entered. By using an FSM it will be possible to enter the digits of the code one after the other. This contrasts with the method of pressing multiple buttons at once, as in Chapter 1.

To implement the new functionality, a matrix keypad such as the one described in [1] should be connected to the smart home system, as shown in Figure 4.2. An MB102 module, such as the one presented in [2], should also be connected. Matrix keypads are usually found in calculators, microwaves, door locks, and similar devices. They are available in different sizes, but 4×4 and 4×3 are the most common. The MB102 module is a breadboard power supply used to avoid overloading power supplied by the NUCLEO board.

In the following examples, new functions will be introduced into the program to gradually replace the buttons connected to D4–D7 by the matrix keypad.



NOTE: The buttons connected to D2 and D4–D7 are used in Examples 4.1 and 4.2. They can be removed from the setup in Example 4.3, as their functionality will be assigned to the B1 User button and the matrix keypad, respectively.

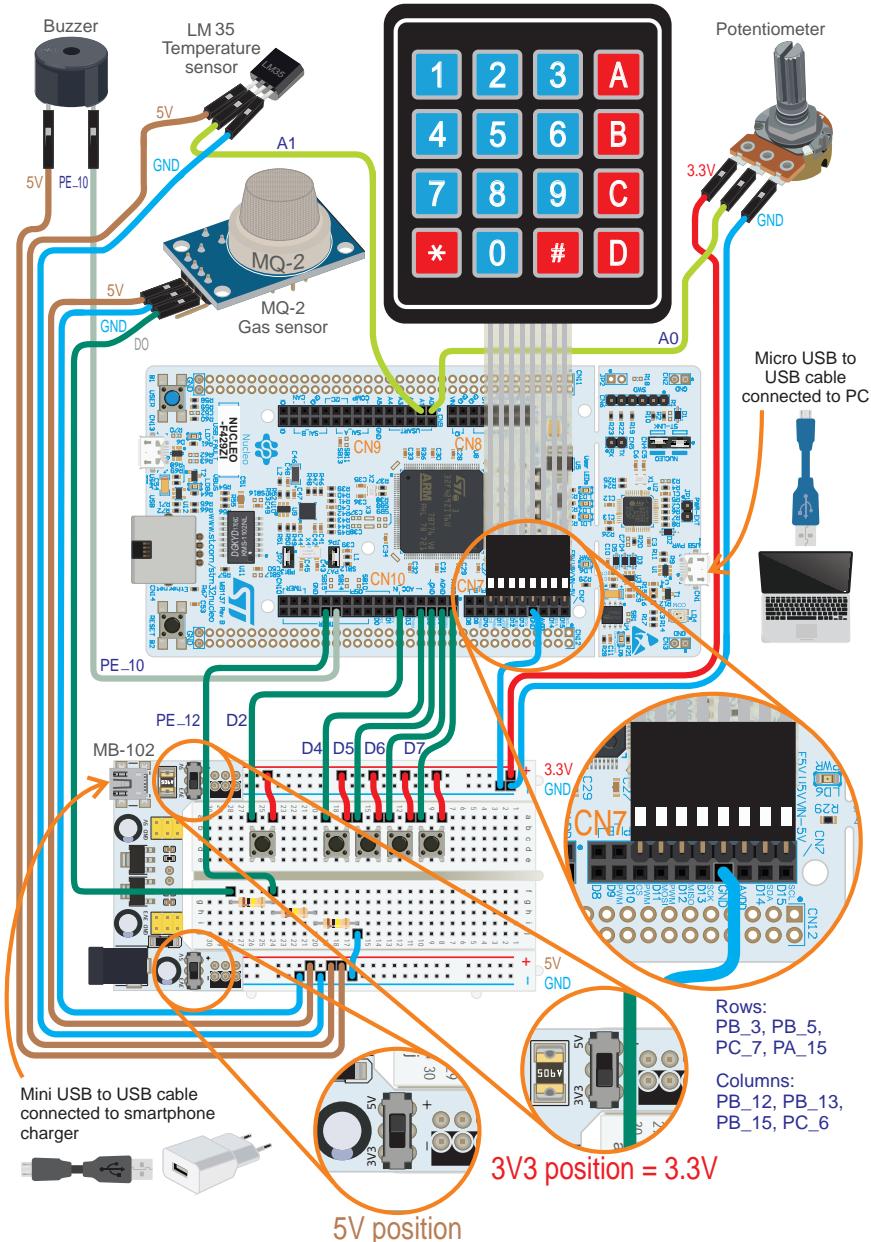


Figure 4.2 The smart home system is now connected to a matrix keypad.

Figure 4.3 shows how to use a 90-degree 2.54 mm (0.1") pitch pin header to prepare the connector of the matrix keypad for the proposed setup.

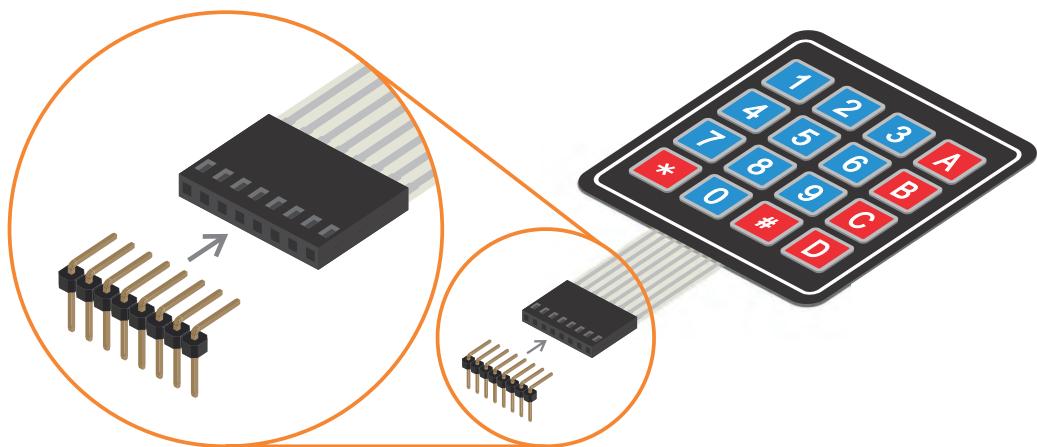


Figure 4.3 Detail showing how to prepare the matrix keypad connector using a pin header.

A diagram illustrating the connections of the matrix keypad is shown in Figure 4.4. It can be seen that four pins are used for the rows (R1–R4) and four pins are used for the columns (C1–C4).

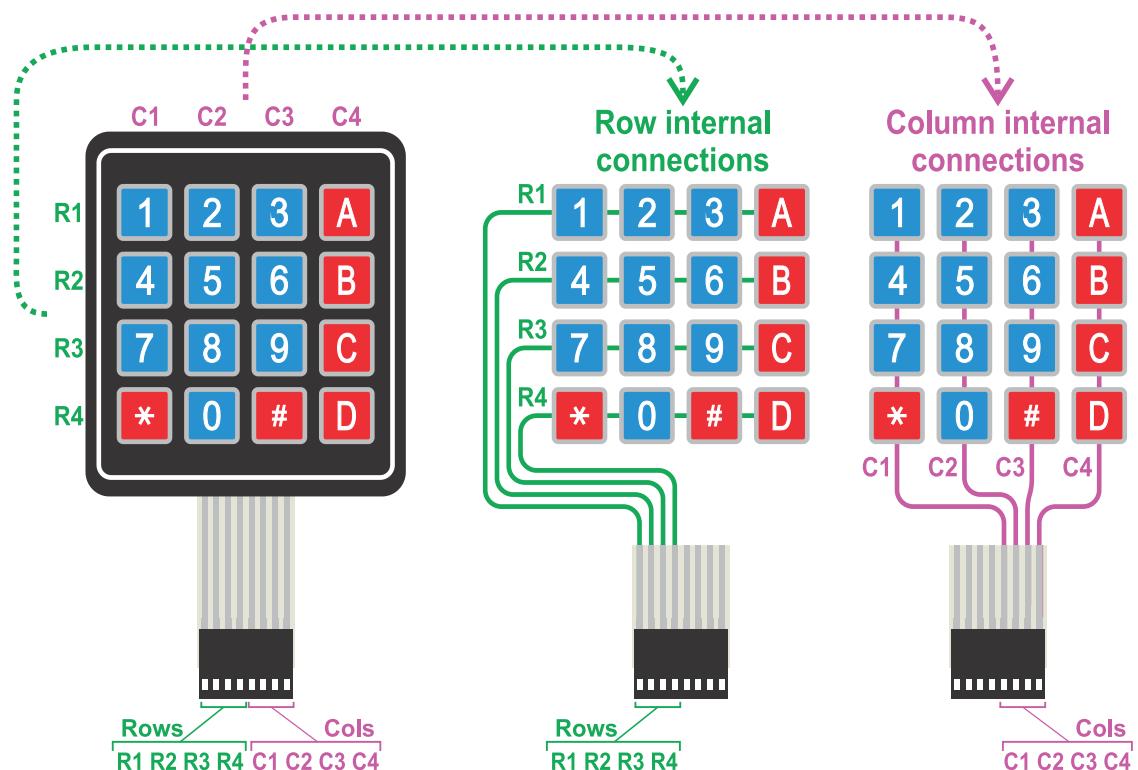


Figure 4.4 Diagram of the connections of the matrix keypad.

The internal connections of the matrix keypad are shown in Figure 4.5. For example, when key “1” is pressed, a connection is established between R1 and C1; when key “2” is pressed, R1 is connected to C2.

In Figure 4.5, the connections of the NUCLEO board and the matrix keypad are also shown. To scan if key “1” is pressed, a 0 V signal is connected to PB_3 (which is connected to R1), and the state of PB_12 (which is connected to C1) is read. If PB_12 is OFF, it means that key “1” is pressed; otherwise it is not pressed. This is because `.mode(PullUp)` is used to configure the digital inputs that are used (PB_12, PB_13, PB_15, and PC_6). The same procedure is used to determine if key “2” is pressed, but replacing PB_12 by PB_13, since PB_13 is connected to C2, as can be seen in Figure 4.5. To scan other keys, the corresponding rows and columns should be used.

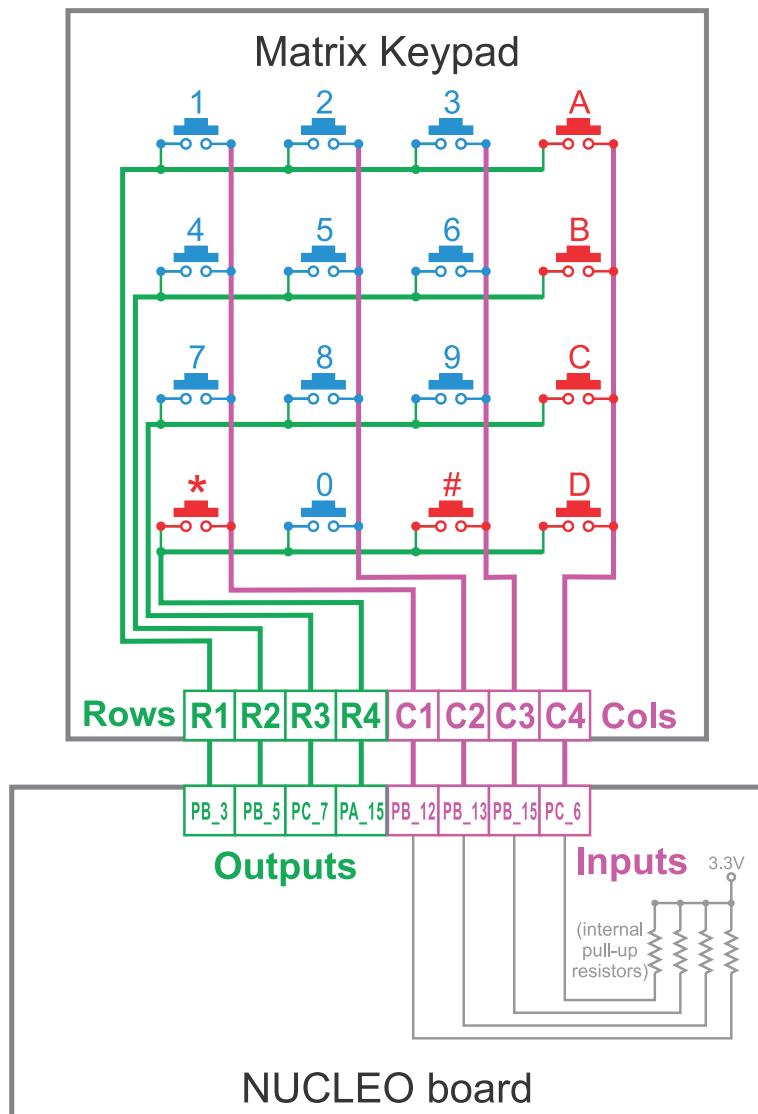


Figure 4.5 Diagram of the connections between the matrix keypad and the NUCLEO board.



NOTE: The use of a matrix for the connection of the keys of the keypad allows a reduction in the number of wires that are used to get the state of the keys. For example, instead of using 17 wires to read 16 keys (one for each key and one for GND), in Figure 4.5, 8 wires are used to read 16 keys.

The following subsection explains how to test if the matrix keypad is working properly. In addition, the examples explain how to tackle glitches and bounces in the signal, which are common in any types of keys or buttons, as shown in Figure 4.6. Glitches are unexpected variations in the signal due to electrical noise, while bounces are a consequence of the spring that is part of the key or button.

Typically, a glitch lasts for less than 1 millisecond, while a bounce can last up to 30 milliseconds. This chapter will explain how the signal can be processed, using an FSM, in order to distinguish between a key or button being pressed and a glitch or a bounce. By filtering glitches and bounces using software, it is possible to avoid, or at least reduce, the usage of electronic components for filtering purposes.



NOTE: The voltage levels shown in Figure 4.6 correspond to the connections in Figure 4.5. The techniques that are explained in this chapter can also be applied if the voltages are reversed in such a way that the signal is 0 V if the key or button is not pressed and 3.3 V if it is pressed, as in the previous chapters.

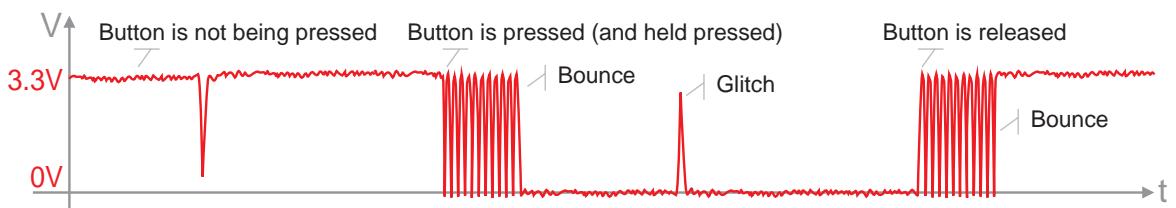


Figure 4.6 Voltage signal over time for a given button, including typical glitches and bounces.



NOTE: In previous chapters, glitches and bounces caused no problems as the system behavior was specially designed in order to avoid problems regarding unwanted consecutive readings of the buttons. For example, the Enter and A, B, C, and/or D buttons were used to enter a code. Therefore, once the buttons had been read, it didn't matter if those buttons were held down for a long time because before entering a new code, the Enter button should be released and A + B + C + D all pressed together. In this chapter, the keys of the matrix keypad will be pressed one after the other, so glitches and bounces could be interpreted as a key being pressed many times, leading to unexpected behavior of the system.

In Figure 4.7, a diagram of the MB102 module is shown. As mentioned previously, this power supply module is used to avoid overloading the capability of the 3.3 V and 5 V pins of the NUCLEO board. In [3] it is stated: "Caution: In case the maximum current consumption of the STM32 Nucleo-144 board and its shield boards exceeds 300 mA, it is mandatory to power the STM32 Nucleo-144 board with an external power supply connected to E5V, VIN or +3.3 V." As the 300 mA limit will be exceeded in the next chapters, hereafter all the elements connected to the NUCLEO board will be powered by the MB102 module, as shown in Figure 4.2. This can provide up to 700 mA [2].

The MB102 module can be supplied either by a standard USB connector or by a 7 to 12 V power supply, as shown in Figure 4.7. The diagram also shows that this module has many 3.3 V and 5 V outputs, some fixed and some selectable.



WARNING: The selectable outputs should be configured as indicated in Figure 4.7. Otherwise, some modules may be harmed.

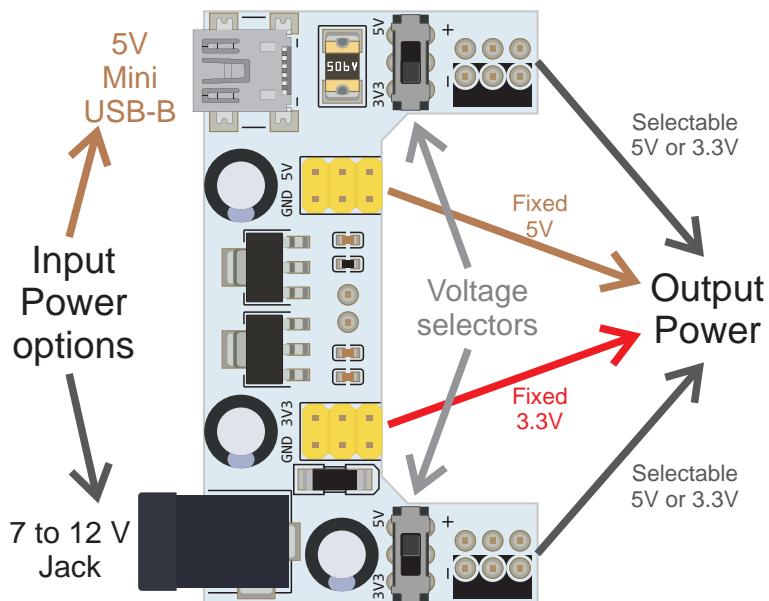


Figure 4.7 Diagram of the MB102 module.

4.2.2 Test the Operation of the Matrix Keypad and the RTC

This subsection explains how to load a program onto the STM32 microcontroller in order to test if the matrix keypad that has been connected is working properly. It will also show how to configure the RTC of the STM32 microcontroller. The serial terminal will display the keys that are pressed on the matrix keypad, and the PC keyboard will be used to configure the date and time of the RTC. The .bin file of the program "Subsection 4.2.2" should be downloaded from the URL available in [4] and dragged onto the NUCLEO board.

In Table 4.1, the available commands for the program that is used in this subsection are shown. If the “k” key is pressed on the PC keyboard, then the buttons pressed on the matrix keypad are shown on the serial terminal. This behavior continues until “q” is pressed on the PC keyboard.

Table 4.1 Available commands for the program used to test the matrix keypad and to configure the RTC.

Key pressed	Description of the commands
k	Show the keys pressed on the matrix keypad
q	Quit the k command
s	Set the current date and time
t	Get the current date and time

Press the “s” key and follow the instructions to set the current date and time for the RTC. Then press the “t” key to get the current date and time. Wait for a few seconds and press “t” again in order to verify that the RTC is working properly. The new date and time shown in the PC should reflect the time progression.



WARNING: The NUCLEO board must be powered in order to keep the RTC working. If the power supply is removed, the time and date of the RTC have to be set again.

Example 4.1: Turn Off the Incorrect Code LED by Double-Pressing the Enter Button

Objective

Introduce enumerated data type definitions and the implementation of FSMs.

Summary of the Expected Behavior

To turn off the Incorrect code LED, the Enter button (B1 USER button of the NUCLEO board) must be double-pressed (pressed twice consecutively). This replaces simultaneously pressing the buttons connected to D4–D7 (A + B + C + D) as implemented in Chapter 1.

Test the Proposed Solution on the Board

Import the project “Example 4.1” using the URL available in [4], build the project, and drag the .bin file onto the NUCLEO board. Press the Alarm test button (button connected to D2) to activate the alarm. The Alarm LED (LD1) should start blinking, at a rate of 100 ms on and 100 ms off. Enter an incorrect deactivation code for the alarm by means of simultaneously pressing the buttons A + C + Enter on the control panel (i.e., D4 + D6 + B1 USER button). The Incorrect code LED (LD3) will turn on. Double press the Enter button (B1 USER button). The Incorrect code LED (LD3) turns off to indicate that a new attempt to enter the code can be made. Deactivate the alarm by simultaneously pressing the buttons A + B + Enter on the control panel (i.e., D4 + D5 + B1 USER button).

Discussion of the Proposed Solution

The proposed solution is based on identifying the state of the Enter button among its four possible states:

- Button released (it is stable “up”)
- Button being pressed (it has just been pressed and is moving from “up” to “down”, i.e., is “falling”)
- Button pressed (it is stable “down”)
- Button being released (it has just been released and is moving from “down” to “up”, i.e., is “rising”)



NOTE: As shown in section 4.2.1, the transition from being “down” to being “up” (or the other way around) entails multiple fast bounces between the two states. For this reason, a *debounce* sequence is used in order to avoid treating bounces as multiple presses of the button.

Implementation of the Proposed Solution

Table 4.2 shows the DEBOUNCE_BUTTON_TIME_MS #define that is used to implement the debounce of the B1 USER button. A value of 40 is used to give a safety margin above the bounce time of 30 milliseconds discussed in subsection 4.2.1.

A new section that is introduced in this chapter is also shown in Table 4.2. This section is called “Declaration of public data types” and is used to implement the definition of new data types that are specified by the programmer. The data type *buttonState_t* is declared using an enumerated data type and four possible values (known as *enumerators*): BUTTON_UP, BUTTON_FALLING, BUTTON_DOWN, and BUTTON_RISING. The suffix “_t” is used to indicate that *buttonState_t* is a user-defined data type.

The variable *accumulatedDebounceButtonTime*, declared as shown in Table 4.2, will be used to account for the debounce time, while the variable *numberOfEnterButtonReleasedEvents* will be used to implement the double-pressed functionality. In addition, a variable *enterButtonState* of the data type *buttonState_t* is declared.

In the section “Declarations (prototypes) of public functions”, three new functions are declared. The function *Im35ReadingsArrayInit()* was introduced and discussed in the proposed exercise of Example 2.5 and therefore is not discussed here again. The functions *debounceButtonInit()* and *debounceButtonUpdate()* will be used to implement the debounce.

Finally, a call to the function *debounceButtonInit()* is included in the function *inputsInit()*.

Table 4.2 Sections and functions in which lines were added to Example 3.5.

Section or function	Lines that were added
Definitions	#define DEBOUNCE_BUTTON_TIME_MS 40
Declaration of public data types	typedef enum { BUTTON_UP BUTTON_FALLING BUTTON_DOWN BUTTON_RISING } buttonState_t;
Declaration and initialization of public global variables	int accumulatedDebounceButtonTime = 0; int numberEnterButtonReleasedEvents = 0; buttonState_t enterButtonState;
Declarations (prototypes) of public functions	void lm35ReadingsArrayInit(); void debounceButtonInit(); bool debounceButtonUpdate();
Function inputsInit()	debounceButtonInit();

In Code 4.1, the function *debounceButtonInit()* is shown. This function is used to establish the initial state of the Enter button. If the Enter button is pressed (line 3), *enterButtonState* is set to *BUTTON_DOWN* (line 4). If it is not pressed, *enterButtonState* is set to *BUTTON_UP* (line 6).

```

1 void debounceButtonInit()
2 {
3     if( enterButton ) {
4         enterButtonState = BUTTON_DOWN;
5     } else {
6         enterButtonState = BUTTON_UP;
7     }
8 }
```

Code 4.1 Details of the function *debounceButtonInit()*.

In Code 4.2, the new implementation of the function *alarmActivationUpdate()* is shown. To allow a new code to be entered after an incorrect code has been entered by pressing the Enter button (B1 USER) twice, lines 4 to 12 are changed from Example 3.5. If the variable *numberOflnCorrectCodes* is less than 5, a Boolean variable called *enterButtonReleasedEvent* is declared (line 4) and is assigned the value returned by the function *debounceButtonUpdate()*. The implementation of *debounceButtonUpdate()* will be discussed in Code 4.3.

On line 5, a check is made as to whether there is a new button released event. If so, line 6 evaluates whether *incorrectCodeLed* is ON, and if so, the variable *numberEnterButtonReleasedEvents* is increased by one. Line 8 evaluates whether there were two or more enter button released events, and if so then the Incorrect code LED is turned off and *numberEnterButtonReleasedEvents* is set to zero (line 9 and line 10). This implements the double-press functionality for the Enter button.



NOTE: In the proposed implementation it doesn't matter how much time elapses between two consecutive presses of the Enter button; they will be considered a double-press.

Lines 12 to 25 show the implementation of the Incorrect code LED deactivation by means of the control panel. The statements are very similar to those used in previous chapters. First, the variable *alarmState* is evaluated (line 13) and, if the alarm LED is on, then the states of the buttons A to D are loaded into the corresponding positions of the array *buttonsPressed*. Then, the function *areEqual()* is used to compare the buttons pressed on the control panel with the corresponding code, and if they are equal, then *alarmState* is set to OFF and *numberOfIncorrectCodes* is set to zero. Otherwise, *incorrectCodeLed* is turned on, and the variable *numberOfIncorrectCodes* is incremented.

Finally, on line 28 it can be seen that if *numberOfIncorrectCodes* is greater than or equal to 5, then *systemBlockedLed* is set to ON.

```

1 void alarmDeactivationUpdate()
2 {
3     if ( numberOfIncorrectCodes < 5 ) {
4         bool enterButtonReleasedEvent = debounceButtonUpdate();
5         if( enterButtonReleasedEvent ) {
6             if( incorrectCodeLed ) {
7                 numberOfEnterButtonReleasedEvents++;
8                 if( numberOfEnterButtonReleasedEvents >= 2 ) {
9                     incorrectCodeLed = OFF;
10                numberOfEnterButtonReleasedEvents = 0;
11            }
12        } else {
13            if ( alarmState ) {
14                buttonsPressed[0] = aButton;
15                buttonsPressed[1] = bButton;
16                buttonsPressed[2] = cButton;
17                buttonsPressed[3] = dButton;
18                if ( areEqual() ) {
19                    alarmState = OFF;
20                    numberOfIncorrectCodes = 0;
21                } else {
22                    incorrectCodeLed = ON;
23                    numberOfIncorrectCodes++;
24                }
25            }
26        }
27    } else {
28        systemBlockedLed = ON;
29    }
30 }
31 }
```

Code 4.2 Modifications introduced in the function *alarmDeactivationUpdate()*.

The implementation of the function *debounceButtonUpdate()* is shown in Code 4.3. On line 3, the Boolean variable *enterButtonReleasedEvent* is declared and initialized to false. On line 5, there is a *switch* statement over the variable *enterButtonState*. In the case of *enterButtonState* being equal to *BUTTON_UP* (line 6), the program verifies if the Enter button has been pressed (line 7). If so, the variable *enterButtonState* is set to *BUTTON_FALLING* (line 8), and *accumulatedDebounceTime* is set to zero.

In the case of `enterButtonState` being equal to `BUTTON_FALLING` (line 13), the program first verifies whether `accumulatedDebounceTime` is greater than or equal to `DEBOUNCE_BUTTON_TIME_MS`. If the Enter button is being pressed (`enterButton` is true, assessed in line 15), then `enterButtonState` is set to `BUTTON_DOWN`; otherwise it is set to `BUTTON_UP`. On line 21, `accumulatedDebounceTime` is incremented by `TIME_INCREMENT_MS`.



NOTE: In this proposed solution, 40 is used in the definition of `DEBOUNCE_BUTTON_TIME_MS`. Depending on the matrix keypad, this time might be too small or too big. The user is encouraged to modify this value if the program behavior is not as expected.

On line 25 it can be seen that the case for `BUTTON_DOWN` is very similar to the case for `BUTTON_UP`. The difference is that if `enterButton` is not true (i.e., is false), then `enterButtonState` is set to `BUTTON_RISING`. The case for `BUTTON_RISING` on line 32 is also very similar to the case for `BUTTON_FALLING`. One difference is that `!enterButton` is used in the `if` statement, as well as the variable `enterButtonReleasedEvent` being set to true on line 36.



NOTE: It should be noted that the following statements are all equivalent:
`if (!enterButton), if (enterButton == 0), if (enterButton == false).`

```

1  bool debounceButtonUpdate()
2  {
3      bool enterButtonReleasedEvent = false;
4      switch( enterButtonState ) {
5
6          case BUTTON_UP:
7              if( enterButton ) {
8                  enterButtonState = BUTTON_FALLING;
9                  accumulatedDebounceButtonTime = 0;
10             }
11             break;
12
13         case BUTTON_FALLING:
14             if( accumulatedDebounceButtonTime >= DEBOUNCE_BUTTON_TIME_MS ) {
15                 if( enterButton ) {
16                     enterButtonState = BUTTON_DOWN;
17                 } else {
18                     enterButtonState = BUTTON_UP;
19                 }
20             }
21             accumulatedDebounceButtonTime = accumulatedDebounceButtonTime +
22                                         TIME_INCREMENT_MS;
23             break;
24
25         case BUTTON_DOWN:
26             if( !enterButton ) {
27                 enterButtonState = BUTTON_RISING;
28                 accumulatedDebounceButtonTime = 0;
29             }
30             break;
31
32         case BUTTON_RISING:
33             if( accumulatedDebounceButtonTime >= DEBOUNCE_BUTTON_TIME_MS ) {
34                 if( !enterButton ) {

```

```
35         enterButtonState = BUTTON_UP;
36         enterButtonReleasedEvent = true;
37     } else {
38         enterButtonState = BUTTON_DOWN;
39     }
40 }
41 accumulatedDebounceButtonTime = accumulatedDebounceButtonTime +
42                                     TIME_INCREMENT_MS;
43 break;
44
45 default:
46     debounceButtonInit();
47     break;
48 }
49 return enterButtonReleasedEvent;
50 }
```

Code 4.3 Details of the function `debounceButtonUpdate()`.



NOTE: In Code 4.3, the four different states are indicated by `BUTTON_UP`, `BUTTON_FALLING`, `BUTTON_DOWN`, and `BUTTON_RISING`. In the Under the Hood section, these four states and their corresponding transitions are shown alongside the signal variations over time, in order to show in more detail how the glitches and bounces are processed.

Proposed Exercise

1. How can the code be modified in order to properly debounce a button with a bouncing time of about 400 ms?

Answer to the Exercise

1. The value of `DEBOUNCE_BUTTON_TIME_MS` could be increased above 400. It should be noted that if the user presses and releases the Enter button in less than 400 ms with this implemented, then the implemented code will ignore the pressing of the Enter Button. The reader is encouraged to test this behavior.

Example 4.2: Introduce the Usage of the Matrix Keypad

Objective

Get familiar with the usage of FSMs.

Summary of the Expected Behavior

The matrix keypad buttons labeled A, B, C, and D should replace the functionality of the buttons connected to D4, D5, D6, and D7, respectively.

Test the Proposed Solution on the Board

Import the project “Example 4.2” using the URL available in [4], build the project, and drag the `.bin` file

onto the NUCLEO board. Press the Alarm test button (button connected to D2) to activate the alarm. The Alarm LED (LD1) should start blinking at a rate of 100 ms on and 100 ms off. Enter an incorrect code to deactivate the alarm by means of pressing first the “A” key, then the “C” key, and finally the “#” key, which is used as the Enter button. The Incorrect code LED (LD3) will turn on. Double click the Enter button (the “#” key). The Incorrect code LED (LD3) turns off to indicate that a new attempt to enter the code can be made. Press the keys “A”, “B”, then “#” and the alarm should deactivate.

Discussion of the Proposed Solution

The proposed solution is based on an FSM that has three states. One state is used to scan the matrix keypad, another state is used to debounce the key pressed at the matrix keypad, and the last state is used to determine if a key has been held pressed or released.

Implementation of the Proposed Solution

In Table 4.3, the #defines that were added to Example 4.1 are shown. The numbers of rows and columns have been defined as four in both cases. A new enumerated data type has also been defined, named *matrixKeypadState_t*, having the three states that will be used in the FSM (MATRIX_KEYPAD_SCANNING, MATRIX_KEYPAD_DEBOUNCE, and MATRIX_KEYPAD_KEY_HOLD_PRESSED).

In the section “Declaration and initialization of public global objects,” two arrays of objects are declared. One, *keypadRowPins*, will be used to introduce signals into the matrix keypad by means of pins PB_3, PB_5, PC_7, and PA_15. This array is declared as an array of DigitalOut. The second array, *keypadColPins*, will be used to read the signals at the pins PB_12, PB_13, PB_15, and PC_6. This is declared as an array of DigitalIn. Note that this is the first time in the book that arrays of DigitalIn and DigitalOut objects are created.

The variables *accumulatedDebounceMatrixKeypadTime* and *matrixKeypadLastKeyPressed* are declared and initialized to zero and the null character ('\0'), respectively. An array of char, *matrixKeypadIndexToCharArray*, is declared and initialized and will be used to identify the keys being pressed on the matrix keypad. Finally, a variable of the user-defined type *matrixKeypadState_t* is declared as *matrixKeypadState*.

In the section “Declarations (prototypes) of public functions,” three new functions are declared: *matrixKeypadInit()*, *matrixKeypadScan()*, and *matrixKeypadUpdate()*. These functions are explained in the example.

Finally, Table 4.3 shows that a call to the function *matrixKeypadInit()* is included in the function *inputsInit()*.

Table 4.3 Sections and functions in which lines were added to Example 4.1.

Section or function	Lines that were added
Definitions	#define KEYPAD_NUMBER_OF_ROWS 4 #define KEYPAD_NUMBER_OF_COLS 4
Declaration of public data types	typedef enum{ MATRIX_KEYPAD_SCANNING, MATRIX_KEYPAD_DEBOUNCE, MATRIX_KEYPAD_KEY_HOLD_PRESSED } matrixKeypadState_t;
Declaration and initialization of public global objects	DigitalOut keypadRowPins[KEYPAD_NUMBER_OF_ROWS] = {PB_3, PB_5, PC_7, PA_15}; DigitalIn keypadColPins[KEYPAD_NUMBER_OF_COLS] = {PB_12, PB_13, PB_15, PC_6};
Declaration and initialization of public global variables	int accumulatedDebounceMatrixKeypadTime = 0; char matrixKeypadLastKeyPressed = '\0'; char matrixKeypadIndexToCharArray[] = { '1', '2', '3', 'A', '4', '5', '6', 'B', '7', '8', '9', 'C', '*', '0', '#', 'D', }; matrixKeypadState_t matrixKeypadState;
Declarations (prototypes) of public functions	void matrixKeypadInit(); char matrixKeypadScan(); char matrixKeypadUpdate();
Function inputsInit()	matrixKeypadInit();

In Code 4.4, the new implementation of the function *alarmDeactivationUpdate()* is shown. The changes begin on line 5, where a variable called *keyReleased* is defined and assigned the returned value of the function *matrixKeypadUpdate()*. On line 6 it can be seen that if *keyReleased* is not equal to the null character or '#', then there is a switch over *keyReleased*; if the key pressed is equal to "A", "B", "C", or "D", then the corresponding position of the array *buttonsPressed* is set to 1.

On line 22 it is determined whether there was an Enter button released event or the "#" key was pressed on the matrix keypad. If so, lines 23 to 27 are executed, having similar behavior to lines 6 to 10 of Code 4.2. The difference is that lines 28 to 31 set all the positions of the array *buttonsPressed* to zero. Note that in the implementations of *alarmDeactivationUpdate()* used in previous chapters, it was not necessary to set all the positions of the array *buttonsPressed* to zero because all the buttons were read simultaneously. In the case of the matrix keypad, the keys are pressed one after the other, and when a given key is pressed, a "1" is stored in the corresponding position of the *buttonsPressed* array. For example, if "A" is pressed, a "1" is stored in *buttonsPressed[0]*, and if "B" is pressed, a "1" is stored in *buttonsPressed[1]*. Because of this, the array must be reset (i.e., all its positions set to zero) in order to allow a new attempt to enter the code.

Note that because this implementation is the same, the order in which the keys are pressed, or even if one of the keys is pressed many times, is irrelevant. For example, if the user presses the keys "A", "B", "#", "#" it will be considered a correct code, but "B", "A", "#", "#" or "A", "A", "B", "#", "#" will also be considered correct.

The remaining lines of Code 4.4 are identical to the corresponding lines of Code 4.2.

```

1 void alarmDeactivationUpdate()
2 {
3     if ( numberOfIncorrectCodes < 5 ) {
4         bool enterButtonReleasedEvent = debounceButtonUpdate();
5         char keyReleased = matrixKeypadUpdate();
6         if( keyReleased != '\0' && keyReleased != '#' ) {
7             switch (keyReleased) {
8                 case 'A':
9                     buttonsPressed[0] = 1;
10                break;
11                case 'B':
12                    buttonsPressed[1] = 1;
13                    break;
14                case 'C':
15                    buttonsPressed[2] = 1;
16                    break;
17                case 'D':
18                    buttonsPressed[3] = 1;
19                    break;
20            }
21        }
22        if( enterButtonReleasedEvent || keyReleased == '#' ) {
23            if( incorrectCodeLed ) {
24                numberOfEnterButtonReleasedEvents++;
25                if( numberOfEnterButtonReleasedEvents >= 2 ) {
26                    incorrectCodeLed = OFF;
27                    numberOfEnterButtonReleasedEvents = 0;
28                    buttonsPressed[0] = 0;
29                    buttonsPressed[1] = 0;
30                    buttonsPressed[2] = 0;
31                    buttonsPressed[3] = 0;
32                }
33            } else {
34                if ( alarmState ) {
35                    if ( enterButtonReleasedEvent ) {
36                        buttonsPressed[0] = aButton;
37                        buttonsPressed[1] = bButton;
38                        buttonsPressed[2] = cButton;
39                        buttonsPressed[3] = dButton;
40                    }
41                    if ( areEqual() ) {
42                        alarmState = OFF;
43                        numberOfIncorrectCodes = 0;
44                    } else {
45                        incorrectCodeLed = ON;
46                        numberOfIncorrectCodes++;
47                    }
48                }
49            }
50        }
51    } else {
52        systemBlockedLed = ON;
53    }
54 }
```

Code 4.4 Details of the function `alarmDeactivationUpdate()`.

Code 4.5 shows the implementation of the function `matrixKeypadInit()`. The initial state of `matrixKeypadState` is set on line 3 to `MATRIX_KEYPAD_SCANNING`. On line 4, the variable `pinIndex` is declared and set to zero. The `for` loop on line 5 is used to properly configure each of the pins of `keypadColPins`.

```
1 void matrixKeypadInit()
2 {
3     matrixKeypadState = MATRIX_KEYPAD_SCANNING;
4     int pinIndex = 0;
5     for( pinIndex=0; pinIndex<KEYPAD_NUMBER_OF_COLS; pinIndex++ ) {
6         (keypadColPins[pinIndex]).mode(PullUp);
7     }
8 }
```

Code 4.5 Details of the function `matrixKeypadInit()`.

The implementation of the function `matrixKeypadScan()` is shown in Code 4.6. On lines 3 and 4, the variables `row` and `col` are declared. They will be used as indexes in `for` loops to indicate which row and column is being scanned. The variable `i` is used in another `for` loop, as is explained below.

On line 7, it can be seen that there is a `for` loop that is used to scan all the rows of the matrix keypad. On line 9, the four keypad row pins are first set to ON by means of a `for` loop. On line 13, the pin of the current row being scanned is set to OFF. On line 15, another `for` loop is used to scan all the columns, one after the other. If a given key is being pressed, its value is returned on line 17 by returning the value in the appropriate position of the array `matrixKeypadIndexToCharArray`. Otherwise, if no key is being pressed in the matrix keypad, then the null character ('\0') is returned on line 21.



NOTE: Once a key press is detected, the scanning is stopped, as can be seen on line 17 of Code 4.6. In this way, if, for example, keys "1" and "2" are pressed simultaneously, only key "1" is reported. In the same way, if keys "A" and "B" are pressed simultaneously, only key "A" is reported.

```
1 char matrixKeypadScan()
2 {
3     int row = 0;
4     int col = 0;
5     int i = 0;
6
7     for( row=0; row<KEYPAD_NUMBER_OF_ROWS; row++ ) {
8
9         for( i=0; i<KEYPAD_NUMBER_OF_ROWS; i++ ) {
10             keypadRowPins[i] = ON;
11         }
12
13         keypadRowPins[row] = OFF;
14
15         for( col=0; col<KEYPAD_NUMBER_OF_COLS; col++ ) {
16             if( keypadColPins[col] == OFF ) {
17                 return matrixKeypadIndexToCharArray[row*KEYPAD_NUMBER_OF_ROWS + col];
18             }
19         }
20     }
21     return '\0';
22 }
```

Code 4.6 Details of the function `matrixKeypadScan()`.

Code 4.7 shows the implementation of the function *matrixKeypadUpdate()*. On lines 3 and 4, the variables *keyDetected* and *keyReleased* are declared and initialized to the null character. On line 6 there is a switch over the variable *matrixKeypadState*. In the case of *MATRIX_KEYPAD_SCANNING*, the matrix keypad is scanned, and the resulting value is stored in *keyDetected*. If no key was pressed (identified on line 10), then *matrixKeypadLastKeyPressed* is assigned the value of *keyDetected*, *accumulatedDebounceMatrixKeypadTime* is set to zero, and *matrixKeypadState* is set to *MATRIX_KEYPAD_DEBOUNCE*.

In the case of *MATRIX_KEYPAD_DEBOUNCE*, if *accumulatedDebounceMatrixKeypadTime* is greater than or equal to *DEBOUNCE_BUTTON_TIME_MS*, then the matrix keypad is scanned, and the resulting value is stored in *keyDetected* (line 20). If *keyDetected* is equal to *matrixKeypadLastKeyPressed*, then *matrixKeypadState* is set to *MATRIX_KEYPAD_HOLD_PRESSED*. Otherwise, *matrixKeypadState* is set to *MATRIX_KEYPAD_SCANNING*. Finally, on line 27, *accumulatedDebounceMatrixKeypadTime* is incremented.

```

1  char matrixKeypadUpdate()
2  {
3      char keyDetected = '\0';
4      char keyReleased = '\0';
5
6      switch( matrixKeypadState ) {
7
8          case MATRIX_KEYPAD_SCANNING:
9              keyDetected = matrixKeypadScan();
10             if( keyDetected != '\0' ) {
11                 matrixKeypadLastKeyPressed = keyDetected;
12                 accumulatedDebounceMatrixKeypadTime = 0;
13                 matrixKeypadState = MATRIX_KEYPAD_DEBOUNCE;
14             }
15             break;
16
17         case MATRIX_KEYPAD_DEBOUNCE:
18             if( accumulatedDebounceMatrixKeypadTime >=
19                 DEBOUNCE_BUTTON_TIME_MS ) {
20                 keyDetected = matrixKeypadScan();
21                 if( keyDetected == matrixKeypadLastKeyPressed ) {
22                     matrixKeypadState = MATRIX_KEYPAD_KEY_HOLD_PRESSED;
23                 } else {
24                     matrixKeypadState = MATRIX_KEYPAD_SCANNING;
25                 }
26             }
27             accumulatedDebounceMatrixKeypadTime =
28                 accumulatedDebounceMatrixKeypadTime + TIME_INCREMENT_MS;
29             break;
30
31         case MATRIX_KEYPAD_KEY_HOLD_PRESSED:
32             keyDetected = matrixKeypadScan();
33             if( keyDetected != matrixKeypadLastKeyPressed ) {
34                 if( keyDetected == '\0' ) {
35                     keyReleased = matrixKeypadLastKeyPressed;
36                 }
37                 matrixKeypadState = MATRIX_KEYPAD_SCANNING;
38             }
39             break;
40
41         default:
42             matrixKeypadInit();
43             break;
44     }
45     return keyReleased;
46 }
```

Code 4.7 Details of the function *matrixKeypadUpdate()*.

The case for MATRIX_KEYPAD_HOLD_PRESSED is shown on line 31. First, the matrix keypad is scanned (line 32). If *keyDetected* is not equal to *matrixKeypadLastKeyPressed* and if *keyDetected* is equal to the null character, then *matrixKeypadLastKeyPressed* is assigned to *keyReleased*. The fact that the state remains in MATRIX_KEYPAD_HOLD_PRESSED avoids the issue of a key being held for a long time and the same value being returned many times. In this way, it exits the state only if the pressed key is released or if a key connected to a row or column with a “higher priority” in the scanning (i.e., a lower number of *row* or *col*) is pressed.

Finally, *matrixKeypadState* is assigned to MATRIX_KEYPAD_SCANNING. This is done to allow the detection of a new key being pressed, as the MATRIX_KEYPAD_SCANNING state is the only one in which the FSM is waiting for a new key to be pressed.

Line 41 implements the “default” statement of the implementation of the FSM. It ensures that the function *matrixKeypadInit()* is executed if for any reason the value of *matrixKeypadState* is neither MATRIX_KEYPAD_SCANNING, MATRIX_KEYPAD_DEBOUNCE, nor MATRIX_KEYPAD_HOLD_PRESSED.



NOTE: Defining a default case in the implementation of the FSM is a safety measure that is strongly recommended to handle errors.

Finally, on line 45, the value of *keyReleased* is returned. This value was used in Code 4.4 as described previously.

Proposed Exercise

1. What should be adapted in the code if a keypad having five rows and five columns is to be used?

Answer to the Exercise

1. The definitions KEYPAD_NUMBER_OF_ROWS and KEYPAD_NUMBER_OF_COLS should be set to 5, and more elements should be added to *keypadRowPins*, *keypadColPins*, and *matrixKeypadIndexToCharArray*.

Example 4.3: Implementation of Numeric Codes using the Matrix Keypad

Objective

Explore more advanced functionality regarding the usage of the matrix keypad.

Summary of the Expected Behavior

The code implemented in the previous example, based only on the keys A, B, C, and D, is replaced by a numeric code that is entered by means of the matrix keypad.



NOTE: In this example, the buttons connected to D4-D7 (*aButton-dButton*) are not used anymore. Consequently, *buttonBeingPressed* will be replaced by *keyBeingPressed*, as discussed below.

Test the Proposed Solution on the Board

Import the project “Example 4.3” using the URL available in [4], build the project, and drag the .bin file onto the NUCLEO board. Press the Alarm test button (implemented hereafter with B1 USER button) to activate the alarm. The Alarm LED (LD1) should start blinking at a rate of 100 ms on and 100 ms off. Press the keys “1”, “8”, “0”, “5”, and “#” on the matrix keypad. The Alarm LED (LD1) should be turned off. Press the Alarm test button to activate the alarm. The Alarm LED (LD1) should start blinking. Press the keys “1”, “8”, “5”, “5” (incorrect code), and “#” on the matrix keypad. The Incorrect code LED (LD3) should be turned on. Press “#” twice in the matrix keypad. The incorrect code LED (LD3) should be turned off.



NOTE: The code “1805” is configured by default in this example; the user can change it by pressing “5” on the PC keyboard. The code would be “1805” again after resetting or powering off the NUCLEO board.

Press the Alarm test button again to activate the alarm. Now press the “4” key on the PC keyboard. Type the code “1805” to deactivate the alarm. Now press the “5” key on the PC keyboard. The code can be modified.

Discussion of the Proposed Solution

The proposed solution is based on the program code that was introduced in previous examples. By means of the matrix keypad functionality that was presented in Example 4.1 and Example 4.2, the keys pressed by the user are read and compared with the correct code (1805). The function *uartTask()* is modified in order to adapt the commands related to pressing keys “4” and “5” on the PC keyboard. These are the commands used to enter a code from the PC and to change the correct code from the PC, respectively.

Implementation of the Proposed Solution

Table 4.4 shows the variables *matrixKeypadCodeIndex* and *numberOfHashKeyReleasedEvents* that are declared in this example. The variable *matrixKeypadCodeIndex* will be used to keep track of the buttons that are pressed on the matrix keypad. The variable *numberOfHashKeyReleasedEvents* will be used to keep track of the number of times that the “#” key of the matrix keypad is pressed. In addition, Table 4.4 shows that *BUTTON1* is assigned to the *alarmTestButton* object. Therefore, B1 User is now the Alarm test button and the button connected to D2 can be removed from the setup.

In Table 4.5, the definitions, variable names, and variable initializations that were modified are shown. It can be seen that “button” was replaced by “key” and the array of char *codeSequence* is assigned { '1', '8', '0', '5' }. Note that it is not a string because it is not ended by a null character, '\0'. Because

the functionality of the Enter button and the buttons connected to D4–D7 is replaced by the matrix keypad, all the data types, variables, and functions related to them are removed. This is shown in Table 4.6 and Table 4.7. Additionally, the implementations of the functions *debounceButtonInit()* and *debounceButtonUpdate()* are removed.



NOTE: *codeSequence* and *keyPressed* are the only arrays of char in this book initialized using = { 'x', 'y', 'z' }. In upcoming chapters, it will be shown how to assign values to an array of char when it is used as a string.

Table 4.4 Sections in which lines were added to Example 4.2.

Section	Lines that were modified
Declaration and initialization of public global variables	int matrixKeypadCodeIndex = 0; int numberOfHashKeyReleasedEvents = 0;
Declaration and initialization of public global objects	DigitalIn alarmTestButton(BUTTON1);

Table 4.5 Definitions, variable names, and variable initializations that were modified from Example 4.2.

Declaration in Example 4.2	Declaration in Example 4.3
#define DEBOUNCE_BUTTON_TIME_MS 40	#define DEBOUNCE_KEY_TIME_MS 40
int buttonBeingCompared = 0;	int keyBeingCompared = 0;
int codeSequence[NODE_OF_KEYS] = { 1, 1, 0, 0 };	char codeSequence[NODE_OF_KEYS] = { '1', '8', '0', '5' };
int buttonsPressed[NODE_OF_KEYS] = { 0, 0, 0, 0 };	char keyPressed[NODE_OF_KEYS] = { '0', '0', '0', '0' };

Table 4.6 Sections in which lines were removed from Example 4.2.

Section	Lines that were removed
Declaration of public data types	typedef enum { BUTTON_UP, BUTTON_DOWN, BUTTON_FALLING, BUTTON_RISING } buttonState_t;
Declaration and initialization of public global objects	DigitalIn enterButton(BUTTON1); DigitalIn alarmTestButton(D2); DigitalIn aButton(D4); DigitalIn bButton(D5); DigitalIn cButton(D6); DigitalIn dButton(D7);
Declaration and initialization of public global variables	int accumulatedDebounceButtonTime = 0; int numberOfEnterButtonReleasedEvents = 0; buttonState_t enterButtonState;
Declarations (prototypes) of public functions	void debounceButtonInit(); bool debounceButtonUpdate();

Table 4.7 Functions in which lines were removed from Example 4.2.

Section	Lines that were removed
void inputsInit()	aButton.mode(PullDown); bButton.mode(PullDown); cButton.mode(PullDown); dButton.mode(PullDown); debounceButtonInit();

Code 4.8 shows the new implementation of the function *alarmDeactivationUpdate()*. On line 3, the number of incorrect codes is checked to see if it is less than 5. If so, the matrix keypad is read on line 4. If there was a released key (i.e., *keyReleased* != '\0') and if the released key was not "#", then the *keyReleased* is assigned to the current position of the *buttonsPressed* array (line 6). On line 7, a check is made to see if *matrixKeypadCodeIndex* is greater than or equal to *NUMBER_OF_KEYS*. If so, *matrixKeypadCodeIndex* is set to zero, and if not, then it is incremented by one.

The remaining lines of Code 4.8 are identical to the corresponding lines of Code 4.2 except for the removal of the lines related to the external buttons and the addition of line 26 that assigns 0 to *matrixKeypadCodeIndex*.

```

1 void alarmDeactivationUpdate()
2 {
3     if ( numberOfIncorrectCodes < 5 ) {
4         char keyReleased = matrixKeypadUpdate();
5         if( keyReleased != '\0' && keyReleased != '#' ) {
6             keyPressed[matrixKeypadCodeIndex] = keyReleased;
7             if( matrixKeypadCodeIndex >= NUMBER_OF_KEYS ) {
8                 matrixKeypadCodeIndex = 0;
9             } else {
10                 matrixKeypadCodeIndex++;
11             }
12         }
13         if( keyReleased == '#' ) {
14             if( incorrectCodeLed ) {
15                 numberOfHashKeyReleasedEvents++;
16                 if( numberOfHashKeyReleasedEvents >= 2 ) {
17                     incorrectCodeLed = OFF;
18                     numberOfHashKeyReleasedEvents = 0;
19                     matrixKeypadCodeIndex = 0;
20                 }
21             } else {
22                 if( alarmState ) {
23                     if( areEqual() ) {
24                         alarmState = OFF;
25                         numberOfIncorrectCodes = 0;
26                         matrixKeypadCodeIndex = 0;
27                     } else {
28                         incorrectCodeLed = ON;
29                         numberOfIncorrectCodes++;
30                     }
31                 }
32             }
33         } else {
34             systemBlockedLed = ON;
35         }
36     }
37 }
```

Code 4.8 Details of the function *alarmDeactivationUpdate()*.

In Code 4.9, the lines that were modified in the function *uartTask()* are shown. In the case of '4', the user is asked to enter the four-digit numeric code (lines 2 and 3), and *incorrectCode* is set to false (line 5). On line 7, there is a *for* loop, where the keys pressed on the PC keyboard are read until *NUMBER_OF_KEYS* keys have been read sequentially. The read keys are stored in *receivedChar* (line 10) and compared with the corresponding position of *codeSequence* on line 11; *incorrectCode* is set to true (line 12) if one of the keys does not match the code sequence. Line 14 is used to print a "*" on the PC in correspondence with each key that is pressed.

If *incorrectCode* is equal to false on line 17, then the user is informed (line 18) and the corresponding values of *alarmState*, *incorrectCodeLed*, and *numberOfIncorrectCodes* are set (lines 19 to 21). Otherwise, if the code is incorrect, the user is informed (line 23), *incorrectCodeLed* is set to ON, and *numberOfIncorrectCodes* is incremented by one.

In the case of '5' (line 29), the user is asked to enter the new four-digit numeric code. The *for* loop from lines 33 to 38 is used to get the keys and store them in *codeSequence*. Line 12 is used to inform the user that the new code has been configured.

```

1  case '4':
2      uartUsb.write( "Please enter the four digits numeric code ", 42 );
3      uartUsb.write( "to deactivate the alarm: ", 25 );
4
5      incorrectCode = false;
6
7      for ( keyBeingCompared = 0;
8          keyBeingCompared < NUMBER_OF_KEYS;
9          keyBeingCompared++ ) {
10         uartUsb.read( &receivedChar, 1 );
11         uartUsb.write( "*", 1 );
12         if ( codeSequence[keyBeingCompared] != receivedChar ) {
13             incorrectCode = true;
14         }
15     }
16
17     if ( incorrectCode == false ) {
18         uartUsb.write( "\r\nThe code is correct\r\n\r\n", 25 );
19         alarmState = OFF;
20         incorrectCodeLed = OFF;
21         numberOfIncorrectCodes = 0;
22     } else {
23         uartUsb.write( "\r\nThe code is incorrect\r\n\r\n", 27 );
24         incorrectCodeLed = ON;
25         numberOfIncorrectCodes++;
26     }
27     break;
28
29 case '5':
30     uartUsb.write( "Please enter the new four digits numeric code ", 46 );
31     uartUsb.write( "to deactivate the alarm: ", 25 );
32
33     for ( keyBeingCompared = 0;
34         keyBeingCompared < NUMBER_OF_KEYS;
35         keyBeingCompared++ ) {
36         uartUsb.read( &receivedChar, 1 );
37         uartUsb.write( "*", 1 );
38     }
39
40     uartUsb.write( "\r\nNew code generated\r\n\r\n", 24 );
41     break;

```

Code 4.9 Lines that were modified in the function *uartTask()*.

Proposed Exercise

1. How can the code be modified in order to use a three-digit code?

Answer to the Exercise

1. In the arrays `codeSequence` and `buttonsPressed`, three positions must be assigned, and `NUMBER_OF_KEYS` must be defined as 3.

Example 4.4: Report Date and Time of Alarms to the PC Based on the RTC

Objective

Introduce the use of data structures and the RTC.

Summary of the Expected Behavior

The smart home system should store up to 20 events, each one with the corresponding date and time of occurrence, and display those events on the serial terminal when they are requested.

Test the Proposed Solution on the Board

Import the project “Example 4.4” using the URL available in [4], build the project, and drag the `.bin` file onto the NUCLEO board. Press “s” on the PC keyboard in order to configure the date and time of the RTC of the NUCLEO board. Press “t” on the PC keyboard to confirm that the RTC is working properly. Press the Alarm test button to activate the alarm. The Alarm LED (LD1) should start blinking at a rate of 100 ms on and 100 ms off. Enter the code to deactivate the alarm (1805#). Press “e” on the PC keyboard to view the date and time that the gas and over temperature detection and alarm activation occurred.

Discussion of the Proposed Solution

The proposed solution is based on the RTC of the STM32 microcontroller of the NUCLEO board. Its date and time are used to tag the events related to the alarm. In order to have a meaningful date and time related to each event, the RTC must be configured.



NOTE: The registered events and the date and time configuration of the RTC are lost when power is removed from the NUCLEO board.

Implementation of the Proposed Solution

Table 4.8 shows the sections in which lines were added to Example 4.3. First, two `#defines` were included: `EVENT_MAX_STORAGE`, to limit the number of stored events to 20, and `EVENT_NAME_MAX_LENGTH`, to limit the number of characters associated with each event.

In the section “Declaration of public data types,” a new public data type is declared. The reserved word `struct` is used to declare special types of variables that have internal members. These members can

have different types and different lengths. The type `systemEvents_t` is declared, having two members: `seconds`, of type `time_t`, and an array of char called `typeOfEvent`.

The type `time_t` used to represent times is part of the standard C++ library and is implemented by the Mbed OS [5]. For historical reasons, it is generally implemented as an integer value representing the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC, which is usually called Unix timestamp or epoch time. The maximum date that can be represented using this format is 03:14:07 UTC on 19 January 2038.

Many variables are also declared. The variable `eventsIndex` will be used to keep track of the number of events stored.

Table 4.8 shows that an array `arrayOfStoredEvents` is declared, being of type `systemEvents_t` (the struct that has been declared in the section “Declaration of public data types”), as well as the variable `timeAux` of type `time_t`.

Finally, two functions are declared: `eventLogUpdate()` and `systemElementStateUpdate()`. These functions will be shown and analyzed in the code below.

Table 4.8 Sections in which lines were added to Example 4.3.

Section	Lines that were added
Definitions	#define EVENT_MAX_STORAGE 20 #define EVENT_NAME_MAX_LENGTH 14
Declaration of public data types	typedef struct systemEvent { time_t seconds; char typeOfEvent[EVENT_NAME_MAX_LENGTH]; } systemEvent_t;
Declaration and initialization of public global variables	bool alarmLastState = OFF; bool gasLastState = OFF; bool tempLastState = OFF; bool ICLastState = OFF; bool SBLastState = OFF; int eventsIndex = 0; systemEvent_t arrayOfStoredEvents[EVENT_MAX_STORAGE];
Declarations (prototypes) of public functions	void eventLogUpdate(); void systemElementStateUpdate(bool lastState, bool currentState, const char* elementName);

In Code 4.10, some of the lines that were included in the function `uartTask()` are shown. Starting at lines 1 and 2, the reader can see that in case of the keys “s” or “S” being pressed, the variable `rtcTime` is declared in line 3, being a struct of the type `tm`. The struct `tm` is part of the standard C++ library,

is implemented by Mbed OS, and has the members detailed in Table 4.9. The member `tm_sec` is generally in the range 0–59, but sometimes 60 is used, or even 61 to accommodate leap seconds in certain systems. The Daylight Saving Time flag (`tm_isdst`) is greater than zero if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and less than zero if the information is not available. In line 4, `strIndex` is declared to be used as discussed below.

Table 4.9 Details of the struct `tm`.

Member	Type	Meaning	Range
<code>tm_sec</code>	int	seconds after the minute	0–59
<code>tm_min</code>	int	minutes after the hour	0–59
<code>tm_hour</code>	int	hours since midnight	0–23
<code>tm_mday</code>	int	day of the month	1–31
<code>tm_mon</code>	int	months since January	0–11
<code>tm_year</code>	int	years since 1900	
<code>tm_wday</code>	int	days since Sunday	0–6
<code>tm_yday</code>	int	days since January 1	0–365
<code>tm_isdst</code>	int	Daylight Saving Time flag	

The user is asked to enter, one after the other, the values of most of the members of the variable `rtcTime` (lines 6 to 58). In each case, first a message is displayed using `uartUsb.write()`. Then, a `for` loop is used to read a character, store that character in a given position of the array, and send that character to the serial terminal using `uartUsb.write()`, so the user has an echo of the entered character. After this, the null character is appended to the string, as, for example, on line 11. Then the function `atoi()`, which is provided by Mbed OS, is used to convert the string to an integer, as, for example, on line 12. The resulting value is stored in the corresponding member of `rtcTime`, as can be seen on line 12. Finally, “\r\n” is written to move to a new line.

Note that some minor operations are made on the values entered by the user, such as on line 12, where 1900 is subtracted, and on line 211, where 1 is subtracted. Also note that `str[strIndex]` is preceded by the reference operator (&), as in line 8 or 9. The reference operator is related to the usage of *pointers* and will be discussed in more detail in upcoming chapters. Finally, note that no check is made on the digits entered by the user. If values outside the ranges indicated in Table 4.9 are entered, unexpected behavior may result.



NOTE: The scope of `rtcTime` and `strIndex` is the switch-case where they are declared (i.e., those variables don't exist outside the brackets of the corresponding switch-case).

```

1 case 's':
2 case 'S':
3     struct tm rtcTime;
4     int strIndex;
5
6     uartUsb.write( "\r\nType four digits for the current year (YYYY): ", 48 );
7     for( strIndex=0; strIndex<4; strIndex++ ) {
8         uartUsb.read( &str[strIndex] , 1 );
9         uartUsb.write( &str[strIndex] ,1 );
10    }
11    str[4] = '\0';
12    rtcTime.tm_year = atoi(str) - 1900;
13    uartUsb.write( "\r\n", 2 );
14
15    uartUsb.write( "Type two digits for the current month (01-12): ", 47 );
16    for( strIndex=0; strIndex<2; strIndex++ ) {
17        uartUsb.read( &str[strIndex] , 1 );
18        uartUsb.write( &str[strIndex] ,1 );
19    }
20    str[2] = '\0';
21    rtcTime.tm_mon  = atoi(str) - 1;
22    uartUsb.write( "\r\n", 2 );
23
24    uartUsb.write( "Type two digits for the current day (01-31): ", 45 );
25    for( strIndex=0; strIndex<2; strIndex++ ) {
26        uartUsb.read( &str[strIndex] , 1 );
27        uartUsb.write( &str[strIndex] ,1 );
28    }
29    str[2] = '\0';
30    rtcTime.tm_mday = atoi(str);
31    uartUsb.write( "\r\n", 2 );
32
33    uartUsb.write( "Type two digits for the current hour (00-23): ", 46 );
34    for( strIndex=0; strIndex<2; strIndex++ ) {
35        uartUsb.read( &str[strIndex] , 1 );
36        uartUsb.write( &str[strIndex] ,1 );
37    }
38    str[2] = '\0';
39    rtcTime.tm_hour = atoi(str);
40    uartUsb.write( "\r\n", 2 );
41
42    uartUsb.write( "Type two digits for the current minutes (00-59): ", 49 );
43    for( strIndex=0; strIndex<2; strIndex++ ) {
44        uartUsb.read( &str[strIndex] , 1 );
45        uartUsb.write( &str[strIndex] ,1 );
46    }
47    str[2] = '\0';
48    rtcTime.tm_min  = atoi(str);
49    uartUsb.write( "\r\n", 2 );
50
51    uartUsb.write( "Type two digits for the current seconds (00-59): ", 49 );
52    for( strIndex=0; strIndex<2; strIndex++ ) {
53        uartUsb.read( &str[strIndex] , 1 );
54        uartUsb.write( &str[strIndex] ,1 );
55    }
56    str[2] = '\0';
57    rtcTime.tm_sec  = atoi(str);
58    uartUsb.write( "\r\n", 2 );
59
60    rtcTime.tm_isdst = -1;
61    set_time( mktime( &rtcTime ) );
62    uartUsb.write( "Date and time has been set\r\n", 28 );
63
64    break;

```

Code 4.10 Lines that were included in the function `uartTask()` (Part 1/2).

On line 60, the value of the member `tm_isdst` is set to “-1” to indicate that the information is not available. On line 61, two operations are carried out. First the function `mktime()`, which is provided by the implementation of the library `time.h` by Mbed OS, is used to convert the variable `rtcTime` from the `tm` structure to the `time_t` structure. Then, the function `set_time()` is called to set the time on the RTC of the STM32 microcontroller. This function is also provided by Mbed OS. Note that the reference operator (&) is used in line 61. Lastly, the message “Date and time has been set” is written in line 62.

The case of the keys “t” or “T” being pressed is shown in Code 4.11. Lines 3 and 4 are used to declare the variable `epochSeconds`, of type `time_t`, and to store the value of the RTC of the STM32 microcontroller in the variable `epochSeconds`. This is done on line 4 by means of the function `time()`, which is also provided by Mbed OS. Then, the function `ctime()` is used on line 5 to convert the `time_t` value of seconds to a string having the format `Www Mmm dd hh:mm:ss yyyy`, where `Www` is the weekday, `Mmm` the month (in letters), `dd` the day of the month, `hh:mm:ss` the time, and `yyyy` the year. The string is written to `uartUsb` on line 6.

In the case of the keys “e” or “E” being pressed, all the events stored in `arrayOfStoredEvents` are transmitted one after the other to the PC, as can be seen on lines 10 to 21 of Code 4.11.

```

1  case 't':
2  case 'T':
3      time_t epochSeconds;
4      epochSeconds = time(NULL);
5      sprintf ( str, "Date and Time = %s", ctime(&epochSeconds) );
6      uartUsb.write( str , strlen(str) );
7      uartUsb.write( "\r\n" , 2 );
8      break;
9
10 case 'e':
11 case 'E':
12     for (int i = 0; i < eventsIndex; i++) {
13         sprintf ( str, "Event = %s\r\n",
14                 arrayOfStoredEvents[i].typeOfEvent);
15         uartUsb.write( str , strlen(str) );
16         sprintf ( str, "Date and Time = %s\r\n",
17                 ctime(&arrayOfStoredEvents[i].seconds));
18         uartUsb.write( str , strlen(str) );
19         uartUsb.write( "\r\n" , 2 );
20     }
21     break;

```

Code 4.11 Lines that were included in the function `uartTask()` (part 2/2).

Code 4.12 shows the new implementation of the function `availableCommands()`. The new values “s”, “t”, and “e” have been included.

```

1 void availableCommands()
2 {
3     uartUsb.write( "Available commands:\r\n", 21 );
4     uartUsb.write( "Press '1' to get the alarm state\r\n", 34 );
5     uartUsb.write( "Press '2' to get the gas detector state\r\n", 41 );
6     uartUsb.write( "Press '3' to get the over temperature detector state\r\n", 54 );
7     uartUsb.write( "Press '4' to enter the code sequence\r\n", 38 );
8     uartUsb.write( "Press '5' to enter a new code\r\n", 31 );
9     uartUsb.write( "Press 'f' or 'F' to get lm35 reading in Fahrenheit\r\n", 52 );
10    uartUsb.write( "Press 'c' or 'C' to get lm35 reading in Celsius\r\n", 49 );
11    uartUsb.write( "Press 's' or 'S' to set the date and time\r\n", 43 );
12    uartUsb.write( "Press 't' or 'T' to get the date and time\r\n", 43 );
13    uartUsb.write( "Press 'e' or 'E' to get the stored events\r\n\r\n", 45 );
14 }

```

Code 4.12 New implementation of the function `availableCommands()`.

In order to periodically check if there is an event to be stored, the `main()` function is modified, as can be seen in Code 4.13. A call to the function `eventLogUpdate()` has been added on line 9.

```

1 int main()
2 {
3     inputsInit();
4     outputsInit();
5     while (true) {
6         alarmActivationUpdate();
7         alarmDeactivationUpdate();
8         uartTask();
9         eventLogUpdate();
10        delay(TIME_INCREMENT_MS);
11    }
12 }

```

Code 4.13 New implementation of the function `main()`.

In Code 4.14, the implementation of the function `eventLogUpdate()` is shown. It calls the function `systemElementStateUpdate()` to determine if there has been a change in the state of any of the elements. For example, on line 3, the function `systemElementStateUpdate()` is called to determine if the state of the alarm has changed. After calling `systemElementStateUpdate()`, the value of `alarmLastState` is updated on line 4. On the following lines (6 to 16), the same procedure is followed for the gas detector, the over temperature, the Incorrect code LED, and the System blocked LED.

```

1 void eventLogUpdate()
2 {
3     systemElementStateUpdate( alarmLastState, alarmState, "ALARM" );
4     alarmLastState = alarmState;
5
6     systemElementStateUpdate( gasLastState, !mq2, "GAS_DET" );
7     gasLastState = !mq2;
8
9     systemElementStateUpdate( tempLastState, overTempDetector, "OVER_TEMP" );
10    tempLastState = overTempDetector;
11
12    systemElementStateUpdate( ICLastState, incorrectCodeLed, "LED_IC" );
13    ICLastState = incorrectCodeLed;
14
15    systemElementStateUpdate( SBLastState, systemBlockedLed, "LED_SB" );
16    SBLastState = systemBlockedLed;
17 }

```

Code 4.14 Implementation of the function `eventLogUpdate()`.

The implementation of the function `systemElementStateUpdate()` is shown in Code 4.15. This function accepts three parameters: the last state, the current state, and the element name. The element names are stored in arrays of char type, so the third parameter of this function is a memory address of an array of char. This is indicated by `char*` (line 3), which means “*a pointer to a char type*.” In this way, `elementName` is a pointer that points to the first position of an array of char. Note that the third parameter type is declared as `const char*` (line 3). In this context, the addition of the reserved word `const` indicates that the content of the memory address pointed by `elementName` cannot be modified by the function `systemElementStateUpdate()`. The usage of pointers is discussed in detail in upcoming chapters.

On line 5, an array of char called `eventAndStateStr` is declared, having `EVENT_NAME_MAX_LENGTH` positions. It is initialized using “”, which assigns the null character to its first position (i.e., `eventAndStateStr[0] = '\0'`), which makes `eventAndStateStr` an *empty string* (a string with no printable characters). On line 7, it is determined if `lastState` is different from `currentState`. If so, on line 9 the content of the array of char pointed by `elementName` is appended to the string `eventAndStateStr` by means of the function `strncat()`, provided by Mbed OS.



NOTE: More functions regarding strings are available in [6]. Some of them are used in the next chapters.

In lines 10 to 14, ON or OFF is appended to `eventAndStateStr`, depending on the value of `currentState`. The members of `arrayOfStoredEvents` (`seconds` and `typeOfEvent`) at the position `eventsIndex` are assigned the time of the RTC of the STM32 microcontroller using the function `time()` (line 16) and the type of event by means of the function `strcpy()`, provided by Mbed OS (line 17). On line 18, a check is made whether `eventsIndex` is smaller than `EVENT_MAX_STORAGE - 1`. If so, there is still space in the array to store new events, and `eventsIndex` is incremented by one. If not, the array is full, and `eventsIndex` is set to zero in order to start filling the array `arrayOfStoredEvents` again from its first position. Finally, the `eventAndStateStr` is printed on the serial terminal (lines 24 and 25).

```

1 void systemElementStateUpdate( bool lastState,
2                               bool currentState,
3                               const char* elementName )
4 {
5     char eventAndStateStr[EVENT_NAME_MAX_LENGTH] = "";
6
7     if ( lastState != currentState ) {
8
9         strcat( eventAndStateStr, elementName );
10        if ( currentState ) {
11            strcat( eventAndStateStr, "_ON" );
12        } else {
13            strcat( eventAndStateStr, "_OFF" );
14        }
15
16        arrayOfStoredEvents[eventsIndex].seconds = time(NULL);
17        strcpy( arrayOfStoredEvents[eventsIndex].typeOfEvent, eventAndStateStr );
18        if ( eventsIndex < EVENT_MAX_STORAGE - 1 ) {

```

```
19         eventsIndex++;
20     } else {
21         eventsIndex = 0;
22     }
23
24     uartUsb.write( eventAndStateStr , strlen(eventAndStateStr) );
25     uartUsb.write( "\r\n" , 2 );
26 }
27 }
```

Code 4.15 Implementation of the function `systemElementStateUpdate()`.



WARNING: The improper usage of pointers can lead to software errors. In upcoming chapters it will be shown that the memory address pointed to by the pointer can be modified (i.e., increased or decreased) and that a value can be assigned to the memory address pointed to by the pointer using the reference operator (&), as, for example, in line 8 of Code 4.10. This means that the pointer can be pointed to a memory address that is already in use and an improper modification of the content of that memory address can be made.

A similar problem can take place when string-related functions are used without the proper precautions. For example, in line 9 of Code 4.15, a copy of `elementName` is appended to `eventAndStateStr`. This operation will happen no matter the number of positions that were reserved for `eventAndStateStr` in line 5 of Code 4.15. Therefore, a “buffer overflow” may occur if `elementName` has more characters than `EVENT_NAME_MAX_LENGTH`. The usage of objects of type `string` instead of using arrays of `char` can be a solution in certain situations, but it may lead to memory issues when applied in the context of embedded systems. In upcoming chapters, different solutions for safely managing strings in embedded systems are discussed.

Proposed Exercise

1. How can a change be implemented in the code in order to allow up to 1000 events to be stored?

Answer to the Exercise

1. The value of `MAX_NUMBER_OF_EVENTS` should be changed to 1000.

4.3 Under the Hood

4.3.1 Graphical Representation of a Finite-State Machine

This section explains how an FSM can be represented by means of a diagram. It is common to start by drawing the diagram of the FSM, then analyze and adjust the behavior of the system using the diagram. Only when the behavior is as expected is the corresponding code implemented.

Before introducing the graphical representation, the behavior of the FSM should be reviewed.

In Figure 4.8, the voltage at a given button over time is shown by a red line, following the diagram that was introduced in section 4.2.1. Initially (at t_0) the button is released, the voltage is 3.3 V, and the FSM is in the **BUTTON_UP** state (as indicated by the light blue line). Then there is a *glitch* at t_1 , after which the signal goes back to 3.3 V. It can be seen that the FSM state changes to **BUTTON_FALLING** (because of the glitch) and then reverts to **BUTTON_UP** at t_2 because it is determined that it was not an actual change in the button state.

At t_3 , bounce in the voltage signal is shown because the button is pressed. It can be seen that the FSM state changed to **BUTTON_FALLING**. At t_4 , the FSM state changes to **BUTTON_DOWN** as the voltage signal is stable at 0 V. At t_5 there is a new glitch, after which the FSM state changes to **BUTTON_RISING**. But, as after the glitch the voltage signal remains at 0 V, the FSM state reverts back to **BUTTON_DOWN** at t_6 .

At t_7 , there is a transition from released to pressed, which is accompanied by a bounce in the signal. It can be seen that the FSM state changed to **BUTTON_RISING**. After the bounce time, the signal stabilizes to 3.3 V and, therefore, at t_8 the FSM state changes to **BUTTON_UP**.

The behavior shown in Figure 4.8 continues over time as the button is pressed and released. It might be that there are no glitches or there are many glitches. If so, the transitions between the FSM states will be the same as shown in Figure 4.8, with the only difference being the number of **BUTTON_FALLING** states between two consecutive **BUTTON_UP** and **BUTTON_DOWN** states and the number of **BUTTON_RISING** states between two consecutive **BUTTON_DOWN** and **BUTTON_UP** states.

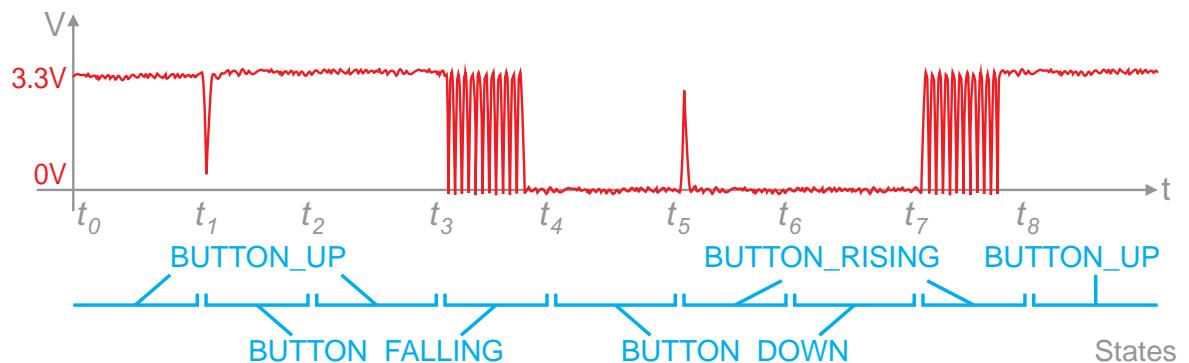


Figure 4.8 Voltage signal over time for a given button as it is pressed or released.

Figure 4.9 shows a state diagram for the FSM discussed above Figure 4.8. This is the FSM that was implemented in Example 4.1 by means of the statements shown in Code 4.3.



NOTE: In Figure 4.9, and in the corresponding discussion that is presented below, “== 1” is used to indicate that a given Boolean variable is in the *true* state, and “== 0” is used to indicate that a given Boolean variable is in the *false* state. Do not confuse this with “= 0”, which is used to indicate that the value zero is assigned to the integer variable *accumulatedDebounceTime*. In the case of the Boolean variable *enterButtonReleasedEvent*, “= true” is used to indicate that the logical value *true* is assigned.

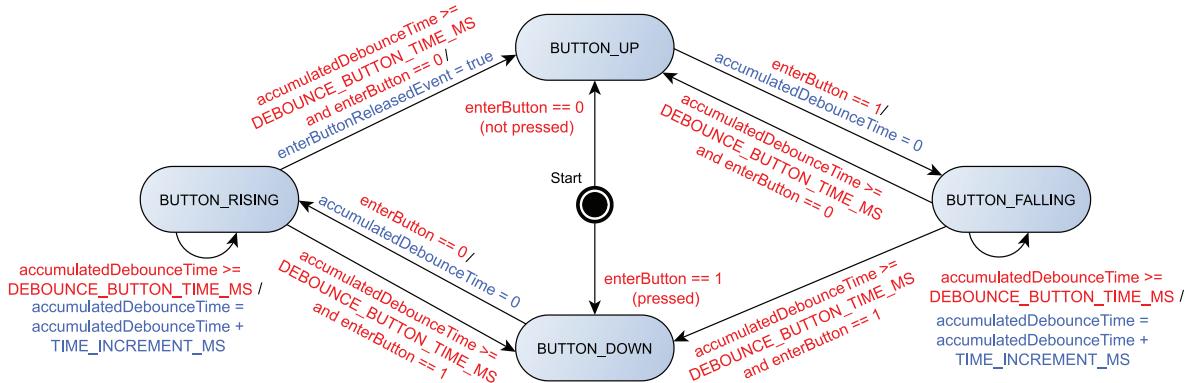


Figure 4.9 Diagram of the FSM implemented in Example 4.1.

The four states of the FSM implemented in Code 4.3 are represented in Figure 4.9 by means of the light blue ovals (i.e., BUTTON_UP, BUTTON_RISING, BUTTON_DOWN, and BUTTON_FALLING). The arrows from one state to another state, or from a given state to itself, or from the start point to a state, represent a *transition*. A transition can be between a given state and another state, to the same state, or could be the initial transition.

The red labels over the arrows indicate the *input* that triggers a given transition, while the blue label over an arrow indicates the *action* that is performed before the transition takes place. Note that every transition is triggered by a given input, but not all the transitions have an action associated with them.

After *power on*, the FSM is at the start point. Depending on the value of *enterButton*, the initial transition from the start point is to BUTTON_UP (if *enterButton* is 0, implying that the button is initially not pressed) or to BUTTON_DOWN (if *enterButton* is 1, implying that the button is initially pressed). This was implemented in the *debounceButtonInit()* function shown in Code 4.1. The FSM will remain in those states (BUTTON_UP or BUTTON_DOWN) while there are no changes in the value of *enterButton*.

Consider, for example, that at the start point *enterButton* is 0 (implying that the button is initially not pressed) and, therefore, the first state is BUTTON_UP. By referring back to Code 4.3, it can be seen that the only way of moving to a different state is if *enterButton* becomes 1 (i.e., the button is pressed). In that condition, BUTTON_FALLING is assigned to *enterButtonState* (line 8 of Code 4.3), and *accumulatedDebounceTime* is set to zero (line 9). This is shown in Figure 4.9 by the arrow from

BUTTON_UP to BUTTON_FALLING, and a red label “enterButton == 1” indicates the condition that triggers the transition, as well as the blue label “accumulatedDebounceTime = 0” indicating the action that is performed during the transition.

Once in the BUTTON_FALLING state, Code 4.3 can be analyzed to determine all the possible transitions. It can be seen on line 14 that in order to have a transition, *accumulatedDebounceTime* has to reach DEBOUNCE_BUTTON_TIME_MS. In that situation, if *enterButton* is 1 (i.e., the button is pressed), the transition is to the BUTTON_DOWN state (line 16), and if *enterButton* is 0 (i.e., the button is not pressed), the transition is to the BUTTON_UP state (line 18). While in BUTTON_FALLING state, *accumulatedDebounceTime* is incremented by TIME_INCREMENT_MS every 10 ms. This is indicated by the re-entering arrow above BUTTON_FALLING and is implemented on line 22 of Code 4.3.

The only possible transition from BUTTON_DOWN is to BUTTON_RISING. This transition takes place if *enterButton* == 0, as is shown in Figure 4.9. This represents the behavior of line 26 in Code 4.3.

Finally, it can be seen in Figure 4.9 that the behavior at the BUTTON_RISING state is very similar to the behavior at the BUTTON_FALLING state, in the same way as lines 13 to 23 of Code 4.3 are very similar to lines 32 to 43 of the same code. The only difference is that in the BUTTON_FALLING state, the variable *enterButtonReleasedEvent()* is set to true on line 36. This call is shown in Figure 4.9 by the arrow that goes from BUTTON_RISING to BUTTON_UP.



NOTE: The states BUTTON_UP and BUTTON_DOWN last until there is a glitch or the button is either pressed or released. The states BUTTON_FALLING and BUTTON_RISING always last the same amount of time (i.e., DEBOUNCE_BUTTON_TIME_MS), as can be seen in Figure 4.8.



WARNING: The diagram used in Figure 4.9 is only one of multiple possible representations of an FSM, known as a “Mealy machine.” It is beyond the scope of this book to introduce other representations of FSMs.

Proposed Exercise

- How can the FSM used in the implementation of Example 4.2 be represented by means of a diagram similar to the one used in Figure 4.9 to illustrate the FSM of Example 4.1?

Answer to the Exercise

- The diagram is shown in Figure 4.10.

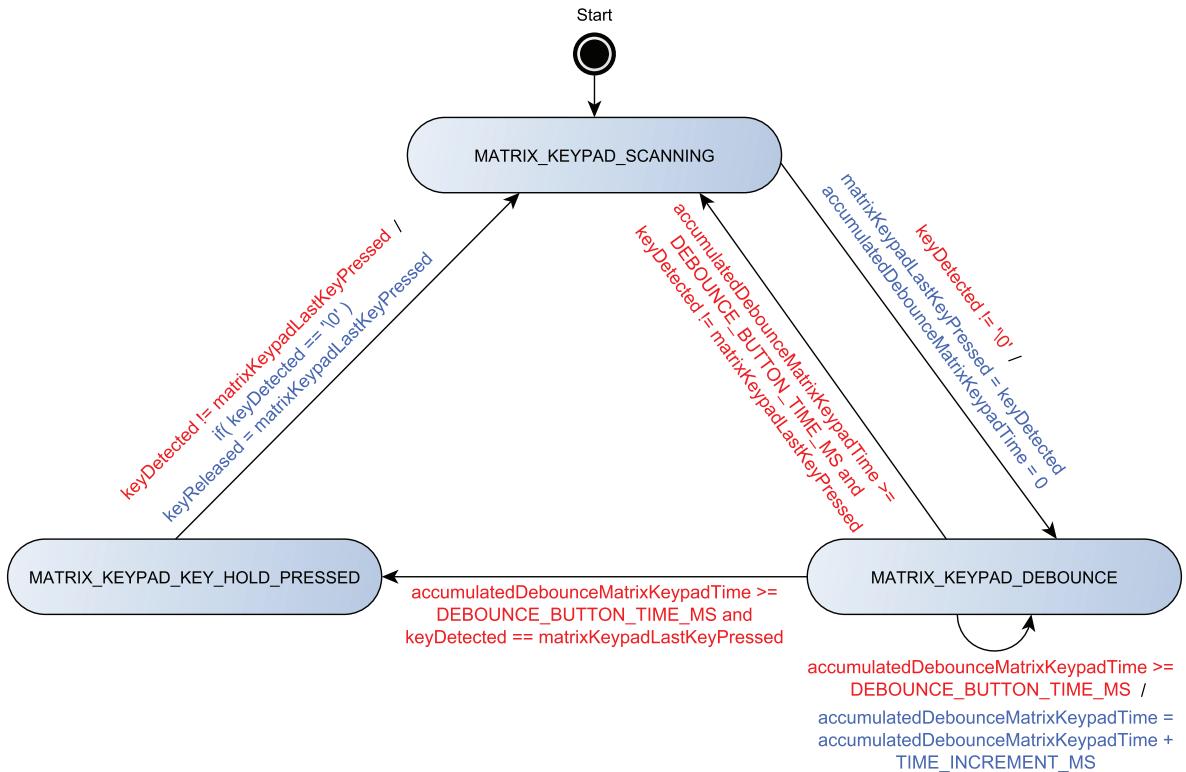


Figure 4.10 Diagram of the FSM implemented in Example 4.2.

4.4 Case Study

4.4.1 Smart Door Locks

In chapter 1, the case study of the smart door lock was introduced [7]. A representation of the keypad that is used by the smart door lock is shown in Figure 4.11. This is very similar to the matrix keypad that was introduced in this chapter.

The smart door lock is not provided with RTC functionality. However, the reader may realize that interesting access control features could be included if information about date and time is available. For example, each user may be provided with a specific time range to open the lock. This idea is explored in the proposed exercise.



Figure 4.11 Smart door lock built with Mbed contains a keypad similar to the one introduced in this chapter.

Proposed Exercise

1. How can a program be implemented in order to achieve the following behavior?

- The lock should open only if the code 1-4-7 is entered, and the current time is within the designated opening hours (8 am to 4 pm).
- To enter the code, the letter “A” should be pressed.

Answer to the Exercise

1. The proposed solution is shown in this section. The aim is to analyze a whole program (from the include files to all the functions used) in order to familiarize the reader with solving real-life problems.

Test the Proposed Solution on the Board

Import the project “Case Study Chapter 4” using the URL available in [4], build the project, and drag the .bin file onto the NUCLEO board. The LED LD2 will turn on to indicate that the door is locked. Press the “s” key on the PC keyboard to set a time between 8 am and 4 pm. Press the “t” key on the PC keyboard to get the current time and date. Press the keys “A”, “1”, “4”, “7” on the matrix keypad. The LED LD1 should turn on to indicate that the door is now open. Press the B1 USER button to represent that the door has been closed. The LED LD2 will turn on to indicate that the door is locked.

Press the keys “A”, “1”, “2”, “3” on the matrix keypad. The LED LD3 will turn on to indicate that an incorrect code has been entered. Press the keys “A”, “1”, “4”, “7” on the matrix keypad. The LED LD1 should turn on to indicate that the door is now open. Press the B1 USER button to represent that the door has been closed. The LED LD2 will turn on to indicate that the door is locked.

Press the “s” key on the PC keyboard to set a time not in the range of 8 am to 4 pm. Press the keys “A”, “1”, “4”, “7” on the matrix keypad. The LED LD1 will not turn on because it is not the correct opening hour.

Discussion of the Proposed Solution

The proposed solution is very similar to the solution presented throughout this chapter. However, some variations have been introduced in order to show the reader other ways to implement the code. These variations are discussed below as the proposed implementation is introduced.

Implementation of the Proposed Solution

In Code 4.16, the libraries that are used in the proposed solution are included: *mbed.h* and *arm_book_lib.h*. The definitions are also shown in Code 4.16. The number of digits of the code is defined as 3 (line 8). Then, the number of rows and columns of the keypad is defined (lines 9 and 10). The two definitions that are used in the FSMs to manage time, *TIME_INCREMENT_MS* and *DEBOUNCE_BUTTON_TIME_MS*, are defined on lines 11 and 12. Finally, on lines 13 and 14, the opening hours are defined.

In Code 4.17, the declaration of the public data type *doorState_t* is shown. It is used to implement an FSM and has three states: DOOR_CLOSED, DOOR_UNLOCKED, and DOOR_OPEN. In Code 4.17, the data type *matrixKeypadState_t* is also declared, just as in Example 4.2.

```
1 //=====[Libraries]=====
2
3 #include "mbed.h"
4 #include "arm_book_lib.h"
5
6 //=====[Defines]=====
7
8 #define CODE_DIGITS          3
9 #define KEYPAD_NUMBER_OF_ROWS 4
10 #define KEYPAD_NUMBER_OF_COLS 4
11 #define TIME_INCREMENT_MS    10
12 #define DEBOUNCE_BUTTON_TIME_MS 40
13 #define START_HOUR           8
14 #define END_HOUR             16
```

Code 4.16 Libraries and definitions used in the proposed solution.

```

1 //=====[Declaration of public data types]=====
2
3 typedef enum {
4     DOOR_CLOSED,
5     DOOR_UNLOCKED,
6     DOOR_OPEN
7 } doorState_t;
8
9 typedef enum {
10    MATRIX_KEYPAD_SCANNING,
11    MATRIX_KEYPAD_DEBOUNCE,
12    MATRIX_KEYPAD_KEY_HOLD_PRESSED
13 } matrixKeypadState_t;

```

Code 4.17 Declaration of the public data types *doorState_t* and *matrixKeypadState_t*.

The section “Declaration and initialization of public global objects” is shown in Code 4.18. On line 3, an array of DigitalOut objects, called *keypadRowPins*, is defined. This array is used to indicate the pins used to connect the matrix keypad pins corresponding to the rows, as in Example 4.2. A similar implementation is used on line 4 to indicate the pins used to connect the matrix keypad columns.

On line 6 of Code 4.18, the DigitalIn object *doorHandle* is declared and linked to the B1 User button. This button will be used to indicate whether the door handle is in the opened or locked position. LD1, LD2, and LD3 are assigned on lines 8 to 10 to *doorUnlockedLed*, *doorLockedLed*, and *incorrectCodeLed*, respectively. Finally, on line 12 the UnbufferedSerial object *uartUsb* is created, in the same way as in the smart home system.

```

1 //=====[Declaration and initialization of public global objects]=====
2
3 DigitalOut keypadRowPins[KEYPAD_NUMBER_OF_ROWS] = {PB_3, PB_5, PC_7, PA_15};
4 DigitalIn keypadColPins[KEYPAD_NUMBER_OF_COLS] = {PB_12, PB_13, PB_15, PC_6};
5
6 DigitalIn doorHandle(BUTTON1);
7
8 DigitalOut doorUnlockedLed(LED1);
9 DigitalOut doorLockedLed(LED2);
10 DigitalOut incorrectCodeLed(LED3);
11
12 UnbufferedSerial uartUsb(USBTX, USBRX, 115200);

```

Code 4.18 Declaration of the public global objects.

In Code 4.19, the section “Declaration and initialization of public global variables” is shown. On lines 3 and 5, the variables *accumulatedDebounceMatrixKeypadTime* and *matrixKeypadLastKeyPressed* are declared and initialized to zero and the null character, respectively. On line 6, an array of char that will be used to get the character corresponding to the pressed key is defined. This is based on the column and row that are activated, as in Example 4.2. On line 12, a variable of the user-defined type *matrixKeypadState_t* is declared as *matrixKeypadState*. On line 14, the variable *DoorState*, which is used to implement the FSM, is declared. On lines 16 and 18, the variables *rtcTime* and *seconds* are declared, just as in Example 4.4. Finally, on line 22 the array *codeSequence* is declared and assigned the code 1-4-7.

```

1 //=====[Declaration and initialization of public global variables]=====
2
3 int accumulatedDebounceMatrixKeypadTime = 0;
4
5 char matrixKeypadLastkeyReleased = '\0';
6 char matrixKeypadIndexToCharArray[] = {
7     '1', '2', '3', 'A',
8     '4', '5', '6', 'B',
9     '7', '8', '9', 'C',
10    '*', '0', '#', 'D',
11 };
12 matrixKeypadState_t matrixKeypadState;
13
14 doorState_t doorState;
15
16 struct tm RTCTime;
17
18 time_t seconds;
19
20 char codeSequence[CODE_DIGITS] = {'1', '4', '7'};

```

Code 4.19 Declaration and initialization of public global variables.

In Code 4.20, the public functions are declared. The functions *uartTask()* and *availableCommands()* have the same role as in the smart home system but will have an implementation that is specific to this proposed exercise. The functions *doorInit()* and *doorUpdate()* are used to initiate and implement the FSM related to the door. The functions *matrixKeypadInit()*, *matrixKeypadScan()*, and *matrixKeypadUpdate()* are used to initiate, scan, and update the state of the matrix keypad, respectively. From lines 10 to 12, three functions that will be used to get and send strings and characters using serial communication with the PC are declared. Note that some of these functions use pointers (lines 10 and 12), a concept that was introduced in Example 3.5.

```

1 //===== [Declarations (prototypes) of public functions] =====
2
3 void uartTask();
4 void availableCommands();
5 void doorInit();
6 void doorUpdate();
7 void matrixKeypadInit();
8 char matrixKeypadScan();
9 char matrixKeypadUpdate();
10 void pcSerialComStringWrite( const char* str );
11 char pcSerialComCharRead();
12 void pcSerialComStringRead( char* str, int strLength );

```

Code 4.20 Declaration of public functions used in the proposed solution.

The implementation of the *main()* function is shown in Code 4.21. First, the door and the matrix keypad are initialized (lines 5 and 6) and then there is a *while (true)* loop to continuously update the door state (line 8) and communicate with the PC using the uart (line 9).

```

1 //=====[Main function, the program entry point after power on or reset]=====
2
3 int main()
4 {
5     doorInit();
6     matrixKeypadInit();
7     while (true) {
8         doorUpdate();
9         uartTask();
10    }
11 }
```

Code 4.21 Declaration of public functions used in the proposed solution.

In Code 4.22, the implementation of the `uartTask()` function is shown. The principal idea is the same as in the `uartTask()` function used in the smart home system implementation. If the key “s” is pressed, then the user is asked to enter the date and time (lines 17 to 52), in quite a similar way to the implementation introduced in Example 4.4. By means of comparing lines 19 to 52 with Code 4.10, the reader can see that the same functionality is now obtained using a more modular program. Finally, in lines 54 to 60, the implementation for the key “t” is shown, as in Example 4.4.



NOTE: The function `pcSerialComStringRead()` that is called on lines 20, 25, 30, 35, 40, and 45 reads the number of characters indicated by its second parameter (for example, four characters when it is called on line 20), stores the read characters in the array of char indicated by its first parameter (e.g., `year` when it is called on line 20), and writes the null character, '\0', in the next position of the array (e.g., the fifth position of `year` when it is called on line 20). The implementation of `pcSerialComStringRead()` is discussed below.

```

1 void uartTask()
2 {
3     char str[100] = "";
4     char receivedChar = '\0';
5     struct tm rtcTime;
6     char year[5] = "";
7     char month[3] = "";
8     char day[3] = "";
9     char hour[3] = "";
10    char minute[3] = "";
11    char second[3] = "";
12    time_t epochSeconds;
13    receivedChar = pcSerialComCharRead();
14    if( receivedChar != '\0' ) {
15        switch (receivedChar) {
16
17            case 's':
18            case 'S':
19                pcSerialComStringWrite("\r\nType four digits for the current year (YYYY): ");
20                pcSerialComStringRead( year, 4);
21                pcSerialComStringWrite("\r\n");
22                rtcTime.tm_year = atoi(year) - 1900;
23
```

```

24         pcSerialComStringWrite("Type two digits for the current month (01-12): ");
25         pcSerialComStringRead( month, 2 );
26         pcSerialComStringWrite("\r\n");
27         rtcTime.tm_mon = atoi(month) - 1;
28
29         pcSerialComStringWrite("Type two digits for the current day (01-31): ");
30         pcSerialComStringRead( day, 2 );
31         pcSerialComStringWrite("\r\n");
32         rtcTime.tm_hour = atoi(hour);
33
34         pcSerialComStringWrite("Type two digits for the current hour (00-23): ");
35         pcSerialComStringRead( hour, 2 );
36         pcSerialComStringWrite("\r\n");
37         rtcTime.tm_hour = atoi(hour);
38
39         pcSerialComStringWrite("Type two digits for the current minutes (00-59): ");
40         pcSerialComStringRead( minute, 2 );
41         pcSerialComStringWrite("\r\n");
42         rtcTime.tm_min = atoi(minute);
43
44         pcSerialComStringWrite("Type two digits for the current seconds (00-59): ");
45         pcSerialComStringRead( second, 2 );
46         pcSerialComStringWrite("\r\n");
47         rtcTime.tm_sec = atoi(second);
48
49         rtcTime.tm_isdst = -1;
50         set_time( mktime( &rtcTime ) );
51         pcSerialComStringWrite("Date and time has been set\r\n");
52         break;
53
54     case 't':
55     case 'T':
56         epochSeconds = time(NULL);
57         sprintf( str, "Date and Time = %s", ctime(&epochSeconds));
58         pcSerialComStringWrite( str );
59         pcSerialComStringWrite("\r\n");
60         break;
61
62     default:
63         availableCommands();
64         break;
65     }
66 }
67 }
```

Code 4.22 Implementation of the function `uartTask()`.

The implementation of the function `availableCommands()` is shown in Code 4.23. This function is used to list all the available commands. In this particular case there are only two: set the time and get the time.

In Code 4.24, the statements used in the function `doorInit()` are shown. The LEDs used to indicate that the door is unlocked and that an incorrect code has been entered are turned off. The LED used to indicate that the door is locked is turned on, and the door state is set to DOOR_CLOSED.

Code 4.25 shows the implementation of the function `doorUpdate()`. From lines 3 to 7, the variables `incorrectCode`, `keyPressed`, `currentTime`, `prevKeyPressed`, and `i` are declared. The variable `currentTime` is preceded by a `*` symbol. This indicates that this variable is a *pointer* and is used because the function `localtime` (on line 14) needs this type of variable, as discussed in Example 4.4.

On line 9, there is a switch over the `doorState` variable. In the case of `DOOR_CLOSED`, the matrix keypad is scanned (line 11), and if the “A” key is pressed, the date and time of the RTC of the STM32 microcontroller is assigned to `currentTime` (line 14). On line 16, it is determined whether the current time corresponds to the opening hours. If so, `incorrectCode` is set to false (line 17), and `prevKeyPressed` is set to “A” (line 18). The `for` loop on lines 21 to 31 is used to read a number of digits equal to `CODE_DIGITS` (line 21). Lines 22 to 26 are used to wait until a new key (different to the previous one) is pressed. On line 27, the new key is stored in `prevKeyReleased`. On line 28, the key pressed is compared with the corresponding digit of the code; if they are not equal, then `incorrectCode` is set to true. On line 33, `incorrectCode` is evaluated and if true then the Incorrect code LED is turned on for one second (lines 34 to 36); otherwise, `doorState` is set to `DOOR_UNLOCKED`, the `doorLockedLED` is turned off, and the `doorUnlockedLED` is turned on.

In the case of `DOOR_UNLOCKED` (line 46), if `doorHandle` is true, then `doorUnlockedLED` is set to OFF and `doorState` is set to `DOOR_OPEN`. Lastly, in the case of `DOOR_OPEN` (line 53), if `doorHandle` is false, `doorLockedLED` is set to ON and `doorState` is set to `DOOR_CLOSED`. In the *default* case, the function `doorInit()` is called, as was described in Example 4.2 (it is safest to always define a default case).

In Code 4.26, Code 4.27, and Code 4.28, the implementation of the functions `matrixKeypadInit()`, `matrixKeypadScan()`, and `matrixKeypadUpdate()` are shown. It can be seen that this is the same code as in Code 4.6. Therefore, the explanation is not repeated here.

```

1 void availableCommands()
2 {
3     pcSerialComStringWrite( "Available commands:\r\n" );
4     pcSerialComStringWrite( "Press 's' or 'S' to set the time\r\n\r\n" );
5     pcSerialComStringWrite( "Press 't' or 'T' to get the time\r\n\r\n" );
6 }
```

Code 4.23 Implementation of the function `availableCommands()`.

```

1 void doorInit()
2 {
3     doorUnlockedLed = OFF;
4     doorLockedLed = ON;
5     incorrectCodeLed = OFF;
6     doorState = DOOR_CLOSED;
7 }
```

Code 4.24 Implementation of the function `doorInit()`.

```

1 void doorUpdate()
2 {
3     bool incorrectCode;
4     char keyReleased;
5     struct tm * currentTime;
6     char prevkeyReleased;
7     int i;
8
9     switch( doorState ) {
10    case DOOR_CLOSED:
11        keyReleased = matrixKeypadUpdate();
12        if ( keyReleased == 'A' ) {
13            seconds = time(NULL);
14            currentTime = localtime ( &seconds );
15
16            if ( ( currentTime->tm_hour >= START_HOUR ) &&
17                ( currentTime->tm_hour <= END_HOUR ) ) {
18                incorrectCode = false;
19                prevkeyReleased = 'A';
20
21                for ( i = 0; i < CODE_DIGITS; i++ ) {
22                    while ( ( keyReleased == '\0' ) ||
23                            ( keyReleased == prevkeyReleased ) ) {
24
25                        keyReleased = matrixKeypadUpdate();
26
27                        prevkeyReleased = keyReleased;
28                        if ( keyReleased != codeSequence[i] ) {
29                            incorrectCode = true;
30                        }
31                    }
32
33                    if ( incorrectCode ) {
34                        incorrectCodeLed = ON;
35                        delay (1000);
36                        incorrectCodeLed = OFF;
37                    } else {
38                        doorState = DOOR_UNLOCKED;
39                        doorLockedLed = OFF;
40                        doorUnlockedLed = ON;
41                    }
42                }
43            }
44            break;
45
46        case DOOR_UNLOCKED:
47            if ( doorHandle ) {
48                doorUnlockedLed = OFF;
49                doorState = DOOR_OPEN;
50            }
51            break;
52
53        case DOOR_OPEN:
54            if ( !doorHandle ) {
55                doorLockedLed = ON;
56                doorState = DOOR_CLOSED;
57            }
58            break;
59
60        default:
61            doorInit();
62            break;
63        }
64    }
}

```

Code 4.25 Implementation of the function `doorUpdate()`.

```

1 void matrixKeypadInit()
2 {
3     matrixKeypadState = MATRIX_KEYPAD_SCANNING;
4     int pinIndex = 0;
5     for( pinIndex=0; pinIndex<KEYPAD_NUMBER_OF_COLS; pinIndex++ ) {
6         (keypadColPins[pinIndex]).mode(PullUp);
7     }
8 }
```

Code 4.26 Implementation of the function keypadInit().

```

1 char matrixKeypadScan()
2 {
3     int r = 0;
4     int c = 0;
5     int i = 0;
6
7     for( r=0; r<KEYPAD_NUMBER_OF_ROWS; r++ ) {
8
9         for( i=0; i<KEYPAD_NUMBER_OF_ROWS; i++ ) {
10             keypadRowPins[i] = ON;
11         }
12
13         keypadRowPins[r] = OFF;
14
15         for( c=0; c<KEYPAD_NUMBER_OF_COLS; c++ ) {
16             if( keypadColPins[c] == OFF ) {
17                 return matrixKeypadIndexToCharArray[r*KEYPAD_NUMBER_OF_ROWS + c];
18             }
19         }
20     }
21     return '\0';
22 }
```

Code 4.27 Implementation of the function matrixKeypadScan().

```

1 char matrixKeypadUpdate()
2 {
3     char keyDetected = '\0';
4     char keyReleased = '\0';
5
6     switch( matrixKeypadState ) {
7
8     case MATRIX_KEYPAD_SCANNING:
9         keyDetected = matrixKeypadScan();
10        if( keyDetected != '\0' ) {
11            matrixKeypadLastkeyReleased = keyDetected;
12            accumulatedDebounceMatrixKeypadTime = 0;
13            matrixKeypadState = MATRIX_KEYPAD_DEBOUNCE;
14        }
15        break;
16
17     case MATRIX_KEYPAD_DEBOUNCE:
18        if( accumulatedDebounceMatrixKeypadTime >=
19            DEBOUNCE_BUTTON_TIME_MS ) {
20            keyDetected = matrixKeypadScan();
21            if( keyDetected == matrixKeypadLastkeyReleased ) {
22                matrixKeypadState = MATRIX_KEYPAD_KEY_HOLD_PRESSED;
23            } else {
24                matrixKeypadState = MATRIX_KEYPAD_SCANNING;
25            }
26        }
27        accumulatedDebounceMatrixKeypadTime =
28            accumulatedDebounceMatrixKeypadTime + TIME_INCREMENT_MS;
29    }
```

```

29         break;
30
31     case MATRIX_KEYPAD_KEY_HOLD_PRESSED:
32         keyDetected = matrixKeypadScan();
33         if( keyDetected != matrixKeypadLastkeyReleased ) {
34             if( keyDetected == '\0' ) {
35                 keyReleased = matrixKeypadLastkeyReleased;
36             }
37             matrixKeypadState = MATRIX_KEYPAD_SCANNING;
38         }
39         break;
40
41     default:
42         matrixKeypadInit();
43         break;
44     }
45     return keyReleased;
46 }
```

Code 4.28 Implementation of the function matrixKeypadUpdate().

Finally, in Code 4.29 the functions related to sending and receiving characters using the serial communication with the PC are shown. On line 1, `pcSerialComStringWrite()` is implemented in order to be able to send a string to the serial terminal. `pcSerialComCharRead()` on line 6 implements the reading of a single character. It returns '\0' if there is not a character to be read, or the received character otherwise. Lastly, `pcSerialComStringRead()` implements the reading of a number of characters specified by its second parameter, `strLength`. The read characters are stored in the array of char pointed to by the first parameter of this function, `str`.



WARNING: If the value of `strLength` is greater than the number of positions in the array of char pointed by `str`, then a buffer overflow will take place. As discussed in Example 4.4, this can lead to software errors.

```

1 void pcSerialComStringWrite( const char* str )
2 {
3     uartUsb.write( str, strlen(str) );
4 }
5
6 char pcSerialComCharRead()
7 {
8     char receivedChar = '\0';
9     if( uartUsb.readable() ) {
10         uartUsb.read( &receivedChar, 1 );
11     }
12     return receivedChar;
13 }
14
15 void pcSerialComStringRead( char* str, int strLength )
16 {
17     int strIndex;
18     for ( strIndex = 0; strIndex < strLength; strIndex++ ) {
19         uartUsb.read( &str[strIndex] , 1 );
20         uartUsb.write( &str[strIndex] , 1 );
21     }
22     str[strLength]='\0';
23 }
```

Code 4.29 Implementation of the functions related to the PC serial communication.

References

- [1] “4x4 Keypad Module Pinout, Configuration, Features, Circuit & Datasheet”. Accessed July 9, 2021.
- [2] “Breadboard Power Supply Module”. Accessed July 9, 2021.
<https://components101.com/modules/5v-mb102-breadboard-power-supply-module>
- [3] “UM1974 User manual - STM32 Nucleo-144 boards (MB1137)”. Accessed July 9, 2021.
https://www.st.com/resource/en/user_manual/dm00244518-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf
- [4] “GitHub - armBookCodeExamples/Directory”. Accessed July 9, 2021.
<https://github.com/armBookCodeExamples/Directory/>
- [5] “Time - API references and tutorials | Mbed OS 6 Documentation”. Accessed July 9, 2021.
<https://os.mbed.com/docs/mbed-os/v6.12/apis/time.html>
- [6] “<cstring> (string.h) - C++ Reference”. Accessed July 9, 2021.
<https://www.cplusplus.com/reference/cstring/>
- [7] “Smart door locks | Mbed”. Accessed July 9, 2021.
<https://os.mbed.com/built-with-mbed/smart-door-locks/>

Chapter 5

Modularization Applied to
Embedded Systems Programming

5.1 Roadmap

5.1.1 What You Will Learn

After you have studied the material in this chapter, you will be able to:

- Describe how the concept of modularity can be applied to embedded systems programming.
- Develop programs that are organized into modules and are separated into different files.
- Summarize the fundamental concepts of public and private variables and functions.

5.1.2 Review of Previous Chapters

In the preceding chapters, the smart home system was incrementally provided with a broad range of functionality. The main goal was to introduce different concepts about embedded systems programming by means of practical examples. The resultant code had hundreds of lines, as can be seen in the final example in Chapter 4. The reader may have noticed that it starts to become hard to remember which part of the program relates to which implemented function of the system. It can be even harder to find specific functionality within the code. It becomes increasingly difficult to introduce new functionality and improvements to the program.

5.1.3 Contents of This Chapter

This chapter will explain how to overcome this issue by means of *modularization*. For this purpose, the code presented in Example 4.4 is revised and the program is reorganized into different *modules*. Each resulting module will contain a piece of code that deals with a particular area of the smart home system functionality. In this way, the resultant code will be easier to understand, maintain, and improve. The original 600 lines of code will be divided into a set of files, each having precise functionality and a correspondingly smaller number of lines.



NOTE: In this chapter, the reader may notice that there are no Example, Case Study, or Under the Hood sections. In addition, in this chapter there are just a few Proposed Exercises, all located at the end. This is due to the fact that this chapter is not about including new functionality in the program, but rather explaining to the reader how the code that was introduced in the previous chapters can be improved.

5.2 Basic Principles of Modularization

5.2.1 Modularity Principle

Modularity is a basic principle in engineering. The principle states that it is better to build systems from loosely coupled components, called *modules*. These modules should have well-defined functionality and must be easy to understand, reuse, and replace.

The advantages of using modularization in embedded systems programming can be summarized as follows:

- It is easier to understand a program made of independent modules (the *Maintainability* is improved).
- It is simple to reuse modules in different programs, which improves the *productivity* of the programmer.

The reader may remember that subsection 2.2.2 *Modularization of a Program in Functions* was an initial introduction to the topic of modularization. A program that was previously organized into a single long piece of code was divided into different, shorter pieces of code called functions. These functions complete one or more specific tasks and can be used in a given program one or more times.

The problem arises when a given function has a very specific task, but this task is closely related to the tasks of another function. For example, consider the following two functions:

- *Function 1*: turns the alarm on or off after reading a code the user enters using the matrix keypad
- *Function 2*: turns the alarm on or off after reading a code the user enters using the PC keyboard

Both of these functions have very specific tasks, but they are closely coupled, as both of them modify the state of the alarm. If something should be modified in the way the alarm is turned on or off, then the two functions (*Function 1* and *Function 2*) must both be modified.

The scenario changes if the same functionality is implemented in the following way:

- *Function A*: read a code that the user enters using the matrix keypad
- *Function B*: read a code that the user enters using the PC keyboard
- *Function C*: gets a code that the user enters using either the matrix keypad (*Function A*) or the PC keyboard (*Function B*) and, depending on the code, turns the alarm on or off.

Here it can be seen that *Functions A and B* are loosely coupled to one other, and that only *Function C* deals with turning off the alarm. This increases software maintainability, which means that the code is easier for different programmers to understand, repair, and enhance over time.

In this context, two very important concepts in computer programming arise, namely *Cohesion* and *Coupling*.



DEFINITION: *Cohesion* refers to the degree to which the elements inside a module belong together, while *Coupling* is a measure of how closely connected two routines or modules are.

This whole chapter is about taking the code from Example 4.4 as the starting point and improving it by means of applying modularization techniques. This is because, for pedagogical reasons, the code used in Example 4.4 was gradually developed through the preceding chapters and is, therefore, not properly modularized.

The process of restructuring existing code – changing the factoring without changing its external behavior – is called *code refactoring*. In the following section, the code used in Example 4.4 will be refactored, as a first step to increase its modularity, by preparing modules with appropriate cohesion and coupling.

5.3 Applying Modularization to the Program Code of the Smart Home System

5.3.1 Refactoring the Program Code of the Smart Home System

As was discussed in the previous section, the code of the proposed solution introduced in Example 4.4 does not follow the principles of modularity. For example, the function *uartTask()* has approximately one hundred and fifty lines and deals with a broad range of functionality:

- Gets and processes the keys that are pressed on the PC keyboard
- Sends the messages that are displayed to the user on the PC
- Turns on/off several LEDs depending on the entered code
- Shows the present temperature read by the LM35 sensor
- Configures the new code used to deactivate the alarm
- Sets the date and time of the real-time clock
- Displays the list of events stored in memory.

The function *uartTask()* is only one example; there are many other functions in the code used in Example 4.4 that do not follow the principles of modularity:

- *alarmActivationUpdate()*: gets the reading of the sensors and also turns on/off the Alarm LED.
- *alarmDeactivationUpdate()*: assesses the entered code and also turns on/off the Incorrect code and System blocked LEDs.

Conversely, *availableCommands()* can be mentioned as an example of a function that does follow the principles of modularity; this function only displays messages to the user on the PC. It has well-defined functionality and is easy to understand, reuse, and replace.

In order to modularize the functionality of the smart home system, the core functionality can initially be grouped into the modules shown in Figure 5.1. The colors and the layout used in Figure 5.1 are intentionally the same as in Figure 1.2 so as to stress that the current smart home system functionality is very similar to the smart home system proposed in Chapter 1. The proposed improvement (i.e., modularize the code) must be done without interfering with the current smart home system functionality.

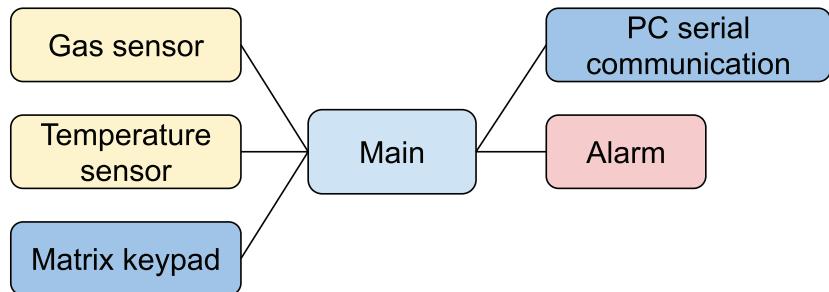


Figure 5.1 Diagram of the first attempt to modularize the smart home system program.

It is important to note that Figure 5.1 represents a first approach to the *software* modules, while Figure 1.2 represents the *hardware* modules. Very often a software module is directly related to a given hardware module, but this is not always the case. A hardware module might not have a software module related to it (because it is not controlled by a microcontroller), or it may have more than one software module related to it.

Another important concept that was mentioned previously is that it is desirable to write the modules in such a way that they can be reused in other projects. Therefore, it is convenient to have a *main()* function as small and simple as possible, its role being to call a few functions that in turn trigger all the functionality of the system.

Finally, the modules should be organized with consideration of future improvements and functionality that could be added to the system. This is a strong reason to group the system functionalities into small modules, with all modules having a well-defined functionality.

Considering all these concepts, the proposed modules for the smart home system implementation are shown in Figure 5.2. The *main.cpp* file is not a module. However, it is included in Figure 5.2 to stress that it only uses functions provided by the smart home system module. In turn, to implement the system functionality, the smart home system module will call functions from other modules. The functionality of each module is briefly described in Table 5.1. In the table, the role of each module is classified as *system*, *subsystem*, or *driver*.

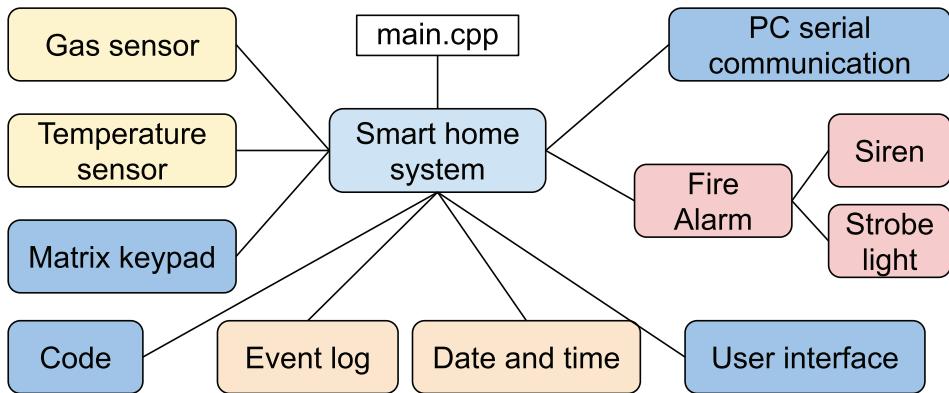


Figure 5.2 Diagram of the modules used in the smart home system program.

Table 5.1 Functionalities and roles of the smart home system modules.

Module	Description of its functionality	Role
Smart home system	Manages the functionality of all the subsystems.	System
Fire alarm	Controls the siren and the strobe light.	Subsystem
Code	Assesses entered codes and manages setting of new codes.	Subsystem
User interface	Manages the user interface (LEDs and matrix keypad).	Subsystem
PC serial com	Manages the communication with the PC (send/receive data).	Subsystem
Event log	Logs the system events.	Subsystem
Siren	Drives the siren (implemented by a buzzer).	Driver
Strobe light	Drives the strobe light (implemented by an LED).	Driver
Gas sensor	Reads the MQ-2 gas sensor.	Driver
Temperature sensor	Reads the LM35 temperature sensor.	Driver
Matrix keypad	Reads the keys pressed on the matrix keypad.	Driver
Date and time	Reads and writes the date and time of the real-time clock.	Driver

In Tables 5.2 to 5.13, the functions provided by each of the proposed modules shown in Table 5.1 are detailed, together with a brief description of their functionality and the modules (or *main.cpp* file) that make use of each function. The tables are presented following the order used in Table 5.1. The code will be shown and discussed in detail in subsection 5.3.2.

It is important to note that in Tables 5.2 to 5.13, there is a bold line to separate functions that are called by other modules from functions that are used only within the module itself. For example, in Table 5.3, the first four functions (*fireAlarmsInit()* to *overTemperatureDetectorStateRead()*) are

used by functions that belong to other modules, while the last six functions (from `gasDetectedRead()` to `fireAlarmStrobeTime()`) are used only by functions that belong within the fire alarm module itself. There is no reason to grant other modules access to the functions `gasDetectedRead()`, `overTemperatureDetectedRead()`, `fireAlarmActivationUpdate()`, `fireAlarmDeactivationUpdate()`, `fireAlarmDeactivate()`, and `fireAlarmStrobeTime()`. The terminology and concepts related to this differentiation are discussed in this chapter.

Table 5.2 Functions of the smart home system module.

Name of the function	Description of its functionality	File that uses it
smartHomeSystemInit()	Initializes the subsystems and drivers of the smart home system.	main.cpp
smartHomeSystemUpdate()	Calls the functions that update the modules Fire alarm, User interface, PC serial com, and Event log. Also manages the system timing.	main.cpp

Table 5.3 Functions of the fire alarm module.

Name of the function	Description of its functionality	Modules that use it
fireAlarmInit()	Initializes the fire alarm subsystem by calling <code>temperatureSensorInit()</code> , <code>gasSensorInit()</code> , <code>sirenInit()</code> , and <code>strobeLightInit()</code> .	Smart home system
fireAlarmUpdate()	Updates the fire alarm subsystem by calling <code>fireAlarmActivationUpdate()</code> , <code>fireAlarmDeactivationUpdate()</code> , <code>sirenUpdate()</code> , and <code>strobeLightUpdate()</code> .	Smart home system
gasDetectorStateRead()	Returns the current state of the gas detector.	Event log PC serial com
overTemperatureDetectorStateRead()	Returns the current state of the over temperature detector.	Event log PC serial com
gasDetectedRead()	Returns true if gas has been detected.	Fire alarm
overTemperatureDetectedRead()	Returns true if over temperature has been detected.	Fire alarm
fireAlarmActivationUpdate()	Controls the activation of the siren and the strobe light.	Fire alarm
fireAlarmDeactivationUpdate()	Controls the deactivation of the siren and the strobe light.	Fire alarm
fireAlarmDeactivate()	Implements the deactivation of the siren and the strobe light.	Fire alarm
fireAlarmStrobeTime()	Controls the siren and strobe light time.	Fire alarm

Table 5.4 Functions of the code module.

Name of the function	Description of its functionality	Modules that use it
codeWrite()	Writes the new code set by the user into <code>CodeSequence[]</code> .	PC serial com
codeMatchFrom()	Checks if a new code is entered and assesses the code.	Fire alarm
codeMatch()	Returns a Boolean indicating if the code is correct.	Code
codeDeactivate()	Sets <code>systemBlockedState</code> and <code>incorrectCodeState</code> to OFF.	Code

Table 5.5 Functions of the user interface module.

Name of the function	Description of its functionality	Modules that use it
userInterfaceInit()	Sets systemBlockedState and incorrectCodeState to OFF and calls matrixKeypadInit().	Smart home system
userInterfaceUpdate()	Calls incorrectCodeIndicatorUpdate(), userInterfaceMatrixKeypadUpdate(), and systemBlockedIndicatorUpdate().	Smart home system
userInterfaceCodeCompleteRead()	Returns a Boolean indicating if the read of the code is complete or not.	Code
userInterfaceCodeCompleteWrite()	Sets the state of codeComplete.	Code
incorrectCodeStateRead()	Returns a Boolean indicating if the code is correct or incorrect.	Event log User interface
incorrectCodeStateWrite()	Sets the state of incorrectCodeState.	Code
systemBlockedStateRead()	Returns a Boolean indicating if the system is blocked or not.	Event log User interface
systemBlockedStateWrite()	Sets the state of systemBlockedState.	Code
incorrectCodeIndicatorUpdate()	Controls the object incorrectCodeLed.	User interface
systemBlockedIndicatorUpdate()	Controls the object systemBlockedLed.	User Interface
userInterfaceMatrixKeypadUpdate()	Gets a new code using the matrix keypad and manages incorrectCodeState.	User interface

Table 5.6 Functions of the PC serial communication module.

Name of the function	Description of its functionality	Modules that use it
pcSerialComInit()	Calls the function availableCommands().	Smart home system
pcSerialComStringWrite()	Writes a string to the PC serial port.	Code Event log
pcSerialComUpdate()	Gets the commands and the entered codes.	Smart home system
pcSerialComCodeCompleteRead()	Returns a Boolean variable indicating if the code read from the PcSerialCom is complete.	Code
pcSerialComCodeCompleteWrite()	Writes the state of codeCompleteFromPcSerialCom.	Code
pcSerialComCharRead()	Returns a char corresponding to a read value.	PC serial com
pcSerialComStringRead()	Reads a string from the PC serial port.	PC serial com
pcSerialComSaveCodeUpdate()	Receives a new char of the entered code.	PC serial com
pcSerialComSaveNewCodeUpdate()	Receives a new char of the new code.	PC serial com
pcSerialComCommandUpdate()	Depending on the entered char, triggers one of the commands of the PC serial com module.	PC serial com
availableCommands()	Displays the available commands on the PC.	PC serial com

Name of the function	Description of its functionality	Modules that use it
commandShowCurrent AlarmState()	Displays whether the alarm is activated or not on the PC.	PC serial com
commandShowCurrent GasDetectorState()	Displays whether the gas is being detected or not on the PC.	PC serial com
commandShowCurrent OverTempDetectorState()	Displays whether the temperature is above the maximum level or not on the PC.	PC serial com
commandEnterCode Sequence()	Configures the UART to receive a code and displays a message on the PC asking for the code.	PC serial com
commandEnterNewCode()	Configures the UART to receive a code and displays a message on the PC asking for the new code.	PC serial com
commandShowCurrent TemperatureInCelsius()	Displays the temperature in Celsius on the PC.	PC serial com
commandShowCurrent TemperatureInFahrenheit()	Displays the temperature in Fahrenheit on the PC.	PC serial com
commandSetDateAndTime()	Gets date and time and writes it to the RTC.	PC serial com
commandShowDateAndTime()	Displays the date and time on the PC.	PC serial com
commandShowStoredEvents()	Displays the stored events on the PC.	PC serial com

Table 5.7 Main functionality of the event log module.

Name of the function	Description of its functionality	Modules that use it
eventLogUpdate()	Updates the log of events.	Smart home system
eventLogNumberOfStoredEvents()	Returns the number of stored events.	PC serial com
eventLogRead()	Reads an event stored in the log.	PC serial com
eventLogWrite()	Stores an event in the log.	Event log
eventLogElementStateUpdate()	Stores the new events in the log.	Event log

Table 5.8 Functions of the siren module.

Name of the function	Description of its functionality	Modules that use it
sirenInit()	Sets the siren to OFF (OpenDrain).	Fire alarm
sirenUpdate()	Updates the state of the siren.	Fire alarm
sirenStateRead()	Returns the Boolean variable sirenState.	Event log Fire alarm PC serial com User interface
sirenStateWrite()	Writes the state of Boolean variable sirenState.	Fire alarm

Table 5.9 Functions of the strobe light module.

Name of the function	Description of its functionality	Modules that use it
strobeLightInit()	Sets alarmLed to OFF.	Fire alarm
strobeLightUpdate()	Updates the state of the strobe light.	Fire alarm
strobeLightStateRead()	Returns the Boolean variable strobeLightState.	Fire alarm User interface
strobeLightStateWrite()	Writes the state of Boolean variable strobeLightState.	Fire alarm

Table 5.10 Functions of the gas sensor module.

Name of the function	Description of its functionality	Modules that use it
gasSensorInit()	Has no functionality. Reserved for future use.	Fire alarm
gasSensorUpdate()	Has no functionality. Reserved for future use.	Fire alarm
gasSensorRead()	Returns the reading of the gas sensor detector.	Fire alarm

Table 5.11 Functions of the temperature sensor module.

Name of the function	Description of its functionality	Modules that use it
temperatureSensorInit()	Has no functionality. Reserved for future use.	Fire alarm
temperatureSensorUpdate()	Updates the temperature reading.	Fire alarm
temperatureSensorReadCelsius()	Returns the temperature in °C.	Fire alarm
temperatureSensorReadFahrenheit()	Returns the temperature in °F.	PC serial com
celsiusToFahrenheit()	Converts a reading in °C to °F.	Temperature sensor
analogReadingScaledWithTheLM35Formula()	Converts an LM35 reading to temp.	Temperature sensor

Table 5.12 Main functionality of the matrix keypad module.

Name of the function	Description of its functionality	Modules that use it
matrixKeypadInit()	Initializes the matrix keypad pins and FSM.	User interface
matrixKeypadUpdate()	Implements the matrix keypad FSM.	User interface
matrixKeypadScan()	Scans the matrix keypad and returns the read char.	Matrix keypad
matrixKeypadReset()	Resets the matrix keypad FSM.	Matrix keypad

Table 5.13 Functions of the date and time module.

Name of the function	Description of its functionality	Modules that use it
dateAndTimeRead()	Returns the RTC date and time.	PC serial com
dateAndTimeWrite()	Writes the RTC date and time.	PC serial com

Looking at Tables 5.2 to 5.13, it can be appreciated how the different modules are related to each other. These relationships are summarized in Figure 5.3.

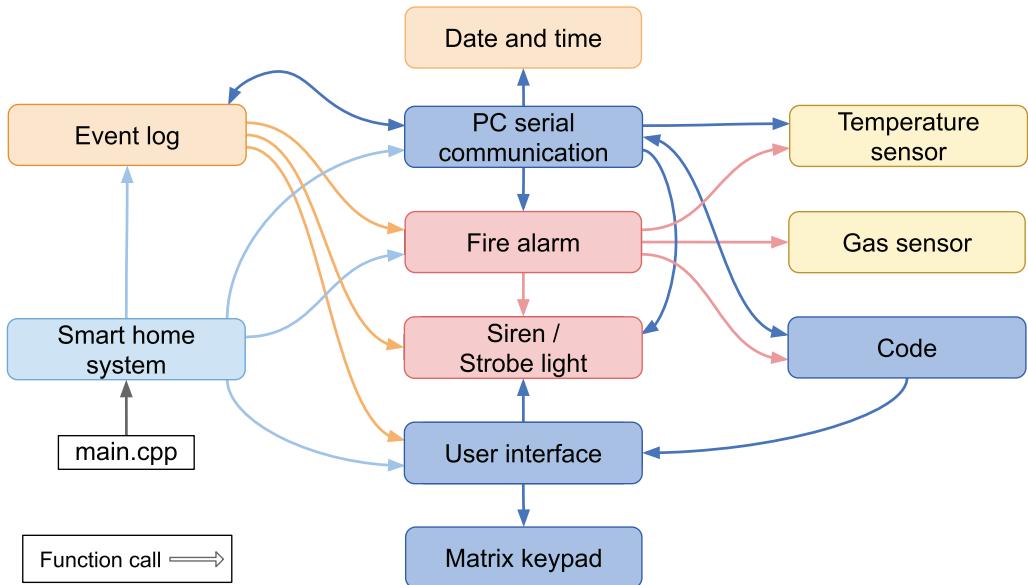


Figure 5.3 Diagram of the relationships between the modules used in the smart home system.



NOTE: For the sake of simplicity, the Siren and Strobe light modules are drawn together in Figure 5.3 despite being independent modules. In section 5.3.2, it will be shown that they have many similarities.

As was briefly described in Table 5.1, the roles of the modules indicated in Figure 5.3 are:

- The Smart home system is the only module that is called from the *main.cpp* (its role is *system*).
- The Fire alarm, Code, User interface, PC serial communication, and Event log are *subsystem* modules. They are called by other modules and call functions from other modules.
- The Siren, Strobe light, Gas sensor, Temperature sensor, Matrix keypad, and Date and time are *driver* modules. They are called by other modules, but they do not call functions from other modules.

5.3.2 Detailed Implementation of the Refactored Code of the Smart Home System

This subsection shows the program code that is proposed in order to implement the functions introduced in Tables 5.2 to 5.13. The whole program can be imported using the URL available in [1].

The libraries and defines that are used are shown in Code 5.1. It can be seen that the #defines are now organized according to the different modules that were defined. Some names have been changed in comparison with Example 4.4 (for example, OVER_TEMP_LEVEL was changed to TEMPERATURE_C_LIMIT_ALARM, KEYPAD_NUMBER_OF_ROWS was changed to MATRIX_KEYPAD_NUMBER_OF_ROWS), and some prefixes have been incorporated, such as "CODE_", "EVENT_LOG_", and "SIREN_".

```

1 //=====[Libraries]=====
2 #include "mbed.h"
3 #include "arm_book_lib.h"
4
5 //=====[Defines]=====
6
7 // Module: code -----
8
9 #define CODE_NUMBER_OF_KEYS    4
10
11 // Module: event_log -----
12
13 #define EVENT_LOG_MAX_STORAGE      20
14 #define EVENT_HEAD_STR_LENGTH      8
15 #define EVENT_LOG_NAME_MAX_LENGTH  13
16 #define DATE_AND_TIME_STR_LENGTH   18
17 #define CTIME_STR_LENGTH          25
18 #define NEW_LINE_STR_LENGTH        3
19 #define EVENT_STR_LENGTH           \
20     (EVENT_HEAD_STR_LENGTH + \
21      EVENT_LOG_NAME_MAX_LENGTH + \
22      DATE_AND_TIME_STR_LENGTH + \
23      CTIME_STR_LENGTH + \
24      NEW_LINE_STR_LENGTH)
25
26 // Module: fire_alarm -----
27
28 #define TEMPERATURE_C_LIMIT_ALARM 50.0
29 #define STROBE_TIME_GAS           1000
30 #define STROBE_TIME_OVER_TEMP     500
31 #define STROBE_TIME_GAS_AND_OVER_TEMP 100
32
33 // Module: matrix_keypad -----
34
35 #define MATRIX_KEYPAD_NUMBER_OF_ROWS 4
36 #define MATRIX_KEYPAD_NUMBER_OF_COLS 4
37 #define DEBOUNCE_BUTTON_TIME_MS    40
38
39 // Module: smart_home_system -----
40
41 #define SYSTEM_TIME_INCREMENT_MS 10
42
43 // Module: temperature_sensor -----
44
45 #define LM35_NUMBER_OF_AVG_SAMPLES 100

```

Code 5.1 Libraries and defines used in the refactored version of the smart home system code.



NOTE: From now on, the names of modules will be written following the snake_case format shown on lines 12, 26, 33, 39, and 43 of Code 5.1. In this format, each space is replaced by an underscore (_) character, and the first letter of each word is written in lowercase.

Many definitions were added to the `event_log` module, as can be seen between lines 14 and 24. Figure 5.4 shows a diagram that illustrates the rationale behind the number of characters assigned to each of these definitions, as it can be obtained after analyzing the implementation of `eventLogRead()`, which is introduced later in this chapter. `EVENT_STR_LENGTH`, which is obtained as the sum of the other values (line 20 to line 24), will be used in the function `commandShowStoredEvents()`, as discussed below.

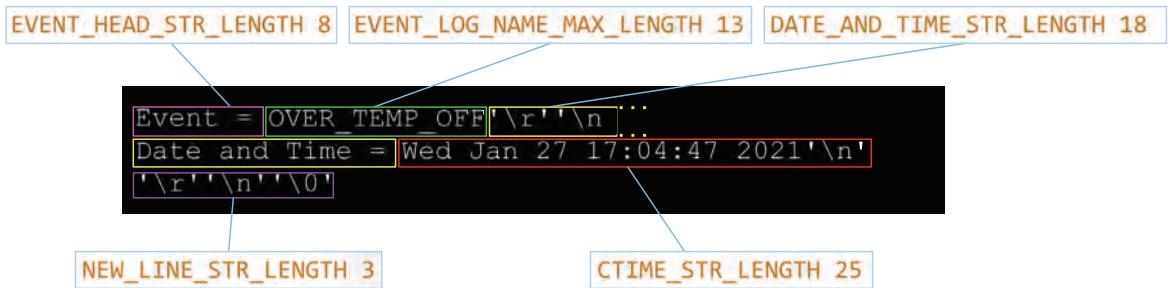


Figure 5.4 Diagram showing how the definitions of the event_log module were made.



NOTE: Remember that the characters '\0' (null), '\r' (carriage return), and '\n' (line feed) are not printed on the serial terminal, but still occupy one position in an array of char. '\0' is stored with the code 0, '\r' with the code 10, and '\n' with the code 13.

In Code 5.2, the declarations of public data types are shown. The first data type that is declared, `codeOrigin_t`, as well as the last data type that is declared, `pcSerialComMode_t`, are introduced in this chapter with the aim of modularizing the program code, as discussed below. All of the remaining data types are the same as in Example 4.4, but they are now grouped with regard to the respective modules where they are used.

```

1 //===== [Declaration of public data types] =====
2
3 // Module: code -----
4
5 typedef enum{
6     CODE_KEYPAD,
7     CODE_PC_SERIAL,
8 } codeOrigin_t;
9
10 // Module: event_log -----
11
12 typedef struct systemEvent {
13     time_t seconds;
14     char typeOfEvent[EVENT_LOG_NAME_MAX_LENGTH];
15 } systemEvent_t;
16
17 // Module: matrix_keypad -----
18
19 typedef enum {
20     MATRIX_KEYPAD_SCANNING,
21     MATRIX_KEYPAD_DEBOUNCE,
22     MATRIX_KEYPAD_KEY_HOLD_PRESSED
23 } matrixKeypadState_t;
24
25 // Module: pc_serial_com -----
26
27 typedef enum{
28     PC_SERIAL_COMMANDS,
29     PC_SERIAL_GET_CODE,
30     PC_SERIAL_SAVE_NEW_CODE,
31 } pcSerialComMode_t;

```

Code 5.2 Public data types of the refactored version of the smart home system code.

In Code 5.3, the declarations and initializations of public global objects are shown. Again, the objects are the same as in Example 4.4, although they have been subdivided into modules. The only object that is declared in a different way is *sirenPin* (PE_10). This object was introduced in Example 3.5 and declared in Chapters 3 and 4 as a *DigitalInOut* object to turn on the buzzer by asserting 0 V in PE_10 and turn off the buzzer by configuring PE_10 as an open drain input.

In Code 5.3, it can be seen that PE_10 is now declared as a *DigitalOut* object. Using this method, the buzzer can be turned on by asserting 0 V to PE_10 and turned off by asserting 3.3 V to PE_10. From now on, when the alarm is activated, the buzzer will be turned off and on intermittently, concurrently with the LD1 *alarmLed*. The code to implement this behavior is simpler if PE_10 is declared as a *DigitalOut* object. In this way, the buzzer will now generate a “beep beep” sound instead of the continuous sound that was implemented in previous chapters.

```

1 //=====[Declaration and initialization of public global objects]=====
2
3 // Module: fire_alarm -----
4
5 DigitalIn alarmTestButton(BUTTON1);
6
7 // Module: gas_sensor -----
8
9 DigitalIn mq2(PE_12);
10
11 // Module: matrix_keypad -----
12
13 DigitalOut keypadRowPins[MATRIX_KEYPAD_NUMBER_OF_ROWS] = {PB_3, PB_5, PC_7, PA_15};
14 DigitalIn keypadColPins[MATRIX_KEYPAD_NUMBER_OF_COLS] = {PB_12, PB_13, PB_15, PC_6};
15
16 // Module: pc_serial_com -----
17
18 Serial uartUsb(USBTX, USBRX, 115200);
19
20 // Module: siren -----
21
22 DigitalOut sirenPin(PE_10);
23
24 // Module: strobe_light -----
25
26 DigitalOut strobeLight(LED1);
27
28 // Module: temperature_sensor -----
29
30 AnalogIn lm35(A1);
31
32 // Module: user_interface -----
33
34 DigitalOut incorrectCodeLed(LED3);
35 DigitalOut systemBlockedLed(LED2);

```

Code 5.3 Public global objects of the refactored version of the smart home system code.



NOTE: When *sirenPin* is defined as a *DigitalOut* object (line 22 of Code 5.3), the buzzer does not turn off completely when PE_10 is configured in HIGH state but makes a very soft sound. This is due to the fact that when 3.3 V is applied to PE_10, there is 1.7 V between the “+” and “-” buzzer pins, as shown in Figure 5.5. In order to completely turn off the buzzer while using PE_10 as a *DigitalOut* object, the circuit shown in Figure 5.6 can be used. In this circuit, PE_10 activates or deactivates transistor Q1, which allows current to circulate through the buzzer, just like a switch. RB1 is used to limit the base current.

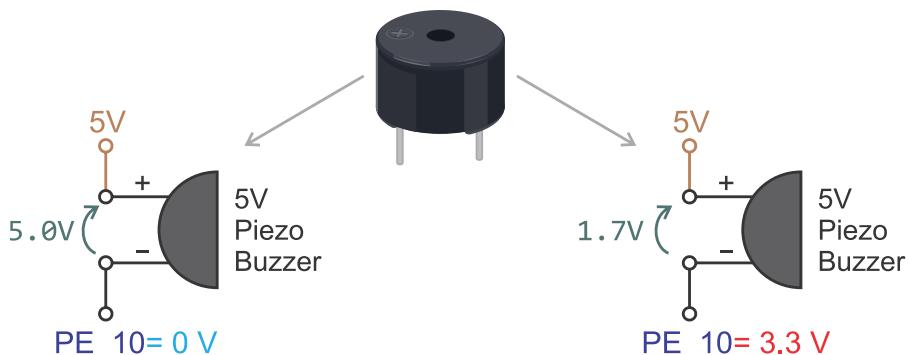


Figure 5.5. Diagram of the voltage through the buzzer pins when PE_10 is set to 0 V and 3.3 V.

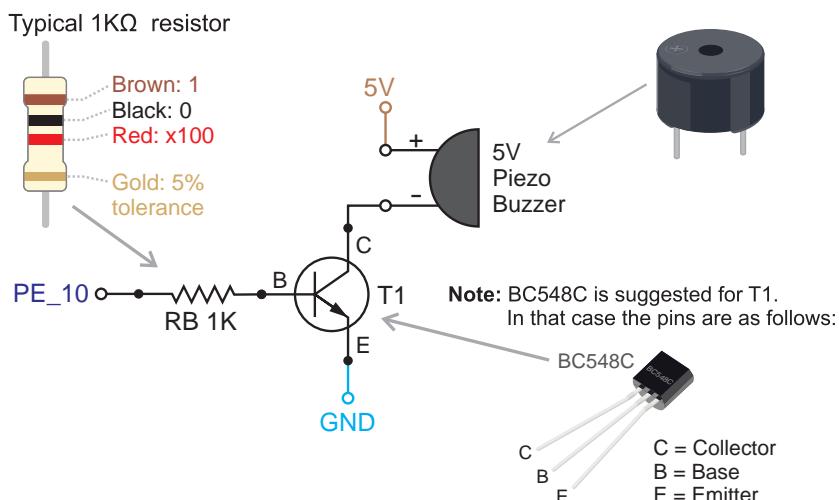


Figure 5.6 Diagram of the circuit that can be used to completely turn on and off the buzzer.



NOTE: In the implementation shown in Figure 5.5, the buzzer is turned on when 0 V is asserted in PE_10 and is turned off when 3.3 V is asserted in PE_10. However, in the circuit shown in Figure 5.6, the buzzer is turned on when 3.3 V is asserted in PE_10 and is turned off when 0 V is asserted in PE_10. As a consequence, the code should be modified if the latter circuit is used. This is discussed below.

In Code 5.4, the declaration and initialization of public global variables is shown. In this new implementation, the variables *gasDetected* and *overTemperatureDetected* are used to indicate that gas and over temperature have been detected, respectively, while *gasDetectorState* and *overTemperatureDetectorState* are used to indicate, respectively, that gas and over temperature are currently being detected.

The reader should note that the new variables *systemBlockedState* and *incorrectCodeState* are used to indicate the state of the LEDs, in order to differentiate them from the LEDs themselves. These are managed by means of the objects *systemBlockedLed* and *incorrectCodeLed*. This is a consequence of the modularization: the state of *systemBlockedState* and *incorrectCodeState* are controlled by the *code* module, while the LEDs are controlled by the *user_interface* module.

The reader should also note that there are many new variables declared, for example in the *pc_serial_com* and *siren* modules. The use of these variables will be explained in the following pages, together with the functions of the modules.

```

1 //===== [Declaration and initialization of public global variables] ======
2
3 // Module: code -----
4
5 int numberOfIncorrectCodes = 0;
6 char codeSequence[CODE_NUMBER_OF_KEYS] = { '1', '8', '0', '5' };
7
8 // Module: event_log -----
9
10 bool sirenLastState = OFF;
11 bool gasLastState = OFF;
12 bool tempLastState = OFF;
13 bool ICLastState = OFF;
14 bool SBLastState = OFF;
15 int eventsIndex = 0;
16 systemEvent_t arrayOfStoredEvents[EVENT_LOG_MAX_STORAGE];
17
18 // Module: fire_alarm -----
19
20 bool gasDetected = OFF;
21 bool overTemperatureDetected = OFF;
22 bool gasDetectorState = OFF;
23 bool overTemperatureDetectorState = OFF;
24
25 // Module: matrix_keypad -----
26
27 matrixKeypadState_t matrixKeypadState;
28 int timeIncrement_ms = 0;
29 int accumulatedDebounceMatrixKeypadTime = 0;
30 char matrixKeypadLastKeyPressed = '\0';
31
32 // Module: pc_serial_com -----
33
34 char codeSequenceFromPcSerialCom[CODE_NUMBER_OF_KEYS];
35 pcSerialComMode_t pcSerialComMode = PC_SERIAL_COMMANDS;
36 bool codeCompleteFromPcSerialCom = false;
37 int numberOfCodeCharsFromPcSerialCom = 0;
38 char newCodeSequence[CODE_NUMBER_OF_KEYS];
39

```

```

40 // Module: siren -----
41
42 bool sirenState = OFF;
43 int accumulatedTimeAlarm = 0;
44
45 // Module: strobe_light -----
46
47 bool strobeLightState = OFF;
48
49 // Module: temperature_sensor -----
50
51 float lm35TemperatureC = 0.0;
52 float lm35ReadingsArray[LM35_NUMBER_OF_AVG_SAMPLES];
53 int lm35SampleIndex = 0;
54
55 // Module: user_interface -----
56
57 char codeSequenceFromUserInterface[CODE_NUMBER_OF_KEYS];
58 bool incorrectCodeState = OFF;
59 bool systemBlockedState = OFF;
60 bool codeComplete = false;
61 int numberOfCodeChars = 0;
62 int numberOfHashKeyReleased = 0;

```

Code 5.4 Public global variables of the refactored version of the smart home system code.

In Code 5.5 and Code 5.6, the declarations of the public functions are shown. All these functions will be discussed one after the other in the following pages.

```

1 //=====[Declarations (prototypes) of public functions]=====
2
3 // Module: code -----
4
5 void codeWrite( char* newCodeSequence );
6 bool codeMatchFrom( codeOrigin_t codeOrigin );
7 bool codeMatch( char* codeToCompare );
8 void codeDeactivate();
9
10 // Module: date_and_time -----
11
12 char* dateAndTimeRead();
13 void dateAndTimeWrite( int year, int month, int day,
14                         int hour, int minute, int second );
15
16 // Module: event_log -----
17
18 void eventLogUpdate();
19 int eventLogNumberOfStoredEvents();
20 void eventLogRead( int index, char* str );
21 void eventLogWrite( bool currentState, const char* elementName );
22 void eventLogElementStateUpdate( bool lastState,
23                                 bool currentState,
24                                 const char* elementName );
25
26 // Module: fire_alarm -----
27
28 void fireAlarmInit();
29 void fireAlarmUpdate();

```

```

30 bool gasDetectorStateRead();
31 bool overTemperatureDetectorStateRead();
32 bool gasDetectedRead();
33 bool overTemperatureDetectedRead();
34 void fireAlarmActivationUpdate();
35 void fireAlarmDeactivationUpdate();
36 void fireAlarmDeactivate();
37 int fireAlarmStrobeTime();
38
39 // Module: gas_sensor -----
40
41 void gasSensorInit();
42 void gasSensorUpdate();
43 bool gasSensorRead();
44
45 // Module: matrix_keypad -----
46
47 void matrixKeypadInit( int updateTime_ms );
48 char matrixKeypadUpdate();
49 char matrixKeypadScan();
50 void matrixKeypadReset();

```

Code 5.5 Declarations of public functions of the refactored version of the smart home system code (Part 1/2).

```

1 // Module: pc_serial_com -----
2
3 void pcSerialComInit();
4 char pcSerialComCharRead();
5 void pcSerialComStringWrite( const char* str );
6 void pcSerialComStringRead( char* str, int strLength );
7 void pcSerialComUpdate();
8 bool pcSerialComCodeCompleteRead();
9 void pcSerialComCodeCompleteWrite( bool state );
10 void pcSerialComGetCodeUpdate( char receivedChar );
11 void pcSerialComSaveNewCodeUpdate( char receivedChar );
12 void pcSerialComCommandUpdate( char receivedChar );
13 void availableCommands();
14 void commandShowCurrentAlarmState();
15 void commandShowCurrentGasDetectorState();
16 void commandShowCurrentOverTemperatureDetectorState();
17 void commandEnterCodeSequence();
18 void commandEnterNewCode();
19 void commandShowCurrentTemperatureInCelsius();
20 void commandShowCurrentTemperatureInFahrenheit();
21 void commandSetDateAndTime();
22 void commandShowDateAndTime();
23 void commandShowStoredEvents();
24
25 // Module: siren -----
26
27 void sirenInit();
28 bool sirenStateRead();
29 void sirenStateWrite( bool state );
30 void sirenUpdate( int strobeTime );
31
32 // Module: strobe_light -----
33
34 void strobeLightInit();
35 bool strobeLightStateRead();

```

```

36 void strobeLightStateWrite( bool state );
37 void strobeLightUpdate( int strobeTime );
38
39 // Module: smart_home_system -----
40
41 void smartHomeSystemInit();
42 void smartHomeSystemUpdate();
43
44 // Module: temperature_sensor -----
45
46 void temperatureSensorInit();
47 void temperatureSensorUpdate();
48 float temperatureSensorReadCelsius();
49 float temperatureSensorReadFahrenheit();
50 float celsiusToFahrenheit( float tempInCelsiusDegrees );
51 float analogReadingScaledWithTheLM35Formula( float analogReading );
52
53 // Module: user_interface -----
54
55 void userInterfaceInit();
56 void userInterfaceUpdate();
57 bool userInterfaceCodeCompleteRead();
58 void userInterfaceCodeCompleteWrite( bool state );
59 bool incorrectCodeStateRead();
60 void incorrectCodeStateWrite( bool state );
61 void incorrectCodeIndicatorUpdate();
62 bool systemBlockedStateRead();
63 void systemBlockedStateWrite( bool state );
64 void systemBlockedIndicatorUpdate();
65 void userInterfaceMatrixKeypadUpdate();

```

Code 5.6 Declarations of public functions of the refactored version of the smart home system code (part 2/2).

The main function of the refactored version of the smart home system code is shown in Code 5.7. Just one function is used to initialize the system (`smartHomeSystemInit()`), and only one function is used to update the system (`smartHomeSystemUpdate()`).

```

1 //=====[Main function, the program entry point after power on or reset]=====
2
3 int main()
4 {
5     smartHomeSystemInit();
6     while (true) {
7         smartHomeSystemUpdate();
8     }
9 }

```

Code 5.7 The main function of the refactored version of the smart home system code.

In Code 5.8, the implementations of some of the functions of the `code` module are shown. The function `codeWrite()` receives as a parameter a pointer to the new code set by the user (recall from Chapter 4 that a pointer is a variable that stores a *memory address*, usually corresponding to another variable) and writes the new code into `CodeSequence[]`. The implementation of this functionality was previously in case '5' of the switch of the `uartTask()` function. In this new implementation, the storage of the code is detached from the functionality of sending the code through the UART. This improves the responsiveness of the system (e.g., in the previous implementation the system remained waiting for the four digits, which impacted the blinking of the LEDs).

The function `codeMatch()` receives as a parameter a pointer to the code to compare (i.e., the memory address where `codeToCompare` is stored) and returns a Boolean indicating whether the code is correct. This functionality was previously implemented by the function `areEqual()` for the control panel and directly coded in case '4' of the `uartTask()` function for the serial communication. By means of this new function, the functionality of assessing the code is unified for both the code entered using the PC and the code entered using the matrix keypad.

```

1 //===== [Implementations of public functions] =====
2
3 // Module: code -----
4
5 void codeWrite( char* newCodeSequence )
6 {
7     int i;
8     for (i = 0; i < CODE_NUMBER_OF_KEYS; i++) {
9         codeSequence[i] = newCodeSequence[i];
10    }
11 }
12
13 bool codeMatch( char* codeToCompare )
14 {
15     int i;
16     for (i = 0; i < CODE_NUMBER_OF_KEYS; i++) {
17         if ( codeSequence[i] != codeToCompare[i] ) {
18             return false;
19         }
20     }
21     return true;
22 }
```

Code 5.8 Implementation of the functions of the code module (Part 1/2).

In Code 5.9, the implementations of the remaining functions of the `code` module are shown. `codeMatchFrom()` checks if a new code is entered, assesses the code, and, if it is correct, deactivates the alarm. This function receives the parameter `codeOrigin` of type `codeOrigin_t`, as introduced in Code 5.2, to indicate where the code came from. It is called by `fireAlarmDeactivationUpdate()` and compares two codes. By using the functions `pcSerialComCodeCompleteRead()` and `userInterfaceCodeCompleteRead()`, it checks if there is a new code to call `codeMatch()` to assess if the code is correct.

On line 8 of Code 5.9, it can be seen that `userInterfaceCodeCompleteWrite()` is used to set the variable `codeCompleteFromUserInterface` to false, while on line 21 `pcSerialComCodeCompleteWrite()` is used to set `codeCompleteFromPcSerialCom` to false. If the code entered by means of the `user_interface` module or the `pc_serial_com` module is correct, the function `codeDeactivate()` is called (lines 10 and 23) in order to turn off the `systemBlockedState` and the `incorrectCodeState` (lines 46 and 47) as well as setting the variable `numberOfIncorrectCodes` (line 48) to zero. If the entered code is incorrect, then `incorrectCodeState` is set to on, and `numberOfIncorrectCodes` is incremented (lines 12 and 13 and lines 26 and 27).

```

1  bool codeMatchFrom( codeOrigin_t codeOrigin )
2  {
3      bool codeIsCorrect = false;
4      switch (codeOrigin) {
5          case CODE_KEYPAD:
6              if( userInterfaceCodeCompleteRead() ) {
7                  codeIsCorrect = codeMatch(codeSequenceFromUserInterface);
8                  userInterfaceCodeCompleteWrite(false);
9                  if ( codeIsCorrect ) {
10                      codeDeactivate();
11                  } else {
12                      incorrectCodeStateWrite(ON);
13                      numberOfIncorrectCodes++;
14                  }
15              }
16              break;
17
18          case CODE_PC_SERIAL:
19              if( pcSerialComCodeCompleteRead() ) {
20                  codeIsCorrect = codeMatch(codeSequenceFromPcSerialCom);
21                  pcSerialComCodeCompleteWrite(false);
22                  if ( codeIsCorrect ) {
23                      codeDeactivate();
24                      pcSerialComStringWrite( "\r\nThe code is correct\r\n\r\n" );
25                  } else {
26                      incorrectCodeStateWrite(ON);
27                      numberOfIncorrectCodes++;
28                      pcSerialComStringWrite( "\r\nThe code is incorrect\r\n\r\n" );
29                  }
30              }
31              break;
32
33          default:
34              break;
35      }
36
37      if ( numberOfIncorrectCodes >= 5 ) {
38          systemBlockedStateWrite(ON);
39      }
40
41      return codeIsCorrect;
42  }
43
44  void codeDeactivate()
45  {
46      systemBlockedStateWrite(OFF);
47      incorrectCodeStateWrite(OFF);
48      numberOfIncorrectCodes = 0;
49  }

```

Code 5.9 Implementation of the functions of the code module (part 2/2).

In Code 5.10, the implementation of the functions of the module *date_and_time* is shown. The function *dateAndTimeRead()* reads the date and time from the RTC, while the function *dateAndTimeWrite()* configures the RTC using the date and time indicated by the received parameters. This functionality was previously implemented inside cases 's' and 't' of the function *uartTask()*.

```

1 // Module: date_and_time -----
2
3 char* dateAndTimeRead()
4 {
5     time_t epochSeconds;
6     epochSeconds = time(NULL);
7     return ctime(&epochSeconds);
8 }
9
10 void dateAndTimeWrite( int year, int month, int day,
11                         int hour, int minute, int second )
12 {
13     struct tm rtcTime;
14
15     rtcTime.tm_year = year - 1900;
16     rtcTime.tm_mon = month - 1;
17     rtcTime.tm_mday = day;
18     rtcTime.tm_hour = hour;
19     rtcTime.tm_min = minute;
20     rtcTime.tm_sec = second;
21
22     rtcTime.tm_isdst = -1;
23
24     set_time( mktime( &rtcTime ) );
25 }
```

Code 5.10 Implementation of the functions of the date_and_time module.

In Code 5.11, the implementation of the function `eventLogUpdate()` of the `event_log` module is shown. This function updates the log of events. The reader should note the usage of the variable `currentState` in each of the five parts of this function, together with calls to functions from different modules (i.e., `sirenStateRead()`, `gasDetectorStateRead()`, etc.).

```

1 // Module: event_log -----
2
3 void eventLogUpdate()
4 {
5     bool currentState = sirenStateRead();
6     eventLogElementStateUpdate( sirenLastState, currentState, "ALARM" );
7     sirenLastState = currentState;
8
9     currentState = gasDetectorStateRead();
10    eventLogElementStateUpdate( gasLastState, currentState, "GAS_DET" );
11    gasLastState = currentState;
12
13    currentState = overTemperatureDetectorStateRead();
14    eventLogElementStateUpdate( tempLastState, currentState, "OVER_TEMP" );
15    tempLastState = currentState;
16
17    currentState = incorrectCodeStateRead();
18    eventLogElementStateUpdate( ICLastState, currentState, "LED_IC" );
19    ICLastState = currentState;
20
21    currentState = systemBlockedStateRead();
22    eventLogElementStateUpdate( SBLastState, currentState, "LED_SB" );
23    SBLastState = currentState;
24 }
```

Code 5.11 Implementation of the functions of the Event log module (Part 1/2).

In Code 5.12, the new function `eventLogNumberOfStoredEvents()`, which returns the number of stored events, is shown. The function `eventLogRead()`, which reads an event stored in the log, is also shown. In Example 4.4, this functionality was implemented in case 'E' of the `uartTask()` function, and is now called by the function `commandShowStoredEvents()`, as will be seen in Code 5.20. The first parameter of `eventLogRead()` is the event index. Its second parameter, `str`, is a pointer to an array of chars. In that array, first "Event = " (line 9) is written, then the content of the corresponding type of event: `arrayOfStoredEvents[index].typeOfEvent` (line 10), then "\r\nDate and Time = " (line 11), then the corresponding time, `ctime(&arrayOfStoredEvents[index].seconds)` (line 12), and finally "\r\n" (line 13). Remember that Figure 5.4 showed a diagram about the content that is obtained for the array `str` after executing this function.

```

1 int eventLogNumberOfStoredEvents()
2 {
3     return eventsIndex;
4 }
5
6 void eventLogRead( int index, char* str )
7 {
8     str[0] = '\0';
9     strcat( str, "Event = " );
10    strcat( str, arrayOfStoredEvents[index].typeOfEvent );
11    strcat( str, "\r\nDate and Time = " );
12    strcat( str, ctime(&arrayOfStoredEvents[index].seconds) );
13    strcat( str, "\r\n" );
14 }
15
16 void eventLogWrite( bool currentState, const char* elementName )
17 {
18     char eventAndStateStr[EVENT_LOG_NAME_MAX_LENGTH] = " ";
19
20     strcat( eventAndStateStr, elementName );
21     if ( currentState ) {
22         strcat( eventAndStateStr, "_ON" );
23     } else {
24         strcat( eventAndStateStr, "_OFF" );
25     }
26
27     arrayOfStoredEvents[eventsIndex].seconds = time(NULL);
28     strcpy( arrayOfStoredEvents[eventsIndex].typeOfEvent, eventAndStateStr );
29     if ( eventsIndex < EVENT_LOG_MAX_STORAGE - 1 ) {
30         eventsIndex++;
31     } else {
32         eventsIndex = 0;
33     }
34
35     pcSerialComStringWrite(eventAndStateStr);
36     pcSerialComStringWrite("\r\n");
37 }
38
39 void eventLogElementStateUpdate( bool lastState,
40                                 bool currentState,
41                                 const char* elementName )
42 {
43     if ( lastState != currentState ) {
44         eventLogWrite( currentState, elementName );
45     }
46 }
```

Code 5.12 Implementation of the functions of the `event_log` module (Part 2/2).

Finally, the implementation of `systemElementStateUpdate()` is replaced by the functions `eventLogWrite()`, which stores an event in the log, and `eventLogElementStateUpdate()`, which calls the function `eventLogWrite()` if the state being evaluated changes. Note that these functions receive a pointer to a constant string as a parameter (named `elementName` in both cases). Remember that more information about the functions `strcat` and `strcpy`, which were introduced in Chapter 4 and are used in Code 5.12, is available in [2].

Many functions of the `fire_alarm` module are shown in Code 5.13. `fireAlarmInit()` initializes the fire alarm subsystem by calling the functions `gasSensorInit()`, `temperatureSensorInit()`, and `sirenInit()`; `fireAlarmUpdate()` updates the fire alarm subsystem by calling the functions `fireAlarmActivationUpdate()`, `fireAlarmDeactivationUpdate()`, and `sirenUpdate()`; `gasDetectorStateRead()` returns the state of the gas detector; `overTemperatureDetectorStateRead()` returns the state of the over temperature detector; `gasDetectedRead()` returns true if gas is being detected; and `overTemperatureDetectedRead()` returns true if over temperature is being detected.

```

1 // Module: fire_alarm -----
2
3 void fireAlarmInit()
4 {
5     temperatureSensorInit();
6     gasSensorInit();
7     sirenInit();
8     strobeLightInit();
9
10    alarmTestButton.mode(PullDown);
11 }
12
13 void fireAlarmUpdate()
14 {
15     fireAlarmActivationUpdate();
16     fireAlarmDeactivationUpdate();
17     sirenUpdate( fireAlarmStrobeTime() );
18     strobeLightUpdate( fireAlarmStrobeTime() );
19 }
20
21 bool gasDetectorStateRead()
22 {
23     return gasDetectorState;
24 }
25
26 bool overTemperatureDetectorStateRead()
27 {
28     return overTemperatureDetectorState;
29 }
30
31 bool gasDetectedRead()
32 {
33     return gasDetected;
34 }
35
36 bool overTemperatureDetectedRead()
37 {
38     return overTemperatureDetected;
39 }
```

Code 5.13 Implementation of the functions of the `fire_alarm` module (Part 1/2).

In Code 5.14, other functions of the *fire_alarm* module are shown. The function *fireAlarmActivationUpdate()* controls the activation of the siren. In the previous code, this functionality was part of *alarmActivationUpdate()*. On lines 3 and 4, the functions *temperatureSensorUpdate()* and *gasSensorUpdate()* are called in order to update the reading of those sensors, since the functions *temperatureSensorReadCelsius()* on line 6 and *gasSensorRead()* on line 15 return the last readings without updating.

```

1 void fireAlarmActivationUpdate()
2 {
3     temperatureSensorUpdate();
4     gasSensorUpdate();
5
6     overTemperatureDetectorState = temperatureSensorReadCelsius() >
7             TEMPERATURE_C_LIMIT_ALARM;
8
9     if ( overTemperatureDetectorState ) {
10         overTemperatureDetected = ON;
11         sirenStateWrite(ON);
12         strobeLightStateWrite(ON);
13     }
14
15     gasDetectorState = !gasSensorRead();
16
17     if ( gasDetectorState ) {
18         gasDetected = ON;
19         sirenStateWrite(ON);
20         strobeLightStateWrite(ON);
21     }
22
23     if ( alarmTestButton ) {
24         overTemperatureDetected = ON;
25         gasDetected = ON;
26         sirenStateWrite(ON);
27         strobeLightStateWrite(ON);
28     }
29 }
30
31 void fireAlarmDeactivationUpdate()
32 {
33     if ( sirenStateRead() ) {
34         if ( codeMatchFrom(CODE_KEYPAD) ||
35             codeMatchFrom(CODE_PC_SERIAL) ) {
36             fireAlarmDeactivate();
37         }
38     }
39 }
40
41 void fireAlarmDeactivate()
42 {
43     sirenStateWrite(OFF);
44     strobeLightStateWrite(OFF);
45     overTemperatureDetected = OFF;
46     gasDetected = OFF;
47 }
48
49 int fireAlarmStrobeTime()
50 {
51     if( gasDetectedRead() && overTemperatureDetectedRead() ) {
52         return STROBE_TIME_GAS_AND_OVER_TEMP;
53     } else if ( gasDetectedRead() ) {
54         return STROBE_TIME_GAS;
55     } else if ( overTemperatureDetectedRead() ) {
56         return STROBE_TIME_OVER_TEMP;
57     } else {
58         return 0;
59     }
60 }
```

Code 5.14 Implementation of the functions of the *fire_alarm* module (Part 2/2).

The function `fireAlarmDeactivationUpdate()` controls the deactivation of the siren. When the alarm is active, the function `codeMatchFrom()` on lines 34 and 35 assesses if there is a new deactivation code to check. If there is a new deactivation code, the `fireAlarmDeactivate()` function is called. This implementation decouples the condition to deactivate the alarm from the actual deactivation of the alarm.

The function `fireAlarmDeactivate()` implements the deactivation of the siren and the strobe light by setting them to the OFF state, as well as setting `overTemperatureDetected` and `gasDetected` to OFF. The function `fireAlarmStrobeTime()` controls the siren and the strobe light on and off time. This function was part of the function `alarmActivationUpdate()` in the former version of the code.

In general, much of the functionality that was included in `alarmActivationUpdate()` in the previous version of the code (Example 4.4) is now organized in the `fire_alarm` and `siren` modules in order to decouple the control of the activation and deactivation of the siren from the activation and deactivation itself. In this way, more actions can be easily included on line 9 and/or line 17, following the modularity principle. For example, water sprinklers could be turned on, or a phone call could be made (which are not included in this example).

In Code 5.15, the implementation of the gas sensor functionality is shown. This module implements the reading of the gas sensor. The first two functions are actually useless but are included in order to keep the same structure as in the other module.

```
1 // Module: gas_sensor -----
2
3 void gasSensorInit( )
4 {
5 }
6
7 void gasSensorUpdate( )
8 {
9 }
10
11 bool gasSensorRead( )
12 {
13     return mq2;
14 }
```

Code 5.15 Implementation of the functions of the `gas_sensor` module.

In Code 5.16, Code 5.17, and Code 5.18 the functions of the `matrix_keypad` module are shown. There are small changes from the previous version of the code:

- The function `matrixKeypadInit()` receives the parameter `timeIncrement_ms` instead of using the value defined by `TIME_INCREMENT_MS`.
- The implementation of the function `matrixKeypadReset()` is used to reset the FSM of the matrix keypad. In the previous version of the code, this was done by `matrixKeypadInit()`.

```

1 // Module: matrix_keypad -----
2
3 void matrixKeypadInit( int updateTime_ms )
4 {
5     timeIncrement_ms = updateTime_ms;
6     matrixKeypadState = MATRIX_KEYPAD_SCANNING;
7     int pinIndex = 0;
8     for( pinIndex=0; pinIndex<MATRIX_KEYPAD_NUMBER_OF_COLS; pinIndex++ ) {
9         (keypadColPins[pinIndex]).mode(PullUp);
10    }
11 }
```

Code 5.16 Implementation of the functions of the matrix_keypad module (Part 1/3).

```

1 char matrixKeypadUpdate()
2 {
3     char keyDetected = '\0';
4     char keyReleased = '\0';
5
6     switch( matrixKeypadState ) {
7
8     case MATRIX_KEYPAD_SCANNING:
9         keyDetected = matrixKeypadScan();
10        if( keyDetected != '\0' ) {
11            matrixKeypadLastKeyPressed = keyDetected;
12            accumulatedDebounceMatrixKeypadTime = 0;
13            matrixKeypadState = MATRIX_KEYPAD_DEBOUNCE;
14        }
15        break;
16
17     case MATRIX_KEYPAD_DEBOUNCE:
18        if( accumulatedDebounceMatrixKeypadTime >=
19            DEBOUNCE_KEY_TIME_MS ) {
20            keyDetected = matrixKeypadScan();
21            if( keyDetected == matrixKeypadLastKeyPressed ) {
22                matrixKeypadState = MATRIX_KEYPAD_KEY_HOLD_PRESSED;
23            } else {
24                matrixKeypadState = MATRIX_KEYPAD_SCANNING;
25            }
26        }
27        accumulatedDebounceMatrixKeypadTime =
28        accumulatedDebounceMatrixKeypadTime + timeIncrement_ms;
29        break;
30
31     case MATRIX_KEYPAD_KEY_HOLD_PRESSED:
32        keyDetected = matrixKeypadScan();
33        if( keyDetected != matrixKeypadLastKeyPressed ) {
34            if( keyDetected == '\0' ) {
35                keyReleased = matrixKeypadLastKeyPressed;
36            }
37            matrixKeypadState = MATRIX_KEYPAD_SCANNING;
38        }
39        break;
40
41     default:
42        matrixKeypadReset();
43        break;
44    }
45    return keyReleased;
46 }
```

Code 5.17 Implementation of the functions of the matrix_keypad module (Part 2/3).

```

1  char matrixKeypadScan()
2  {
3      int row = 0;
4      int col = 0;
5      int i = 0;
6
7      char matrixKeypadIndexToCharArray[] = {
8          '1', '2', '3', 'A',
9          '4', '5', '6', 'B',
10         '7', '8', '9', 'C',
11         '*', '0', '#', 'D',
12     };
13
14     for( row=0; row<MATRIX_KEYPAD_NUMBER_OF_ROWS; row++ ) {
15
16         for( i=0; i<MATRIX_KEYPAD_NUMBER_OF_ROWS; i++ ) {
17             keypadRowPins[i] = ON;
18         }
19
20         keypadRowPins[row] = OFF;
21
22         for( col=0; col<MATRIX_KEYPAD_NUMBER_OF_COLS; col++ ) {
23             if( keypadColPins[col] == OFF ) {
24                 return matrixKeypadIndexToCharArray[
25                     row*MATRIX_KEYPAD_NUMBER_OF_ROWS + col];
26             }
27         }
28     }
29     return '\0';
30 }
31
32 void matrixKeypadReset()
33 {
34     matrixKeypadState = MATRIX_KEYPAD_SCANNING;
35 }
```

Code 5.18 Implementation of the functions of the *matrix_keypad* module (Part 3/3).

The communication with the PC is implemented in the *pc_serial_com* module, as shown from Code 5.19 to Code 5.22. The function *uartTask()*, which was implemented in previous chapters, was removed, and its behavior is now implemented in a different way. One of the reasons for this change is that *uartUsb.read()* was used in *uartTask()* in such a way that the responsiveness of the program was affected. In particular, *uartUsb.read()* was used four times in *uartTask()*: first, to assess if there is a readable character in *uartUsb*; second, to get the four digits of the numeric code, one after the other; third, to get the four digits to set a new numeric code, one after the other; and fourth, to flush *uartUsb* once a new date and time had been set. The usage of *uartUsb.read()* in the second and third cases blocked the program execution until four new characters were entered, which reduced the program's responsiveness to other inputs.

In order to solve this problem, the new implementation of the program uses *uartUsb.read()* only twice: first, to assess if there is a readable character in *uartUsb* (in the new function *pcSerialComCharRead()*, line 12 of Code 5.19); second, to read one character in *pcSerialComStringRead()*, in line 5 of Code 5.20. With the new implementation, if the alarm is activated, the Alarm LED does not stop blinking when the program is waiting for the user to enter the alarm deactivation code, as can be concluded from the following explanation.

In this new implementation, `pcSerialComCharRead()` is called only in line 24 of Code 5.19 by the function `pcSerialComUpdate()`. This function implements an FSM, as can be seen from lines 22 to 41 of Code 5.19. There is a switch over `pcSerialComMode` and, depending on its value, the functions `pcSerialComCommandUpdate()`, `pcSerialComGetCodeUpdate()`, or `pcSerialComSaveNewCodeUpdate()` are executed. The implementation of these three functions is shown on Code 5.20 and is discussed below.

It is important to note that prior to the first execution of `pcSerialComUpdate()`, `pcSerialComMode` is initialized as `PC_SERIAL_COMMANDS` (line 35 of Code 5.4). So, the first time a character is received, the FSM will call `pcSerialComCommandUpdate()` in order to determine what function to call depending on the received command, as can be seen from lines 37 to 52 of Code 5.20.

If the received command is '4', the function `commandEnterCodeSequence()` is called (line 43 of Code 5.20). This function modifies the state of `pcSerialComMode` to `PC_SERIAL_GET_CODE`, as can be seen on line 31 of Code 5.21. In this way, the next time the FSM is executed, the function `pcSerialComGetCodeUpdate()` is called (line 31 of Code 5.19). This function will be called by the FSM until four characters are received, because once the statement of line 16 of Code 5.20 becomes true, `pcSerialComMode` is set to `PC_SERIAL_COMMANDS` on line 17. In this way, a new command will be expected by the FSM. A similar behavior is true for `pcSerialComSaveNewCodeUpdate()`, which can be seen from lines 23 to 35 of Code 5.20.

The remaining program code shown in Code 5.19 to Code 5.22 is very similar to the program code discussed in the previous chapters, the only difference being that the code is now refactored into functions in order to increase its modularity, with appropriate cohesion and coupling. Also note that `pcSerialComInit()` calls `availableCommands()`. Thus, the list of available commands is printed during the initialization process.

```

1 // Module: pc_serial_com -----
2
3 void pcSerialComInit()
4 {
5     availableCommands();
6 }
7
8 char pcSerialComCharRead()
9 {
10    char receivedChar = '\0';
11    if( uartUsb.readable() ) {
12        uartUsb.read( &receivedChar, 1 );
13    }
14    return receivedChar;
15 }
16
17 void pcSerialComStringWrite( const char* str )
18 {
19     uartUsb.write( str, strlen(str) );
20 }
21
22 void pcSerialComUpdate()
23 {
24     char receivedChar = pcSerialComCharRead();
25     if( receivedChar != '\0' ) {
26         switch ( pcSerialComMode ) {

```

```

27         case PC_SERIAL_COMMANDS:
28             pcSerialComCommandUpdate( receivedChar );
29             break;
30         case PC_SERIAL_GET_CODE:
31             pcSerialComGetCodeUpdate( receivedChar );
32             break;
33         case PC_SERIAL_SAVE_NEW_CODE:
34             pcSerialComSaveNewCodeUpdate( receivedChar );
35             break;
36     default:
37         pcSerialComMode = PC_SERIAL_COMMANDS;
38         break;
39     }
40 }
41
42 bool pcSerialComCodeCompleteRead()
43 {
44     return codeComplete;
45 }
46
47 void pcSerialComCodeCompleteWrite( bool state )
48 {
49     codeComplete = state;
50 }

```

Code 5.19 Implementation of the functions of the `pc_serial_com` module (Part 1/4).

```

1 void pcSerialComStringRead( char* str, int strLength )
2 {
3     int strIndex;
4     for ( strIndex = 0; strIndex < strLength; strIndex++ ) {
5         uartUsb.read( &str[strIndex] , 1 );
6         uartUsb.write( &str[strIndex] , 1 );
7     }
8     str[strLength]='\0';
9 }
10
11 void pcSerialComGetCodeUpdate( char receivedChar )
12 {
13     codeSequenceFromPcSerialCom[numberOfCodeChars] = receivedChar;
14     pcSerialComStringWrite( "*" );
15     numberOfCodeChars++;
16     if ( numberOfCodeChars >= CODE_NUMBER_OF_KEYS ) {
17         pcSerialComMode = PC_SERIAL_COMMANDS;
18         codeComplete = true;
19         numberOfCodeChars = 0;
20     }
21 }
22
23 void pcSerialComSaveNewCodeUpdate( char receivedChar )
24 {
25     char newCodeSequence[CODE_NUMBER_OF_KEYS];
26     newCodeSequence[numberOfCodeChars] = receivedChar;
27     pcSerialComStringWrite( "*" );
28     numberOfCodeChars++;
29     if ( numberOfCodeChars >= CODE_NUMBER_OF_KEYS ) {
30         pcSerialComMode = PC_SERIAL_COMMANDS;
31         numberOfCodeChars = 0;
32         codeWrite( newCodeSequence );
33         pcSerialComStringWrite( "\r\nNew code configured\r\n\r\n" );
34     }

```

```

35 }
36
37 void pcSerialComCommandUpdate( char receivedChar )
38 {
39     switch (receivedChar) {
40         case '1': commandShowCurrentAlarmState(); break;
41         case '2': commandShowCurrentGasDetectorState(); break;
42         case '3': commandShowCurrentOverTemperatureDetectorState(); break;
43         case '4': commandEnterCodeSequence(); break;
44         case '5': commandEnterNewCode(); break;
45         case 'c': case 'C': commandShowCurrentTemperatureInCelsius(); break;
46         case 'f': case 'F': commandShowCurrentTemperatureInFahrenheit(); break;
47         case 's': case 'S': commandSetDateAndTime(); break;
48         case 't': case 'T': commandShowDateAndTime(); break;
49         case 'e': case 'E': commandShowStoredEvents(); break;
50     default: availableCommands(); break;
51 }
52 }
53
54 void availableCommands()
55 {
56     pcSerialComStringWrite( "Available commands:\r\n" );
57     pcSerialComStringWrite( "Press '1' to get the alarm state\r\n" );
58     pcSerialComStringWrite( "Press '2' to get the gas detector state\r\n" );
59     pcSerialComStringWrite( "Press '3' to get the over temperature detector state\r\n" );
60     pcSerialComStringWrite( "Press '4' to enter the code to deactivate the alarm\r\n" );
61     pcSerialComStringWrite( "Press '5' to enter a new code to deactivate the alarm\r\n" );
62     pcSerialComStringWrite( "Press 'f' or 'F' to get lm35 reading in Fahrenheit\r\n" );
63     pcSerialComStringWrite( "Press 'c' or 'C' to get lm35 reading in Celsius\r\n" );
64     pcSerialComStringWrite( "Press 's' or 'S' to set the date and time\r\n" );
65     pcSerialComStringWrite( "Press 't' or 'T' to get the date and time\r\n" );
66     pcSerialComStringWrite( "Press 'e' or 'E' to get the stored events\r\n" );
67     pcSerialComStringWrite( "\r\n" );
68 }

```

Code 5.20 Implementation of the functions of the `pc_serial_com` module (Part 2/4).

```

1 void commandShowCurrentAlarmState()
2 {
3     if ( sirenStateRead() ) {
4         pcSerialComStringWrite( "The alarm is activated\r\n" );
5     } else {
6         pcSerialComStringWrite( "The alarm is not activated\r\n" );
7     }
8 }
9
10 void commandShowCurrentGasDetectorState()
11 {
12     if ( gasDetectorStateRead() ) {
13         pcSerialComStringWrite( "Gas is being detected\r\n" );
14     } else {
15         pcSerialComStringWrite( "Gas is not being detected\r\n" );
16     }
17 }
18
19 void commandShowCurrentOverTemperatureDetectorState()
20 {
21     if ( overTemperatureDetectorStateRead() ) {
22         pcSerialComStringWrite( "Temperature is above the maximum level\r\n" );
23     } else {
24         pcSerialComStringWrite( "Temperature is below the maximum level\r\n" );
25     }

```

```

26 }
27
28 void commandEnterCodeSequence()
29 {
30     if( sirenStateRead() ) {
31         pcSerialComStringWrite( "Please enter the four digits numeric code " );
32         pcSerialComStringWrite( "to deactivate the alarm: " );
33         pcSerialComMode = PC_SERIAL_GET_CODE;
34         codeComplete = false;
35         numberOfCodeChars = 0;
36     } else {
37         pcSerialComStringWrite( "Alarm is not activated.\r\n" );
38     }
39 }
40
41 void commandEnterNewCode()
42 {
43     pcSerialComStringWrite( "Please enter the new four digits numeric code " );
44     pcSerialComStringWrite( "to deactivate the alarm: " );
45     numberOfCodeChars = 0;
46     pcSerialComMode = PC_SERIAL_SAVE_NEW_CODE;
47
48 }
49
50 void commandShowCurrentTemperatureInCelsius()
51 {
52     char str[100] = "";
53     sprintf ( str, "Temperature: %.2f \u00b0 C\r\n",
54               temperatureSensorReadCelsius() );
55     pcSerialComStringWrite( str );
56 }
57
58 void commandShowCurrentTemperatureInFahrenheit()
59 {
60     char str[100] = "";
61     sprintf ( str, "Temperature: %.2f \u00b0 C\r\n",
62               temperatureSensorReadFahrenheit() );
63     pcSerialComStringWrite( str );
64 }
```

Code 5.21 Implementation of the functions of the `pc_serial_com` module (Part 3/4).

```

1 void commandSetDateAndTime()
2 {
3     char year[5] = "";
4     char month[3] = "";
5     char day[3] = "";
6     char hour[3] = "";
7     char minute[3] = "";
8     char second[3] = "";
9
10    pcSerialComStringWrite("\r\nType four digits for the current year (YYYY): ");
11    pcSerialComStringRead( year, 4);
12    pcSerialComStringWrite("\r\n");
13
14    pcSerialComStringWrite("Type two digits for the current month (01-12): ");
15    pcSerialComStringRead( month, 2);
16    pcSerialComStringWrite("\r\n");
17
18    pcSerialComStringWrite("Type two digits for the current day (01-31): ");
19    pcSerialComStringRead( day, 2);
```

```

20     pcSerialComStringWrite( "\r\n" );
21
22     pcSerialComStringWrite("Type two digits for the current hour (00-23): ");
23     pcSerialComStringRead( hour, 2 );
24     pcSerialComStringWrite(" \r\n");
25
26     pcSerialComStringWrite("Type two digits for the current minutes (00-59): ");
27     pcSerialComStringRead( minute, 2 );
28     pcSerialComStringWrite(" \r\n");
29
30     pcSerialComStringWrite("Type two digits for the current seconds (00-59): ");
31     pcSerialComStringRead( second, 2 );
32     pcSerialComStringWrite(" \r\n");
33
34     pcSerialComStringWrite("Date and time has been set\r\n");
35
36     dateAndTimeWrite( atoi(year), atoi(month), atoi(day),
37                       atoi(hour), atoi(minute), atoi(second) );
38 }
39
40 void commandShowDateAndTime()
41 {
42     char str[100] = "";
43     sprintf ( str, "Date and Time = %s", dateAndTimeRead() );
44     pcSerialComStringWrite( str );
45     pcSerialComStringWrite(" \r\n");
46 }
47
48 void commandShowStoredEvents()
49 {
50     char str[EVENT_STR_LENGTH];
51     int i;
52     for ( i = 0; i < eventLogNumberOfStoredEvents(); i++ ) {
53         eventLogRead( i, str );
54         pcSerialComStringWrite( str );
55         pcSerialComStringWrite( "\r\n" );
56     }
57 }
```

Code 5.22 Implementation of the functions of the `pc_serial_com` module (Part 4/4).

NOTE: The implementations of `pcSerialComCharRead()` and `pcSerialComStringWrite()`, which are shown in Code 5.19, and `pcSerialComStringRead()`, which is shown in Code 5.20, were already introduced in the Case Study section of Chapter 4 and, therefore, are not discussed here.

Code 5.23 shows the implementation of the functions of the siren module. The functions of this module are called by the functions `fireAlarmActivationUpdate()` and `fireAlarmDeactivationUpdate()`. They have the duty of turning on and off the siren (implemented by means of the buzzer), as can be seen on line 26, which is used to toggle the state of the buzzer every time `accumulatedTimeAlarm` reaches `strobeTime`. In this way, the buzzer now generates an intermittent sound instead of the continuous sound that was implemented in Chapter 3.



NOTE: On lines 5 and 29 of Code 5.23, the `sirenPin` is set to ON in order to turn off the buzzer. This is because of the assumption that the circuit introduced in Figure 5.5 is being used. If, instead, the circuit introduced in Figure 5.6 is being used, lines 5 and 29 of Code 5.23 should be modified to “`sirenPin = OFF`” in order to turn off the buzzer.

```

1 // Module: siren -----
2
3 void sirenInit()
4 {
5     sirenPin = ON;
6 }
7
8 bool sirenStateRead()
9 {
10    return sirenState;
11 }
12
13 void sirenStateWrite( bool state )
14 {
15     sirenState = state;
16 }
17
18 void sirenUpdate( int strobeTime )
19 {
20     static int accumulatedTimeAlarm = 0;
21     accumulatedTimeAlarm = accumulatedTimeAlarm + SYSTEM_TIME_INCREMENT_MS;
22
23     if( sirenState ) {
24         if( accumulatedTimeAlarm >= strobeTime ) {
25             accumulatedTimeAlarm = 0;
26             sirenPin= !sirenPin;
27         }
28     } else {
29         sirenPin = ON;
30     }
31 }
```

Code 5.23 Implementation of the functions of the siren module.

The functions of the `smart_home_system` module are shown in Code 5.24. Both functions `smartHomeSystemInit()` and `smartHomeSystemUpdate()` are called by `main()`, as was shown in Code 5.7. By means of these functions, the `user_interface`, `fire_alarm`, and `pc_serial_com` modules are initialized, and those modules, together with the `event_log` module, are updated at a rate given by the delay on line 16.

```

1 // Module: smart_home_system -----
2
3 void smartHomeSystemInit()
4 {
5     userInterfaceInit();
6     fireAlarmInit();
7     pcSerialComInit();
8 }
9
10 void smartHomeSystemUpdate()
11 {
12     fireAlarmUpdate();
13     userInterfaceUpdate();
14     pcSerialComUpdate();
15     eventLogUpdate();
16     delay(SYSTEM_TIME_INCREMENT_MS);
17 }
```

Code 5.24 Implementation of the functions of the smart_home_system module.

Code 5.25 shows the functions of the *strobe_light* module. Its functionality is very similar to the functionality of the *siren* module and, therefore, is not discussed here.

```

1 // Module: strobe_light -----
2
3 void strobeLightInit()
4 {
5     strobeLight = OFF;
6 }
7
8 bool strobeLightStateRead()
9 {
10     return strobeLightState;
11 }
12
13 void strobeLightStateWrite( bool state )
14 {
15     strobeLightState = state;
16 }
17
18 void strobeLightUpdate( int strobeTime )
19 {
20     static int accumulatedTimeAlarm = 0;
21     accumulatedTimeAlarm = accumulatedTimeAlarm + SYSTEM_TIME_INCREMENT_MS;
22
23     if( strobeLightState ) {
24         if( accumulatedTimeAlarm >= strobeTime ) {
25             accumulatedTimeAlarm = 0;
26             strobeLight= !strobeLight;
27         }
28     } else {
29         strobeLight = OFF;
30     }
31 }
```

Code 5.25 Implementation of the functions of the strobe light module.

Code 5.26 shows the functions of the *temperature_sensor* module. The functionality implemented by *temperatureSensorUpdate()* was, in the previous code (Example 4.4), implemented by the function *alarmActivationUpdate()*. The other functions of this module are very similar to the functions of Example 4.4, besides the changes in their names (which were in order to indicate that they belong to the *temperature_sensor* module).

```

1 // Module: temperature_sensor -----
2
3 void temperatureSensorInit()
4 {
5     int i;
6
7     for( i = 0; i < LM35_NUMBER_OF_AVG_SAMPLES ; i++ ) {
8         lm35ReadingsArray[i] = 0;
9     }
10 }
11
12 void temperatureSensorUpdate()
13 {
14     static int lm35SampleIndex = 0;
15     float lm35ReadingsSum = 0.0;
16     float lm35ReadingsAverage = 0.0;
17
18     int i = 0;
19
20     lm35ReadingsArray[lm35SampleIndex] = lm35.read();
21     lm35SampleIndex++;
22     if ( lm35SampleIndex >= LM35_NUMBER_OF_AVG_SAMPLES ) {
23         lm35SampleIndex = 0;
24     }
25
26     lm35ReadingsSum = 0.0;
27     for (i = 0; i < LM35_NUMBER_OF_AVG_SAMPLES; i++) {
28         lm35ReadingsSum = lm35ReadingsSum + lm35ReadingsArray[i];
29     }
30     lm35ReadingsAverage = lm35ReadingsSum / LM35_NUMBER_OF_AVG_SAMPLES;
31     lm35TemperatureC = analogReadingScaledWithTheLM35Formula ( lm35ReadingsAverage );
32 }
33
34 float temperatureSensorReadCelsius()
35 {
36     return lm35TemperatureC;
37 }
38
39 float temperatureSensorReadFahrenheit()
40 {
41     return celsiusToFahrenheit( lm35TemperatureC );
42 }
43
44 float celsiusToFahrenheit( float tempInCelsiusDegrees )
45 {
46     return ( tempInCelsiusDegrees * 9.0 / 5.0 + 32.0 );
47 }
48
49 float analogReadingScaledWithTheLM35Formula( float analogReading )
50 {
51     return ( analogReading * 3.3 / 0.01 );
52 }
```

Code 5.26 Implementation of the functions of the *temperature_sensor* module.

Finally, Code 5.27 and Code 5.28 show the implementation of the functions of the user interface. The core of the functionality of this module is the function `userInterfaceMatrixKeypadUpdate()`. Two details must be highlighted about this function:

1. On line 12 it can be seen that `codeComplete` is set to true once four keys have been pressed on the matrix keypad. Hence, the “#” is not used anymore to signal the end of a code being entered.
2. If `sirenStateRead()` returns true and `systemBlockedStateRead()` returns false, then the entered keys are stored. Alternatively, the Incorrect code LED is turned off only if the “#” key is pressed twice (line 18).

```

1 // Module: user_interface -----
2
3 void userInterfaceInit()
4 {
5     incorrectCodeLed = OFF;
6     systemBlockedLed = OFF;
7     matrixKeypadInit( SYSTEM_TIME_INCREMENT_MS );
8 }
9
10 void userInterfaceUpdate()
11 {
12     userInterfaceMatrixKeypadUpdate();
13     incorrectCodeIndicatorUpdate();
14     systemBlockedIndicatorUpdate();
15 }
16
17 bool incorrectCodeStateRead()
18 {
19     return incorrectCodeState;
20 }
21
22 void incorrectCodeStateWrite( bool state )
23 {
24     incorrectCodeState = state;
25 }
26
27 void incorrectCodeIndicatorUpdate()
28 {
29     incorrectCodeLed = incorrectCodeStateRead();
30 }
31
32 bool systemBlockedStateRead()
33 {
34     return systemBlockedState;
35 }
36
37 void systemBlockedStateWrite( bool state )
38 {
39     systemBlockedState = state;
40 }
41
42 void systemBlockedIndicatorUpdate()
43 {
44     systemBlockedLed = systemBlockedState;
45 }
46
47 bool userInterfaceCodeCompleteRead()
48 {
49     return codeComplete;
50 }
51
52 void userInterfaceCodeCompleteWrite( bool state )
53 {
54     codeComplete = state;
55 }
```

Code 5.27 Implementation of the functions of the `user_interface` module (Part 1/2).

```

1 void userInterfaceMatrixKeypadUpdate()
2 {
3     char keyReleased = matrixKeypadUpdate();
4
5     if( keyReleased != '\0' ) {
6
7         if( sirenStateRead() && !systemBlockedStateRead() ) {
8             if( !incorrectCodeStateRead() ) {
9                 codeSequenceFromUserInterface[numberOfCodeChars] = keyReleased;
10                numberOfCodeChars++;
11                if( numberOfCodeChars >= CODE_NUMBER_OF_KEYS ) {
12                    codeComplete = true;
13                    numberOfCodeChars = 0;
14                }
15            } else {
16                if( keyReleased == '#' ) {
17                    numberOfHashKeyReleased++;
18                    if( numberOfHashKeyReleased >= 2 ) {
19                        numberOfHashKeyReleased = 0;
20                        numberOfCodeChars = 0;
21                        codeComplete = false;
22                        incorrectCodeState = OFF;
23                    }
24                }
25            }
26        }
27    }
28 }

```

Code 5.28 Implementation of the functions of the user_interface module (Part 2/2).



NOTE: Given the changes introduced by the `userInterfaceMatrixKeypadUpdate()` function, to deactivate the alarm, keys “1”, “8”, “0”, and “5” must be pressed on the matrix keypad (i.e., it is no longer necessary to press key “#” after entering a code). If an incorrect code is entered, the “#” key must be pressed twice on the matrix keypad to enable the entering of a new code, just as in the implementation shown in Chapter 4.

5.4 Organizing the Modules of the Smart Home System into Different Files

5.4.1 Principles Followed to Organize the Modules into Files: Variables and Functions

In order to organize the modules into files, each variable will be declared only in the file of the specific module that makes use of it. Some variables are used only inside a given function, and their value must remain in memory from one execution of that given function to the next one. For example, when the FSM of the matrix keypad is in the `MATRIX_KEYPAD_DEBOUNCE` state, then the variable `accumulatedDebounceMatrixKeypadTime` is incremented by `timeIncrement_ms` with each execution of the `matrixKeypadUpdate()` function until it reaches the value `DEBOUNCE_BUTTON_TIME_MS`. For this reason, `accumulatedDebounceMatrixKeypadTime` must remain in memory from one execution of `matrixKeypadUpdate()` to the next one and, therefore, is declared as `static`. This can be seen on line 3 of Code 5.29.

```

1  char matrixKeypadUpdate()
2  {
3      static int accumulatedDebounceMatrixKeypadTime = 0;
4      static char matrixKeypadLastKeyPressed = '\0';
5
6      char keyDetected = '\0';
7      char keyReleased = '\0';
8
9      switch( matrixKeypadState ) {

```

Code 5.29 The first lines of the function `matrixKeypadUpdate()`, where some variables are declared as `static`.

On line 4 of Code 5.29, it can be seen that the variable `matrixKeypadLastKeyPressed` is also declared as `static`, as its value must remain in memory from one execution of `matrixKeypadUpdate()` to the next one. In contrast, the values of `keyDetected` and `keyReleased` are not declared as `static`, as the value of `keyDetected` is assigned after a call to the function `matrixKeypadScan()`, while `keyReleased` is assigned the value of `matrixKeypadLastKeyPressed` inside the `MATRIX_KEYPAD_KEY_HOLD_PRESSED` case.

In Table 5.14, the variables declared as `static` inside different functions are listed. These `static` local variables remain in memory while the program is running, even after the execution of those functions is completed. A variable that is not declared as `static` inside a function is erased when the execution of the function is over.

Table 5.14 Variables that will be declared as `static` inside given functions.

Module	Function	Variables declaration
matrix_keypad	matrixKeypadUpdate()	static int accumulatedDebounceMatrixKeypadTime = 0; static char matrixKeypadLastKeyPressed = '\0';
pc_serial_com	pcSerialComSaveNew CodeUpdate()	static char newCodeSequence[CODE_NUMBER_OF_KEYS];
temperature_ sensor	temperatureSensor Update()	static int lm35SampleIndex = 0;
user_interface	userInterfaceMatrix KeypadUpdate()	static int numberOfHashKeyReleased = 0;

With the aim of guaranteeing that the *private scope* of each module is not invaded by other modules, some variables that are used by multiple functions of a single module (but not outside the module) are declared as private inside that module. This is done by means of declaring those variables as `static`, but outside the functions. In this way, a variable declared as `static` outside of a function can only be used by other functions within the same `.c/cpp` file. The public and private variables that will be declared in each module are listed in Table 5.15. The modules `date_and_time`, `gas_sensor`, and `smart_home_system` have neither private nor public variables.



WARNING: Variables declared as *static* inside a function (as in Code 5.29 and Table 6.14) are local variables that retain their values as explained, while variables declared as *static* outside of a function (as in Table 5.15) are global variables that can only be accessed by functions declared in the same file.

Table 5.15 Public and private variables declared in each module.

Module	Public variables
code	<p>None</p> <p>Private variables</p> <pre>static int numberOfIncorrectCodes = 0; static char codeSequence[CODE_NUMBER_OF_KEYS] = {'1','8','0','5'};</pre>
Module	Public variables
event_log	<p>None</p> <p>Private variables</p> <pre>static bool sirenLastState = OFF; static bool gasLastState = OFF; static bool tempLastState = OFF; static bool ICLastState = OFF; static bool SBLastState = OFF; static int eventsIndex = 0; static systemEvent_t arrayOfStoredEvents[EVENT_LOG_MAX_STORAGE];</pre>
Module	Public variables
fire_alarm	<p>None</p> <p>Private variables</p> <pre>static bool gasDetected = OFF; static bool overTemperatureDetected = OFF; static bool gasDetectorState = OFF; static bool overTemperatureDetectorState = OFF;</pre>
Module	Public variables
matrix_keypad	<p>None</p> <p>Private variables</p> <pre>static matrixKeypadState_t matrixKeypadState; static int timeIncrement_ms = 0;</pre>
Module	Public variables
pc_serial_com	<p>char codeSequenceFromPcSerialCom[CODE_NUMBER_OF_KEYS];</p> <p>Private variables</p> <pre>static pcSerialComMode_t pcSerialComMode = PC_SERIAL_COMMANDS; static bool codeCompleteFromPcSerialCom = false; static int numberOfCodeCharsFromPcSerialCom = 0;</pre>

Module	Public variables
siren	<p>None</p> <p>Private variables</p> <pre>static bool sirenState = OFF;</pre>
Module	Public variables
strobe_light	<p>None</p> <p>Private variables</p> <pre>static bool strobeLightState = OFF;</pre>
Module	Public variables
temperature_sensor	<p>None</p> <p>Private variables</p> <pre>float lm35TemperatureC = 0.0; float lm35AvgReadingsArray[LM35_NUMBER_OF_AVG_SAMPLES];</pre>
Module	Public variables
user_interface	<pre>char codeSequenceFromUserInterface[CODE_NUMBER_OF_KEYS];</pre> <p>Private variables</p> <pre>static bool incorrectCodeState = OFF; static bool systemBlockedState = OFF; static bool codeComplete = false; static int numberOfCodeChars = 0;</pre>

In Table 5.2 to Table 5.13, some functions were shown that are used by different modules, while other functions are used only by functions in the same module. The functions that must be available to other modules are called *public functions*, while the functions that must be available only for functions in the same module are called *private functions*. Table 5.16 shows which public and private functions will be declared in each module. The private functions are identified by the word *static* prior to their declaration.

Table 5.16 Public and private functions.

Module	Public functions
code	<pre>void codeWrite(char* newCodeSequence); bool codeMatchFrom(codeOrigin_t codeOrigin);</pre> <p>Private functions</p> <pre>static bool codeMatch(char* codeToCompare); static void codeDeactivate();</pre>

Module	Public functions
date_and_time	<pre>char* dateAndTimeRead(); void dateAndTimeWrite(int year, int month, int day, int hour, int minute, int second);</pre>
	Private functions
	None
Module	Public functions
event_log	<pre>void eventLogUpdate(); int eventLogNumberOfStoredEvents(); void eventLogRead(int index, char* str); void eventLogWrite(bool currentState, const char* elementName);</pre>
	Private functions
	<pre>static void eventLogElementStateUpdate(bool lastState, bool currentState, const char* elementName);</pre>
Module	Public functions
fire_alarm	<pre>void fireAlarmInit(); void fireAlarmUpdate(); bool gasDetectorStateRead(); bool overTemperatureDetectorStateRead(); bool gasDetectedRead(); bool overTemperatureDetectedRead();</pre>
	Private functions
	<pre>static void fireAlarmActivationUpdate(); static void fireAlarmDeactivationUpdate(); static void fireAlarmDeactivate(); static int fireAlarmStrobeTime();</pre>
Module	Public functions
gas_sensor	<pre>void gasSensorInit(); void gasSensorUpdate(); bool gasSensorRead();</pre>
	Private functions
	None
Module	Public functions
matrix_keypad	<pre>void matrixKeypadInit(int updateTime_ms); char matrixKeypadUpdate();</pre>
	Private functions
	<pre>static char matrixKeypadScan(); static void matrixKeypadReset();</pre>

Module	Public functions
pc_serial_com	<pre>void pcSerialComInit(); char pcSerialComCharRead(); void pcSerialComStringWrite(const char* str); void pcSerialComUpdate(); bool pcSerialComCodeCompleteRead(); void pcSerialComCodeCompleteWrite(bool state);</pre>
	Private functions
	<pre>static void pcSerialComStringRead(char* str, int strLength); static void pcSerialComGetCodeUpdate(char receivedChar); static void pcSerialComSaveNewCodeUpdate(char receivedChar); static void pcSerialComCommandUpdate(char receivedChar); static void availableCommands(); static void commandShowCurrentSirenStrobeLightState(); static void commandShowCurrentGasDetectorState(); static void commandShowCurrentOverTemperatureDetectorState(); static void commandEnterCodeSequence(); static void commandEnterNewCode(); static void commandShowCurrentTemperatureInCelsius(); static void commandShowCurrentTemperatureInFahrenheit(); static void commandSetDateAndTime(); static void commandShowDateAndTime(); static void commandShowStoredEvents();</pre>
Module	Public functions
siren	<pre>void sirenInit(); bool sirenStateRead(); void sirenStateWrite(bool state); void sirenUpdate(int strobeTime);</pre>
	Private functions
	None
Module	Public functions
strobe_light	<pre>void strobeLightInit(); bool strobeLightStateRead(); void strobeLightStateWrite(bool state); void strobeLightUpdate(int strobeTime);</pre>
	Private functions
	None
Module	Public functions
smart_home_system	<pre>void smartHomeSystemInit(); void smartHomeSystemUpdate();</pre>
	Private functions
	None

Module	Public functions
Temperature sensor	<pre>void temperatureSensorInit(); void temperatureSensorUpdate(); float temperatureSensorReadCelsius(); float temperatureSensorReadFahrenheit(); float celsiusToFahrenheit(float tempInCelsiusDegrees);</pre>
	Private functions
	<pre>static float analogReadingScaledWithTheLM35Formula(float analogReading);</pre>

Module	Public functions
User interface	<pre>void userInterfaceInit(); void userInterfaceUpdate(); bool userInterfaceCodeCompleteRead(); void userInterfaceCodeCompleteWrite(bool state); bool incorrectCodeStateRead(); void incorrectCodeStateWrite(bool state); bool systemBlockedStateRead(); void systemBlockedStateWrite(bool state);</pre>
	Private functions
	<pre>static void incorrectCodeIndicatorUpdate(); static void systemBlockedIndicatorUpdate(); static void userInterfaceMatrixKeypadUpdate();</pre>



NOTE: The function `eventLogWrite()` is public even though no other modules use it. The reason it is declared public instead of private is that it will be used by a module that will be created in Example 10.3.

5.4.2 Detailed Implementation of the Code of the Smart Home System in Different Files

In order to implement modularization in C/C++, every module must have a well-defined *interface*. By means of its interface, each module specifies how its public functions can be requested by other modules. It is very important to understand that the modules that call a public function from another module should not get involved in, or even get access to, the way in which those functions are implemented. This concept is known as *encapsulation*. For this purpose, header (.h) files are used. This model is shown in Figure 5.7.

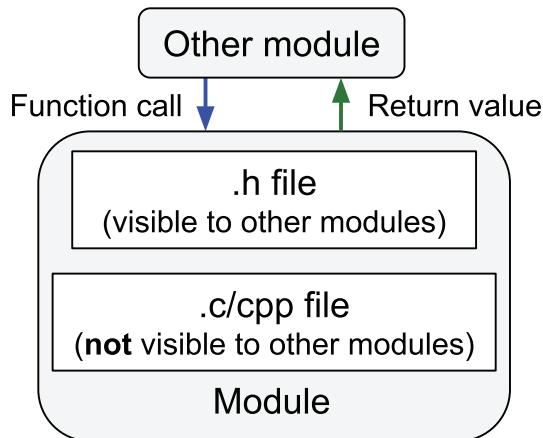


Figure 5.7 Diagram of modularization in C/C++ using header files.

The main goal is to improve the program organization and to guarantee that each module does not get involved in the *responsibilities* of other modules. In order to achieve this goal, it is common to separate the *.h* files and the *.c/cpp* files into different file folders, or even to only grant other programmers access to the *.h* files, and provide the *.c/cpp* files as *object* files, which cannot be read by a programmer.

Another important concept arises here: the *prototypes* of the functions declared in the *.h* file are *public*, while the prototypes of the functions declared as static in the *.c/cpp* file are *private*.

To implement the *.h* and *.cpp* files of each module, some templates that are used are available in subsection 5.4.2 at [1].

The *.h* file template begins with “*#include guards - begin*”, where it uses the preprocessor directive “*#ifndef*” to indicate that the *.h* file must be included only if it was not previously included. This is to ensure that each *.h* file is included only once, because if a *.h* file is included more than once, the compiler will report an error. The line “*#include guards - end*” at the end of the *.h* file template is used to indicate the ending of the preprocessor directive by an *#endif*.

In Table 5.17, the sections of the template that are used to write the *.h* file of each module are shown. The three sections of the *.h* file template (*Declaration of public #defines*, *Declaration of public data types*, and *Declarations (prototypes) of public functions*) are all public declarations. As explained in subsection 5.2.2 *Implementation of Modularization in C/C++ Programs*, these *.h* files are the ones that should be provided to the users of each module.

Table 5.17 Sections of the template that are used to write the *.h* file of each module.

Name of the section	Purpose of the section
Declaration of public defines	Declaration of #defines that are public.
Declaration of public data types	Declaration of data types that are public.
Declarations (prototypes) of public functions	Declaration functions that are public.

In Table 5.18, the sections of the template used to write the .cpp file of each module are shown. The reader should note that this template includes:

- The libraries that are used by the module (i.e., a set of .h files).
- The declaration of definitions, data types, variables, and functions.
- The implementation of public and private functions.

Table 5.18 Sections of the template used to write the .cpp file of each module.

Name of the section	Purpose of the section
Libraries	Include .h files used by the module.
Declaration of private defines	Declare the defines used only by the module.
Declaration of private data types	Declare the data types used only by the module.
Declaration and initialization of public global objects	Declare the objects that are used only by other modules and maybe also by the same module.
Declaration of external public global variables	Declare the <i>extern</i> public global variables (this concept is explained at the end of this section).
Declaration and initialization of public global variables	Declare the variables that are used by other modules and maybe also by the same module.
Declaration and initialization of private global variables	Declare the variables that are used only by the module.
Declarations (prototypes) of private functions	Declare the private functions that are used only by the module.
Implementations of public functions	Implement the public functions used by other modules.
Implementations of private functions	Implement the private functions.



NOTE: The #defines and data types declared in sections *Declaration of private definitions* and *Declaration of private data types* of a .cpp file can be used only by code in the same .cpp file, while #defines and data types in sections *Declaration of public definitions* and *Declaration of public data types* of a .h file are public.

Using these .h and .cpp file templates and the public and private classification of variables and functions discussed in Table 5.15 and Table 5.16, the files and folders shown in Figure 5.8 were prepared. An extra .cpp file for the *main* function and an extra .h file for the *arm_book_lib* library are included.

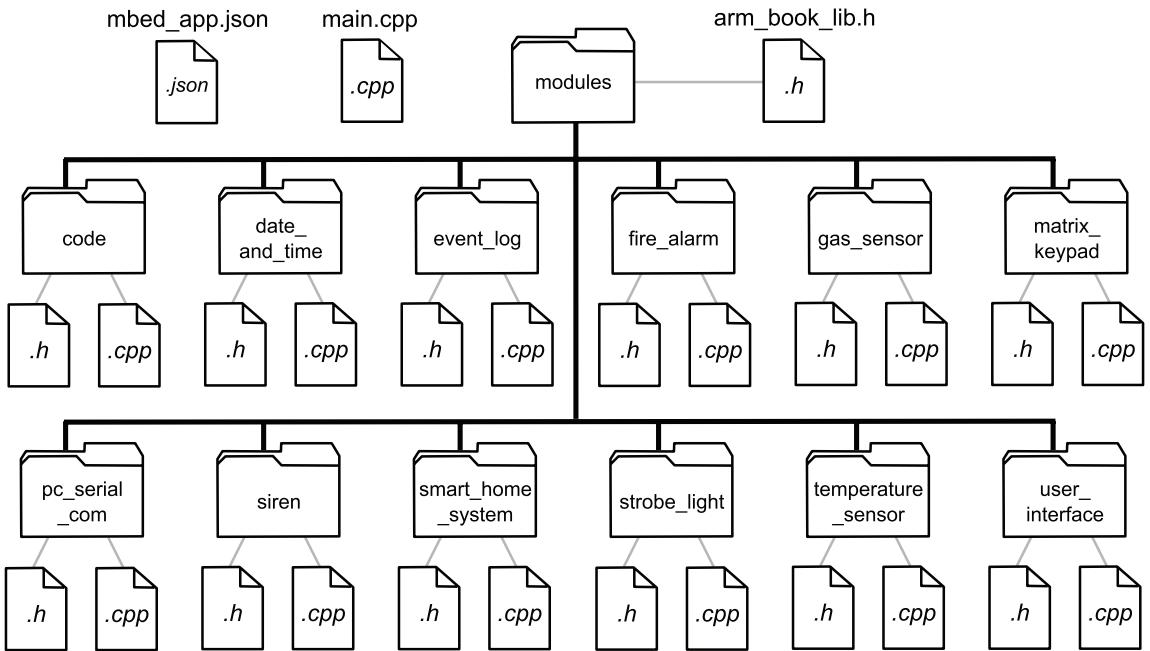


Figure 5.8 File organization proposed for the smart home system code.

It is important to mention that the libraries that are included in each module are just the ones that are needed by that module. For example, the section “Libraries” of the file *siren.cpp* includes the following files: *mbed.h*, *arm_book.lib.h*, *siren.h*, *smart_home_system.h*, and *fire_alarm.h*.

Given that all the functions have already been discussed in subsection 5.3.2, no code will be shown or discussed in this section. The reader is encouraged to download the files and explore the code behavior themselves. All the folders and files shown in Figure 5.8 are available in subsection 5.4.2 at [1]. Even though the project is organized in modules, it is built in the same way as in previous chapters. Drag the *.bin* file onto the NUCLEO board, and test how the program works in order to verify that its functionality is similar to Example 4.4. Some differences are that now if the alarm is activated, the Alarm LED does not stop blinking when the program is waiting for the user to enter the alarm deactivation code (because of the changes introduced in the *pc_serial_com* module), and the list of available commands is shown after power on even if the user does not press any key.

Table 5.19 shows the only two variables that are defined using the prefix *extern*. These two variables, *codeSequenceFromUserInterface* and *codeSequenceFromPcSerialCom*, are declared in the *user_interface* module and in the *pc_serial_com* module, respectively, and are used also in the *code* module. In order to make them usable by other modules, they are declared as public variables in the section “Declaration and initialization of public global variables” of the *user_interface* and *pc_serial_com* modules, respectively, and due to the *extern* prefix, the compiler is notified that these two variables, which are referred to in the section “Declaration of external public global variables” of the *code* module, are actually declared somewhere else.

Table 5.19 Example of extern variables that are used in the implementation of the smart home system.

Module	External public global variables
code	extern char codeSequenceFromUserInterface[CODE_NUMBER_OF_KEYS]; extern char codeSequenceFromPcSerialCom[CODE_NUMBER_OF_KEYS];

This concept of extern variables applies whenever a global variable must be used in different modules. Therefore, in that case it is not valid to declare the global variable only in one of the modules and use it in the other modules, because an error of “use of undeclared identifier” will be obtained. Neither is it valid to declare the global variable in each of the modules, because in that case an error of “multiply defined” will be obtained. As explained above, in that situation, the global variable must be declared in one of the modules as usual, and in the other modules it should be declared using the reserved word *extern*, as shown in Table 5.19.

Lastly, it is very important to note that as a consequence of the modularization process, some libraries were created that can be reused in other systems. For example, the *temperature_sensor* module can be used in any other system provided with an LM35 sensor. The *matrix_keypad* module could also be reused by the reader in future projects.



NOTE: This book started from a non-modular design for pedagogical reasons. It is always recommended to start with a modularized design from the beginning.

Proposed Exercises

1. What should be modified in order to change the blinking time of the siren?
2. What should be modified in order to change the conversion formula of the temperature sensor?

Answers to the Exercises

1. The values of the #defines STROBE_TIME_GAS, STROBE_TIME_OVER_TEMP, and STROBE_TIME_GAS_AND_OVER_TEMP should be modified in the file *fire_alarm.cpp*.
2. The conversion formula should be modified in the function *analogReadingScaledWithTheLM35Formula()* of the *temperature_sensor* module.

References

- [1] “GitHub - armBookCodeExamples/Directory”. Accessed July 9, 2021.
<https://github.com/armBookCodeExamples/Directory/>
- [2] “<cstring> (string.h) - C++ Reference”. Accessed July 9, 2021.
<https://www.cplusplus.com/reference/cstring/>

Chapter 6

LCD Displays and Communication
between Integrated Circuits

6.1 Roadmap

6.1.1 What You Will Learn

After you have studied the material in this chapter, you will be able to:

- Explain and compare the characteristics of the most commonly used buses for connecting integrated circuits.
- Describe how to connect an LCD display to the NUCLEO board using GPIOs, I2C, and SPI buses.
- Develop programs to show information as characters and graphics on suitable LCD displays.
- Summarize the concept of a hardware abstraction layer (HAL).



DEFINITION: A typical definition of *bus* in computer architecture is a communication system that transfers data between components inside a computer. The term covers all related hardware components and software, including communication protocols. Buses can have parallel or serial wired connections (i.e., not wireless).

DEFINITION: A communication protocol is a system of rules that allows two or more entities to transmit information via any kind of variation of a physical quantity. The protocol defines the rules, syntax, semantics, and synchronization of communication and possible error recovery methods.

6.1.2 Review of Previous Chapters

In previous chapters, different sensors and elements were connected to the NUCLEO board using GPIOs and analog inputs. The NUCLEO board was connected to a PC using serial communication implemented through a UART. In this way, the information gathered by means of the sensors was shown on the PC screen using the serial terminal.

6.1.3 Contents of This Chapter

It is not always feasible or possible to use a PC to show information, due to room and cost limitations; an LCD display can be more convenient. There are also modules whose interface is neither a set of GPIO pins nor an analog output, but a serial communication protocol based on something other than the UART technology introduced in previous chapters.

In this chapter, two different types of LCD displays will be connected to the NUCLEO board: character displays and graphical displays. The former is able to display only characters, while the latter is able to display graphics as well as characters. The character LCD display will be connected to the NUCLEO board by means of GPIOs (General Purpose Input Output pins) and the I2C (Inter-Integrated Circuit) bus, while the graphical LCD display connection will be made using the SPI (Serial Peripheral Interface) bus.

The aim is to introduce most of the buses used to connect integrated circuits and sensors, and through examples show how software implementation can be made independent from the hardware details. This will be explained by means of a software module that displays information on an LCD display, regardless of whether the LCD is a character or a graphical display, or whether the connection to the display is by means of GPIOs, the I2C bus, or the SPI bus. In this way, the concept of a *hardware abstraction layer* (HAL) will be introduced.



NOTE: In previous chapters, the details of the connections to be made and the explanation of the technical concepts that were used in the examples were all located at the beginning of the chapter. In this chapter, these sections are interleaved with the examples as the same character display is connected to the NUCLEO board in different ways, using different technologies. In this chapter, the explanations of the technologies involved have a higher level of detail than in the previous chapters. This level of detail is necessary to understand how LCD displays can be controlled using I2C and SPI buses.

6.2 LCD Display Connection using GPIOs, I2C, and SPI Buses

6.2.1 Connect a Character LCD Display to the Smart Home System using GPIOs

In this chapter, an LCD display is connected to the smart home system, as shown in Figure 6.1. In this way, it is possible to show in the Alarm control panel information regarding temperature reading, as well as the state of the gas detector and the activation of the alarm.

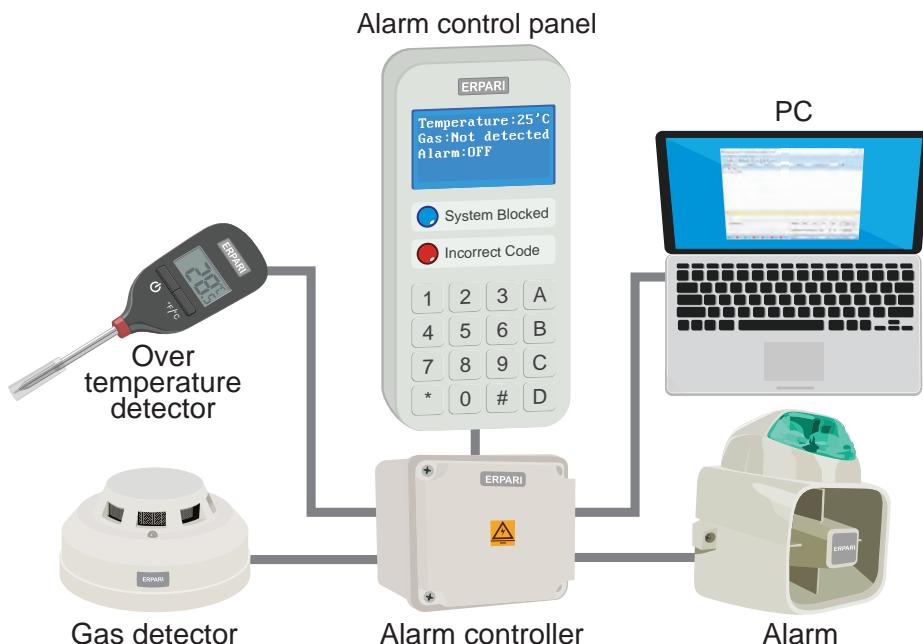


Figure 6.1 The smart home system is now connected to an LCD display.

Figure 6.2 shows how to connect the character LCD display module [1], which is based on the HD44780 dot matrix liquid crystal display (LCD) controller [2]. The reader may notice that standard inputs and outputs of the NUCLEO board, usually called GPIOs (General Purpose Input Output), are used, as summarized in Figure 6.3. The aim of this setup is to introduce the basics of character LCD displays.

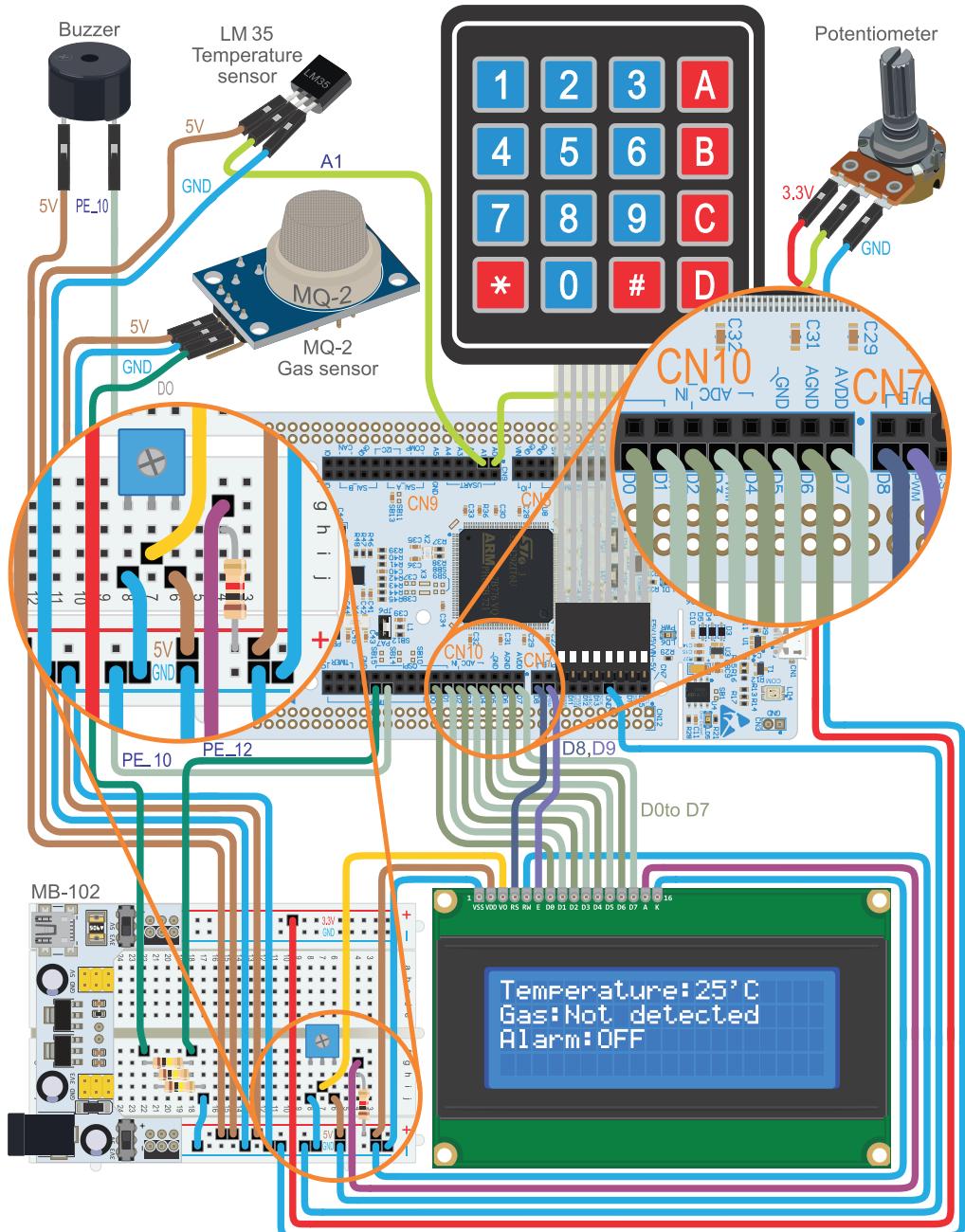


Figure 6.2 The smart home system connected to the character LCD display using GPIOs.



WARNING: Some displays have a different pin layout. In that case make the connections following the pin names indicated in Figure 6.3, even if the pins are arranged in a different way on the display.

In Figure 6.3, the connections that must be made are shown. The GPIOs D0–D9 are used to send commands to the LCD display, as discussed below. The contrast of the character LCD display can be adjusted by means of the trimmer potentiometer or “trimpot.” The 1 k Ω resistor connected to the A (anode) pin together with the K (cathode) pin are used to power the backlight of the character LCD display.

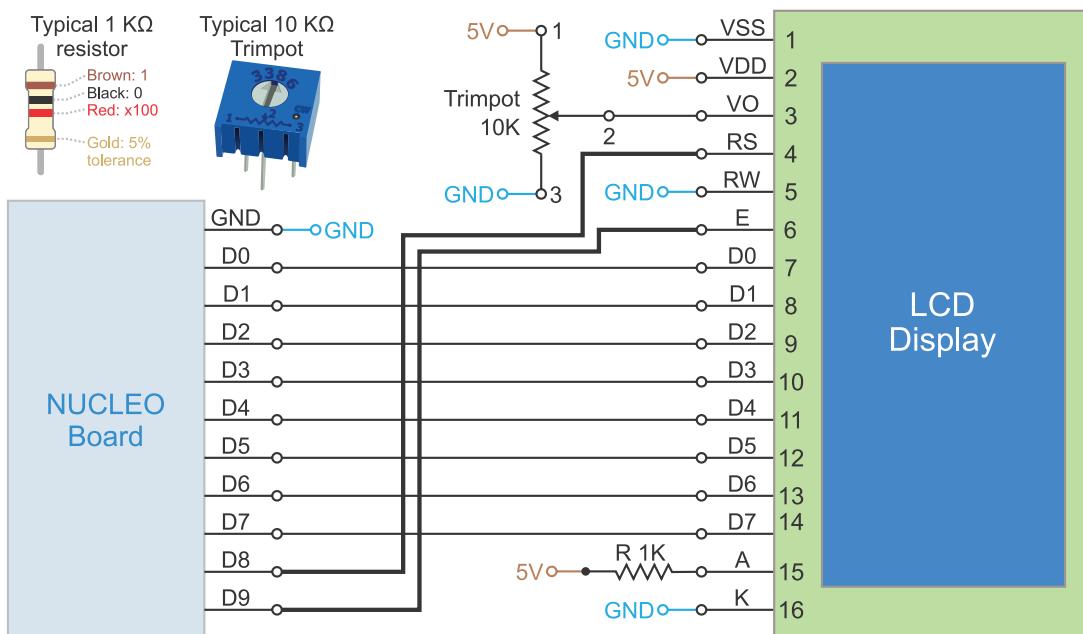


Figure 6.3 Diagram of the connections between the character LCD display and the NUCLEO board using GPIOs.



WARNING: Some displays require different resistor values. Check the datasheet of the display you use.

To test if the character LCD display is working, the .bin file of the program “Subsection 6.2.1” should be downloaded from the URL available in [3] and dragged onto the NUCLEO board. After power on, the most pertinent information from the smart home system should be shown on the character LCD display, as in Figure 6.2.



NOTE: As in previous chapters, the code that is provided to test the connections will not be discussed. The code to control the LCD character and graphical displays will be explained in detail in the examples.

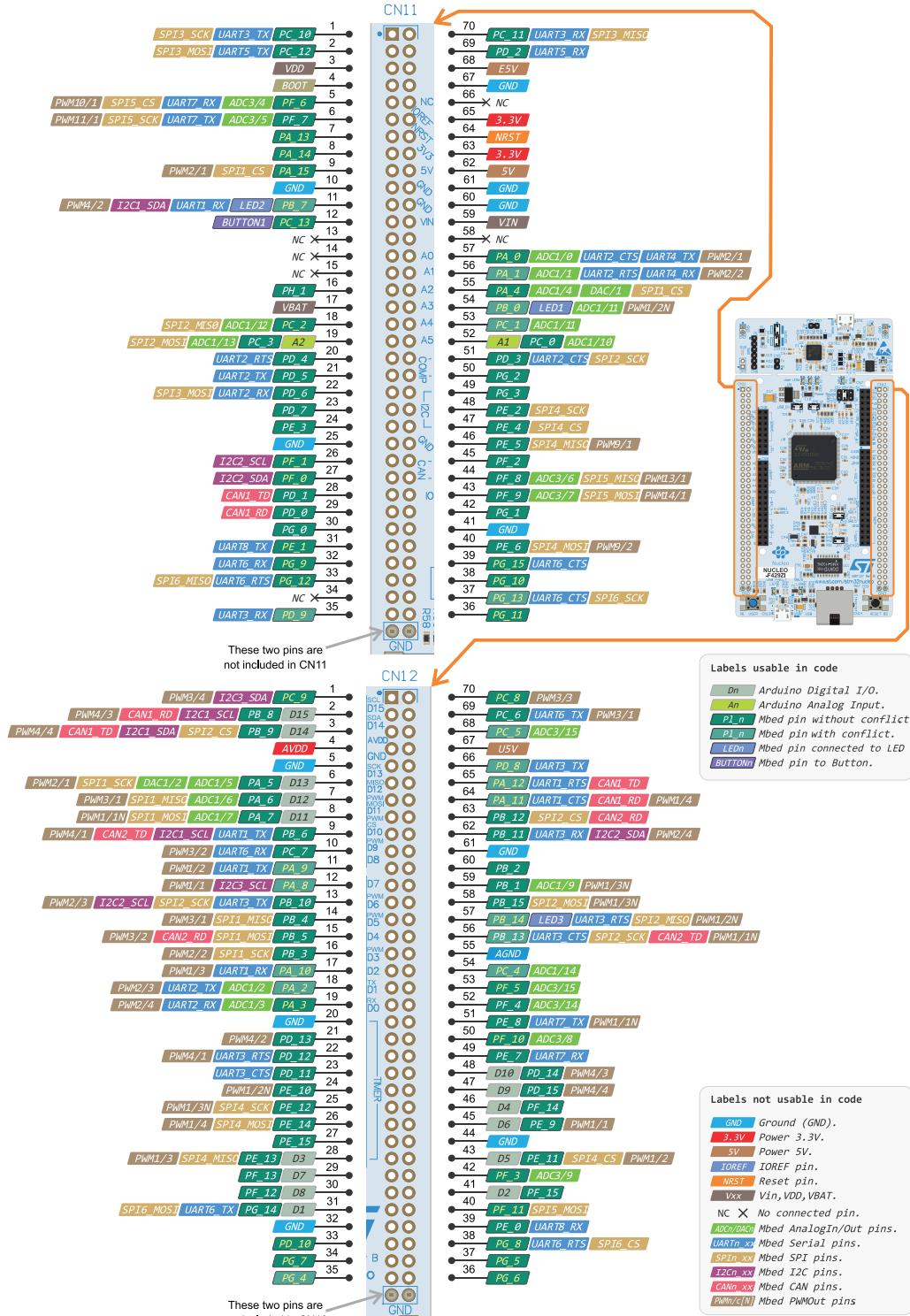


Figure 6.4 CN11 and CN12 headers of the NUCLEO-F429ZI board scheme made using information available from [4].



TIP: If the display is not working as expected, the corresponding connections can be checked using the CN11 and CN12 headers and a multimeter. For example, in Figure 6.3 it can be seen that pin D8 of the NUCLEO board (which corresponds to PF_12, as can be seen in [4]) should be connected by means of a wire to pin RS of the character LCD display. However, once this wire is connected it is not easy to access the D8 pin of the CN7 header to make a continuity test using a multimeter. Nevertheless, PF_12 is also available on the CN12 connector, as can be seen in Figure 6.4 (adapted from [4]). Hence, a continuity test can be made between PF_12 and pin RS of the character LCD display by placing one probe at the pin corresponding to PF_12 in the CN12 header and placing the other probe on pin RS of the character LCD display. If they are continuous, and the multimeter is properly configured, it beeps and displays a value near zero. This procedure can be used to check all the connections shown on Figure 6.3.



WARNING: In strict terms, the display must be controlled by signals that have a high level of at least $0.7 \times 5\text{ V} = 3.5\text{ V}$, as indicated in [2], while the expected high level of the NUCLEO board digital outputs is about 3.3 V. However, in order to avoid the usage of many voltage-level converters, the character display is connected directly because it was proven to work without the voltage converters. Section 6.2.5 shows an example of how voltage converters can be used to adapt voltage levels when it is necessary.

6.2.2 Basic Principles of Character LCD Displays

Character displays, such as the one used in this chapter, are available in a range of different layouts, such as 20×4 (4 lines of 20 characters), 16×2 , 8×2 , 8×1 , etc. Each character is displayed on a 5×8 pixel matrix (the 8th pixel line is reserved for the cursor) as shown in Table 6.1. The code corresponding to each character is obtained by adding the row and column values. For example, the character "A" is in the intersection of the column labeled 64 and the row labeled 1, so its code is 65. The character "a" is in the intersection of the column labeled 96 with the row labeled 1, so its code is 97. The characters corresponding to the first eight codes (0 to 7) in Table 6.1 correspond to the *Custom Generated Random-Access Memory* (CGRAM) characters that can be defined pixel-by-pixel by the user. The next eight codes (8 to 15) are also mapped to the same eight CGRAM characters and are not shown in Table 6.1. CGRAM characters are not covered in this book. Characters corresponding to codes 16 to 31 and 128 to 159 are not included in Table 6.1 because they may vary between different versions of the HD44780 dot matrix LCD controller, as can be seen in [2].

Table 6.1 A typical character set of an LCD character display.

	0	32	40	48	56	64	72	80	88	96	104	112	120	160	168	176	184	192	200	208	216	224	232	240	248
0	CG RAM (0)	(8	a	H	P	X	^	h	P	X	×	-	ワ	ネ	タ	ミ	リ	ヨ	ル	メ	ソ	メ	ソ	
1	CG RAM (1))	1	9	A	I	Q	y	a	i	q	y	。	ア	ケ	リ	チ	カ	ハ	ー	カ	ウ	ー	カ	
2	CG RAM (2)	*	*	2	:	B	J	R	Z	b	J	R	z	エ	イ	コ	ハ	ツ	レ	ビ	イ	8	フ	シ	
3	CG RAM (3)	#	+	3	:	C	K	S	L	c	K	S	l	オ	ウ	サ	ヒ	テ	モ	ロ	ス	×	・	フ	
4	CG RAM (4)	,	,	4	<	D	L	T	Y	d	I	t	।	、	エ	シ	フ	ト	ア	ワ	レ	フ	৳	৳	
5	CG RAM (5)	-	-	5	=	E	M	U	J	e	m	u	j	ュ	オ	ス	ヘ	ナ	コ	ン	স	ট	৳	৳	
6	CG RAM (6)	8	.	6	>	F	N	V	^	f	n	v	~	়	ৰ	ৱ	ৱ	ৱ	ৱ	ৱ	ৱ	ৱ	ৱ	ৱ	ৱ
7	CG RAM (7)	/	/	7	?	G	O	W	-	g	o	w		়	-	・	়	়	়	়	়	়	়	়	়

In Chapter 2, it was mentioned that characters transferred between the PC and the NUCLEO board (for example, 'H', 'e', 'I', 'l', and 'o') are codified using the ASCII standard (*American Standard Code for Information Interchange*), which is described in [5]. ASCII was created in the 1960s, having 128 characters. Of these, 95 are printable (digits 0 to 9, lowercase letters a to z, uppercase letters A to Z, punctuation symbols, etc.). The other 33 are non-printing control codes, most of which are now obsolete, although a few are still commonly used, such as the carriage return (\r), line feed (\n), and tab codes (\t).

While 95 printable ASCII characters are sufficient in English, other languages that use Latin alphabets need additional symbols. ISO/IEC 8859 sought to solve this problem using the eighth bit in an 8-bit byte to allow positions for another 96 printable characters. In Table 6.2, some of the corresponding characters are shown. The reader may notice its similarity to the character set shown in Table 6.1. Therefore, in this chapter, characters to be sent to the display are stated in the program code in a similar way to previous chapters. However this may not be valid in some cases, where Table 6.1 and Table 6.2 may differ.

Table 6.2 Part of the character set defined by ASCII and ISO/IEC 8859.

	0	32	40	48	56	64	72	80	88	96	104	112	120	160	168	176	184	192	200	208	216	224	232	240	248
0	\0	space	(0	8	@	H	P	X	`	h	p	x	-	°	,	À	È	Ð	Ø	à	è	ð	ø	
1	!)	1	9	A	I	Q	Y	a	i	q	y	í	©	±	¹	Á	É	Ñ	Ü	á	é	ñ	ü	
2	"	*	2	:	B	J	R	Z	b	j	r	z	¢	³	²	º	Â	Ê	Ò	Ú	â	ê	ò	ú	
3	#	+	3	;	C	K	S	[c	k	s	{	£	«	³	»	Ã	Ë	Ó	Ù	ã	ë	ó	ù	
4	\$,	4	<	D	L	T	\	d	l	t		¤	¬	'	¼	Ä	Ì	Ó	Ü	ä	ì	ö	ü	
5	%	-	5	=	E	M	U]	e	m	u	}	¥	µ	½	Å	Í	Ó	Ý	å	í	ö	ý		
6	&	.	6	>	F	N	V	^	f	n	v	~	¡	®	¶	¾	Æ	Ï	Ö	Þ	æ	í	ö	þ	
7	'	/	7	?	G	O	W	-	g	o	w		§	-	·	¿	Ҫ	Ӯ	Х	Ծ	Ӯ	÷	Ӷ		



NOTE: For space reasons, other character mappings are not analyzed in this book.

In an LCD character display, there is *Display Data Random Access Memory* (DDRAM), which stores the character to be displayed in each position of the LCD display. In a 20×4 character LCD display, the code of the character to be displayed in the first position of the first line is written into address 0 of the DDRAM, the character to be displayed in the second position of the first line is written in address 1, and so on. The address of each position of the 20×4 character LCD display is shown in Figure 6.5.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103

Figure 6.5 Addresses corresponding to each of the positions of a 20×4 LCD character display.

From Figure 6.6 to Figure 6.8, the addresses of each of the positions of some typical LCD character display layouts are shown. The reader will notice that there are more addresses than characters that can be shown on the display. To show the characters that are written in those addresses (for example, 16 and 80 in Figure 6.6), a *shift* instruction is used. This idea is illustrated in Figure 6.9.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	39
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	...	103

Figure 6.6 Addresses corresponding to each of the positions of a 16×2 LCD character display.

0	1	2	3	4	5	6	7
64	65	66	67	68	69	70	71

8	9	...	39
72	73	...	103

Figure 6.7 Addresses corresponding to each of the positions of an 8×2 LCD character display.

0	1	2	3	4	5	6	7
8	9	...	79				

Figure 6.8 Addresses corresponding to each of the positions of an 8×1 LCD character display.

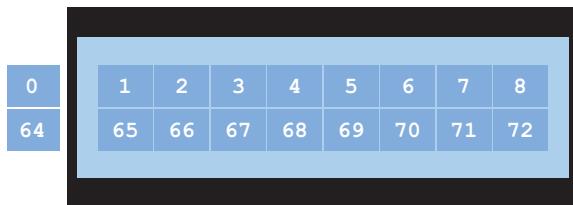


Figure 6.9 An 8×2 LCD character display where a left shift has been applied once.



NOTE: The number 64 is written in binary notation as 01000000. Therefore, it makes sense to use the number 64 for the position 0 of the second line considering that "01" indicates line 2 and "000000" its first position.

NOTE: Usually DDRAM addresses are expressed in datasheets in hexadecimal notation. Therefore, the number "10" is indicated as "0A", "11" is "0B", ..., "20" is "14", ..., "64" is "40", ..., "84" is "54", etc. In this book, decimal notation is used in order to make addresses easier to understand for the reader.

The instructions that are used in this chapter are summarized in Table 6.3. These instructions are sent to the display following the timing diagram shown in Figure 6.10. First, the states of the pins E (Enable), RS (Register Select), R/W (Read, Write), and DB7 to DB0 (Data Bus) are established. Then, a pulse is set into the E pin (it should last at least 1 μ s). During the falling edge of the E pin the code is written: if RS is set to low, the code is written into the *instruction register*, while if RS is set to high, the code is written into the *data register*. These registers are internally used by the HD44780 dot matrix LCD controller to process the codes received from the microcontroller [2].

Table 6.3 Summary of the character LCD display instructions that are used in this chapter.

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift.	1.52 ms
Display control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor (B).	37 μ s
Function set	0	0	0	0	1	DL	N	F	*	*	Sets interface data length (DL), number of display lines (N), and character font (F).	37 μ s
Set DDRAM address	0	0	1	A6	A5	A4	A3	A2	A1	A0	Sets DDRAM address.	37 μ s

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Write data to DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Writes data into DDRAM.	37 µs
I/D = 1: Increment, I/D = 0: Decrement S = 1: Accompany display shift, S = 0: Don't accompany display shift, D = 1: Display on, D = 0: Display off C = 1: Cursor on, C = 0: Cursor off B = 1: Cursor blink on, B = 0: Cursor blink off												
DL = 1: 8 bits, DL = 0: 4 bits N = 1: 2 lines, N = 0: 1 line F = 1: 5 × 10 dots, F = 0: 5 × 8 dots * = don't care A6 ... A0 = Address, D7 ... D0 = Data												

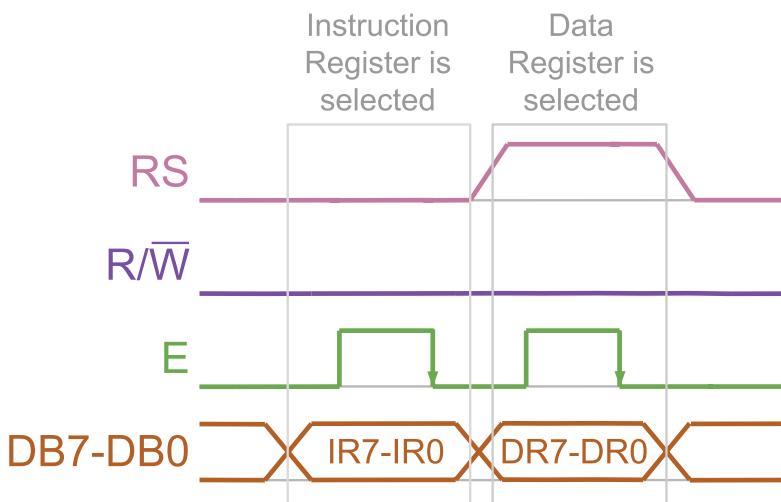


Figure 6.10 Transfer timing sequence of writing instructions when an 8-bit interface is configured.



NOTE: In the examples in this book, the R/W pin is connected to GND because only write operations are made to the registers. Sometimes it might be necessary to read a register to confirm if the previous instruction sent to the display was successfully executed, but this is not the case in this book.

The procedure to initialize the display when an 8-bit interface is used is described in [2] and is shown in Figure 6.11. First, there should be a waiting period of more than 40 milliseconds after power on. Then, the “Function Set” instruction should be sent four times with different delays in between and with DB4 (corresponding to the DL, Data Length configuration) set to 1. The first three times “Function Set” is sent, the values of DB3 to DB0 do not matter (those bits can be set either to 1 or 0), while the fourth time “Function Set” is sent, the values of N (number of lines in the display) and F (font size) must be set. In the case of the 20 × 4 character display, N must be set to 1 (2 lines) and F must be set to 0 (5 × 8 dots).

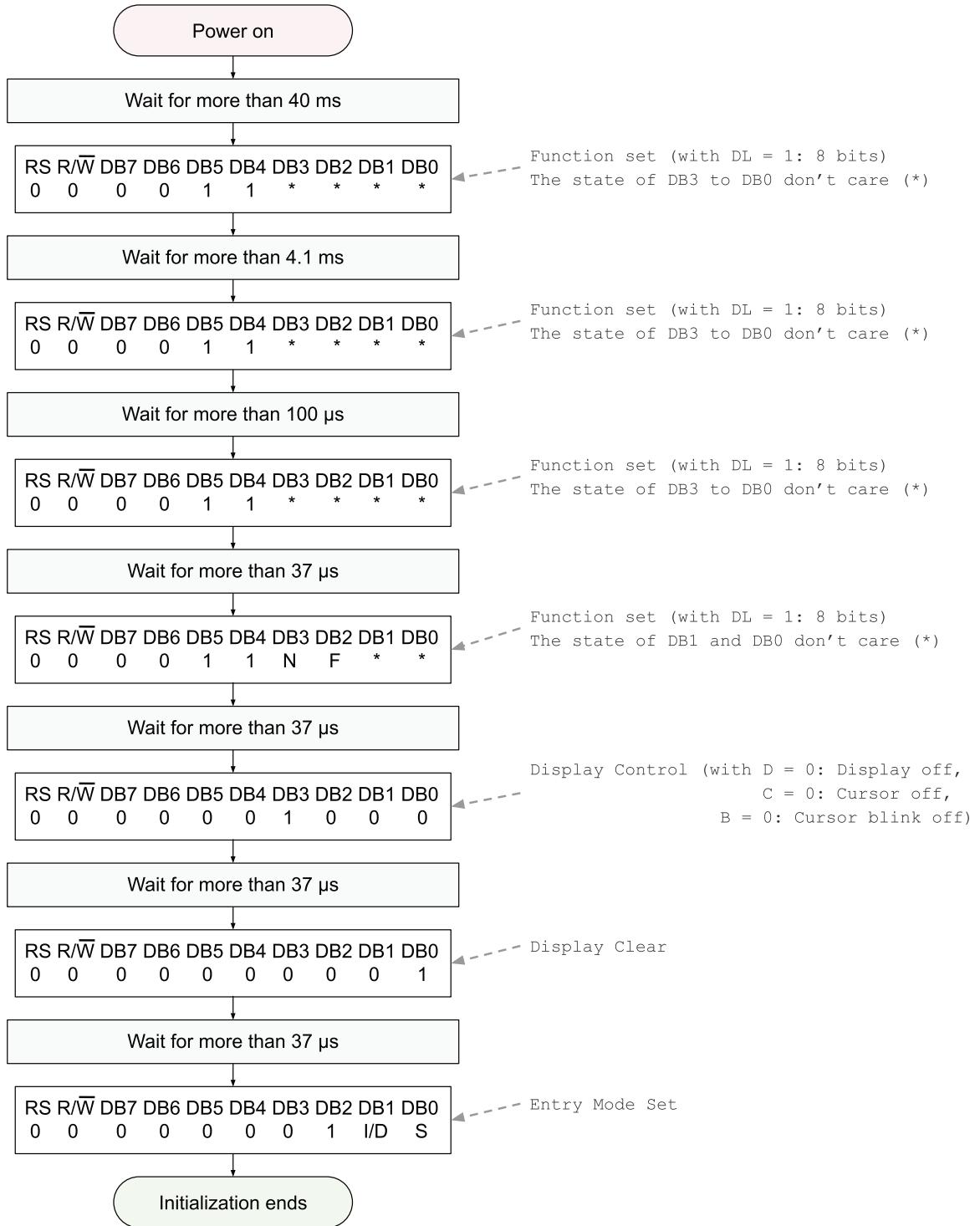


Figure 6.11 Initialization procedure of the graphic display when an 8-bit interface is used.



NOTE: The addresses of a 20×2 LCD character display are very similar to those shown in Figure 6.6. In a 20×2 character display, the addresses of line 1 range from 0 to 39 and the addresses of line 2 from 64 to 103. Only addresses 0 to 19 and 64 to 83 are visible without a shift instruction. Addresses 20 to 39 and 84 to 103 can be seen only if a shift is made. A 20×4 display (Figure 6.5) is considered a special case of a 20×2 display, where all the DDRAM content is distributed in four lines and is shown at the same time. For this reason, N is set to two lines in a 20×4 character display.

In Figure 6.11, it can be seen that the “Display Control” instruction is followed by D = 0 (Display off), C = 0 (Cursor off), and B = 0 (Cursor blink off). This is followed by the “Display Clear” instruction. Lastly, the “Entry Mode Set” instruction is set, where the value of I/D and S can be configured according to programmer preference. In this book, I/D is configured to 1 (Increment), in order to automatically increment the DDRAM address immediately after a given character is written into the display, and S is set to 0, because it is not necessary to shift the display. In this way, the initialization ends, and messages can be shown on the display.

Typically, to write a message on the display, the “Set DDRAM address” instruction is used to indicate the position of the first letter of the message. Then the “Write data to DDRAM” instruction is used to write the corresponding character according to Table 6.1. Given that I/D is configured to 1, the next character of the message can be sent to the display using the “Write data to DDRAM” instruction, without the need to increment the DDRAM address by means of the “Set DDRAM address” instruction. In this way, the characters of the message can easily be written one after the other, as shown in Example 6.1.

Example 6.1: Indicate Present Temperature, Gas Detection, and Alarm on the Display

Objective

Introduce the usage of a character-based LCD display by means of a GPIO connection.

Summary of the Expected Behavior

The present temperature is shown on the first line of the character LCD display, the state of the gas detector is shown on the second line, and the state of the alarm is shown on the third line. The fourth line is left empty to reserve space so that in the future more information can be shown on the display.

Test the Proposed Solution on the Board

Import the project “Example 6.1” using the URL available in [3], build the project, and drag the .bin file onto the NUCLEO board. The present temperature, the gas detection state, and the alarm state should be shown on the display. Hold the temperature sensor between two fingers in order to change its reading. The corresponding value should be displayed on the first line of the display. Activate the alarm by pressing the Alarm test button. This condition should be indicated on the display. When the alarm is turned off (use the same steps as in the previous chapters), its state should be updated on the display.

Discussion of the Proposed Solution

The proposed solution is based on a new software module named *display*. This new module is composed of two files, *display.cpp* and *display.h*, following the modularized structure discussed in the previous chapter. The *main()* function in the *main.cpp* remains the same as in the last section of Chapter 5. Furthermore, the functions *smartHomeSystemInit()* and *smartHomeSystemUpdate()*, called from the *main()* function, have no changes. Those functions call the functions *userInterfaceInit()* and *userInterfaceUpdate()*, respectively, and these are the ones that make the corresponding calls to the functions of the *display* module, as detailed below.

Implementation of the Proposed Solution

In Table 6.4, the sections where lines have been added to the file *user_interface.cpp* are summarized. Besides the definition of *DISPLAY_REFRESH_TIME_MS* and the declaration of the two functions that will be explained below, it should be noted that *fire_alarm.h* and *display.h* have been included.

Table 6.4 Sections in which lines were added to user_interface.cpp.

Section or function	Lines that were added
Libraries	#include "fire_alarm.h" #include "display.h"
Definitions	#define DISPLAY_REFRESH_TIME_MS 1000
Declarations (prototypes) of private functions	static void userInterfaceDisplayInit(); static void userInterfaceDisplayUpdate();

As previously mentioned, the functions *userInterfaceInit()* and *userInterfaceUpdate()* are modified in order to include the initialization and update of the display, respectively, as can be seen in Code 6.1 and Code 6.2.

```

1 void userInterfaceInit()
2 {
3     incorrectCodeLed = OFF;
4     systemBlockedLed = OFF;
5     matrixKeypadInit( SYSTEM_TIME_INCREMENT_MS );
6     userInterfaceDisplayInit();
7 }
```

Code 6.1 New implementation of the function userInterfaceInit(), including userInterfaceDisplayInit().

```

1 void userInterfaceUpdate()
2 {
3     userInterfaceMatrixKeypadUpdate();
4     incorrectCodeIndicatorUpdate();
5     systemBlockedIndicatorUpdate();
6     userInterfaceDisplayUpdate();
7 }
8
```

Code 6.2 New implementation of the function userInterfaceUpdate(), including userInterfaceDisplayUpdate().

The implementation of the function `userInterfaceDisplayInit()` is shown in Code 6.3. It is declared as a private function by means of the reserved word `static` on line 1 of the corresponding code. On line 3, the display is initialized by means of the function `displayInit()`. The details of the implementation of `displayInit()` will be shown below this example.

In Code 6.3, the functions `displayCharPositionWrite()` and `displayStringWrite()` are used to move the cursor to a given position and write a given string in that position. In this way, the strings “Temperature”, “Gas”, and “Alarm” are written in specific positions of the display. The way in which the corresponding positions of those strings are indicated using x and y coordinates is shown in Figure 6.12. These coordinates are the parameters of `displayCharPositionWrite()`, as discussed below. The details of the implementation of `displayStringWrite()` are also discussed below.



NOTE: The messages in lines 6, 9, and 12 are strings because the '\0' (null character) is automatically added to the end of an array of char when it is written between quotes, as, for example, in “Temperature:”.

```

1 static void userInterfaceDisplayInit()
2 {
3     displayInit();
4
5     displayCharPositionWrite ( 0,0 );
6     displayStringWrite( "Temperature:" );
7
8     displayCharPositionWrite ( 0,1 );
9     displayStringWrite( "Gas:" );
10
11    displayCharPositionWrite ( 0,2 );
12    displayStringWrite( "Alarm:" );
13 }
```

Code 6.3 Implementation of the function `userInterfaceDisplayInit()`.

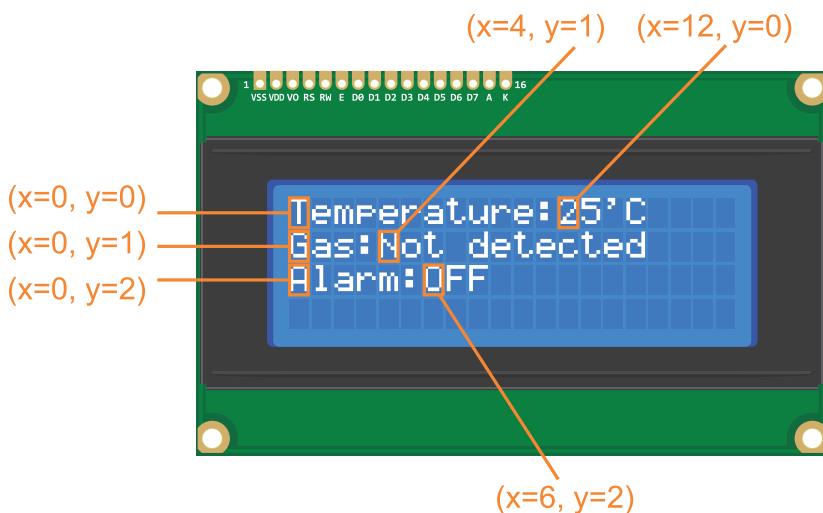


Figure 6.12 Position of the strings that are placed in the character LCD display.

The implementation of the function `userInterfaceDisplayUpdate()` is shown in Code 6.4. On lines 3 and 4, two variables are declared: a static int `accumulatedDisplayTime` that is initialized to 0, and a char array `temperatureString`, which has three positions because it is considered that the temperature will be in the range of 0 to 40 °C (a third char position is reserved for the null character). On line 6, a check is made whether `accumulatedDisplayTime` has reached the value established by `DISPLAY_REFRESH_TIME_MS`, which is defined in `user_interface.cpp` as 1000. If so, the present temperature value, the state of the gas detector, and the state of the alarm are updated on the display by means of the code on lines 6 to 31. In this way, the display is updated every 1000 ms. The corresponding (x, y) positions of the temperature value, the gas detection, and the alarm state can be seen in Figure 6.12. Finally, if `accumulatedDisplayTime` is lower than `DISPLAY_REFRESH_TIME_MS`, then the value of `accumulatedDisplayTime` is incremented by `SYSTEM_TIME_INCREMENT_MS` (lines 33 to 36).



NOTE: The function `sprintf`, used in line 11 of Code 6.4, was introduced in Chapter 4. This function creates a string (an array of char ending with a null character, '\0') in the indicated destination (in this case `temperatureString`), following the indicated format (in this case "%.0f", a float displayed without decimals) using as input the stated value (the return value of the function `temperatureSensorReadCelsius()`).

```

1  static void userInterfaceDisplayUpdate()
2  {
3      static int accumulatedDisplayTime = 0;
4      char temperatureString[3] = "";
5
6      if( accumulatedDisplayTime >=
7          DISPLAY_REFRESH_TIME_MS ) {
8
9          accumulatedDisplayTime = 0;
10
11         sprintf(temperatureString, "%.0f", temperatureSensorReadCelsius());
12         displayCharPositionWrite ( 12,0 );
13         displayStringWrite( temperatureString );
14         displayCharPositionWrite ( 14,0 );
15         displayStringWrite( "'C" );
16
17         displayCharPositionWrite ( 4,1 );
18
19         if ( gasDetectorStateRead() ) {
20             displayStringWrite( "Detected      " );
21         } else {
22             displayStringWrite( "Not Detected" );
23         }
24
25         displayCharPositionWrite ( 6,2 );
26
27         if ( sirenStateRead() ) {
28             displayStringWrite( "ON      " );
29         } else {
30             displayStringWrite( "OFF" );
31         }
32
33     } else {
34         accumulatedDisplayTime =
35             accumulatedDisplayTime + SYSTEM_TIME_INCREMENT_MS;
36     }
37 }
```

Code 6.4 Implementation of the function `userInterfaceDisplayUpdate()`.

The implementation of the function *displayInit()* is shown in Code 6.5. It follows the initialization procedure that was introduced in Figure 6.11. In line 3, there is a 50-millisecond delay in order to have a safety margin above the 40 millisecond wait after power on. In line 5, *displayCodeWrite()* is used to send the first “Function Set” instruction to the display. The implementation of this function is discussed below, but it can be seen that the first parameter (*DISPLAY_RS_INSTRUCTION*) is used to indicate that the code corresponds to an instruction, while the second parameter indicates that it is a *DISPLAY_IR_FUNCTION_SET* instruction, and the third parameter (*DISPLAY_IR_FUNCTION_SET_8BITS*) indicates that the 8-bit interface bit is set.

The statements between lines 8 and 42 follow the steps indicated in Figure 6.11. The only difference is that at the end the display is turned on (lines 44 to 49).

```

1 void displayInit()
2 {
3     delay( 50 );
4
5     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
6                       DISPLAY_IR_FUNCTION_SET |
7                           DISPLAY_IR_FUNCTION_SET_8BITS );
8     delay( 5 );
9
10    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
11                      DISPLAY_IR_FUNCTION_SET |
12                          DISPLAY_IR_FUNCTION_SET_8BITS );
13    delay( 1 );
14
15    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
16                      DISPLAY_IR_FUNCTION_SET |
17                          DISPLAY_IR_FUNCTION_SET_8BITS );
18    delay( 1 );
19
20    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
21                      DISPLAY_IR_FUNCTION_SET |
22                          DISPLAY_IR_FUNCTION_SET_8BITS |
23                          DISPLAY_IR_FUNCTION_SET_2LINES |
24                          DISPLAY_IR_FUNCTION_SET_5x8DOTS );
25    delay( 1 );
26
27    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
28                      DISPLAY_IR_DISPLAY_CONTROL |
29                          DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_OFF |
30                          DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
31                          DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
32    delay( 1 );
33
34    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
35                      DISPLAY_IR_CLEAR_DISPLAY );
36    delay( 1 );
37
38    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
39                      DISPLAY_IR_ENTRY_MODE_SET |
40                          DISPLAY_IR_ENTRY_MODE_SET_INCREMENT |
41                          DISPLAY_IR_ENTRY_MODE_SET_NO_SHIFT );
42    delay( 1 );
43
44    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
45                      DISPLAY_IR_DISPLAY_CONTROL |
46                          DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_ON |
47                          DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
48                          DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
49    delay( 1 );
50 }
```

Code 6.5 Implementation of the function *displayInit()*.

Code 6.6 shows the #defines that are used by *displayCodeWrite()*. The corresponding values follow the information that was summarized in Table 6.3. The OR bitwise operator (|) is used to set the values of the corresponding bits of the code. For example, “DISPLAY_IR_FUNCTION_SET | DISPLAY_IR_FUNCTION_SET_8BITS” implies the OR bitwise operator between the binary values 0b00100000 and 0b00010000, which is equal to 0b00110000. This value corresponds to the first value that should be sent over the data bus after power on, according to Figure 6.11.

```

1 #define DISPLAY_IR_CLEAR_DISPLAY      0b00000001
2 #define DISPLAY_IR_ENTRY_MODE_SET    0b00000100
3 #define DISPLAY_IR_DISPLAY_CONTROL   0b00001000
4 #define DISPLAY_IR_FUNCTION_SET     0b00100000
5 #define DISPLAY_IR_SET_DDRAM_ADDR   0b10000000
6
7 #define DISPLAY_IR_ENTRY_MODE_SET_INCREMENT 0b00000010
8 #define DISPLAY_IR_ENTRY_MODE_SET_DECREMENT 0b00000000
9 #define DISPLAY_IR_ENTRY_MODE_SET_SHIFT    0b00000001
10 #define DISPLAY_IR_ENTRY_MODE_SET_NO_SHIFT 0b00000000
11
12 #define DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_ON 0b00000100
13 #define DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_OFF 0b00000000
14 #define DISPLAY_IR_DISPLAY_CONTROL_CURSOR_ON 0b00000010
15 #define DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF 0b00000000
16 #define DISPLAY_IR_DISPLAY_CONTROL_BLINK_ON 0b00000001
17 #define DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF 0b00000000
18
19 #define DISPLAY_IR_FUNCTION_SET_8BITS    0b00010000
20 #define DISPLAY_IR_FUNCTION_SET_4BITS    0b00000000
21 #define DISPLAY_IR_FUNCTION_SET_2LINES   0b00001000
22 #define DISPLAY_IR_FUNCTION_SET_1LINE    0b00000000
23 #define DISPLAY_IR_FUNCTION_SET_5x10DOTS 0b00000100
24 #define DISPLAY_IR_FUNCTION_SET_5x8DOTS  0b00000000
25
26 #define DISPLAY_20x4_LINE1_FIRST_CHARACTER_ADDRESS 0
27 #define DISPLAY_20x4_LINE2_FIRST_CHARACTER_ADDRESS 64
28 #define DISPLAY_20x4_LINE3_FIRST_CHARACTER_ADDRESS 20
29 #define DISPLAY_20x4_LINE4_FIRST_CHARACTER_ADDRESS 84
30
31 #define DISPLAY_RS_INSTRUCTION 0
32 #define DISPLAY_RS_DATA        1
33
34 #define DISPLAY_RW_WRITE 0
35 #define DISPLAY_RW_READ 1

```

*Code 6.6 Defines that are used by the *displayCodeWrite()* function.*

The implementation of *displayCodeWrite()* is shown in Code 6.7. It has two parameters; the first is used to indicate the type of code to be written, and the second is to indicate the value that should be loaded into the data bus (DB7 to DB0). Line 3 assesses if *type* corresponds to *DISPLAY_RS_INSTRUCTION*. In that case, in line 4 the RS pin is assigned a value of 0 by means of the function *displayPinWrite()*. This function receives two parameters, the pin that should be written and the value that should be written into that pin. If *type* is not *DISPLAY_RS_INSTRUCTION*, then the RS pin is assigned a value of 1 (*DISPLAY_RS_DATA*) in line 6.

In line 7, the R/W pin is assigned a value of 0 using the *displayPinWrite()* function and the definition *DISPLAY_RW_WRITE*. Lastly, in line 8 the values of DB7 to DB0 are written into the data bus using *displayDataBusWrite()*. This function also generates the pulse in the E pin, as will be discussed below.

```

1 static void displayCodeWrite( bool type, uint8_t dataBus )
2 {
3     if ( type == DISPLAY_RS_INSTRUCTION )
4         displayPinWrite( DISPLAY_PIN_RS, DISPLAY_RS_INSTRUCTION );
5     else
6         displayPinWrite( DISPLAY_PIN_RS, DISPLAY_RS_DATA );
7     displayPinWrite( DISPLAY_PIN_RW, DISPLAY_RW_WRITE );
8     displayDataBusWrite( dataBus );
9 }
```

Code 6.7 Implementation of the function *displayCodeWrite()*.

NOTE: The *if-else* structure in Code 6.7 is intentionally written without using `{ }` in order to show that if only one statement is used (as in line 4 and line 6), then the braces are not mandatory.

Code 6.8 shows the implementation of *displayPinWrite()*. The parameter *value* is assigned to a `DigitalOut` object indicated by the parameter *pinName*. Code 6.9 shows the `DigitalOut` objects that are declared. In Code 6.10, the `#defines` used in this function are shown (the numbers follow the pin numeration).

It can be seen that in the case of *pinName* equal to `DISPLAY_PIN_RW`, no `DigitalOut` object is assigned because the R/W pin of the display is connected to GND (only write operations can be made on the display).

```

1 static void displayPinWrite( uint8_t pinName, int value )
2 {
3     switch( pinName ) {
4         case DISPLAY_PIN_D0: displayD0 = value; break;
5         case DISPLAY_PIN_D1: displayD1 = value; break;
6         case DISPLAY_PIN_D2: displayD2 = value; break;
7         case DISPLAY_PIN_D3: displayD3 = value; break;
8         case DISPLAY_PIN_D4: displayD4 = value; break;
9         case DISPLAY_PIN_D5: displayD5 = value; break;
10        case DISPLAY_PIN_D6: displayD6 = value; break;
11        case DISPLAY_PIN_D7: displayD7 = value; break;
12        case DISPLAY_PIN_RS: displayRS = value; break;
13        case DISPLAY_PIN_EN: displayEN = value; break;
14        case DISPLAY_PIN_RW: break;
15    default: break;
16 }
17 }
```

Code 6.8 Implementation of the function *displayPinWrite()*.

```

1 DigitalOut displayD0( D0 );
2 DigitalOut displayD1( D1 );
3 DigitalOut displayD2( D2 );
4 DigitalOut displayD3( D3 );
5 DigitalOut displayD4( D4 );
6 DigitalOut displayD5( D5 );
7 DigitalOut displayD6( D6 );
8 DigitalOut displayD7( D7 );
9 DigitalOut displayRS( D8 );
10 DigitalOut displayEN( D9 );
```

Code 6.9 Declaration of public global objects in *display.cpp*.

```
1 #define DISPLAY_PIN_RS 4
2 #define DISPLAY_PIN_RW 5
3 #define DISPLAY_PIN_EN 6
4 #define DISPLAY_PIN_D0 7
5 #define DISPLAY_PIN_D1 8
6 #define DISPLAY_PIN_D2 9
7 #define DISPLAY_PIN_D3 10
8 #define DISPLAY_PIN_D4 11
9 #define DISPLAY_PIN_D5 12
10 #define DISPLAY_PIN_D6 13
11 #define DISPLAY_PIN_D7 14
```

Code 6.10 Defines that are used by the `displayPinWrite()` function.

The implementation of `displayDataBusWrite()` is shown in Code 6.11. In line 3, the E pin is assigned a low state. From line 4 to line 11, the pins of the data bus are written. For this purpose the AND bitwise operator (`&`) is used with a value expressed in binary format. For example, line 4 uses “`& 0b10000000`”, which implies that a high state (1) will be written into `DISPLAY_PIN_D7` if the most significant bit of `dataBus` is 1, while a low state (0) will be written into `DISPLAY_PIN_D7` if the most significant bit of `dataBus` is 0.

The code in lines 12 to 15 is used to generate a pulse in the E pin, as was explained in section 6.2.2.



NOTE: Every time `displayDataBusWrite()` is executed, there is an extra delay of 2 milliseconds (lines 13 and 15) incorporated into `smartHomeSystemUpdate()`. Given that the execution rate of `smartHomeSystemUpdate()` is controlled using a 10-millisecond delay, this 2-millisecond extra delay impacts on how many times per second the function `smartHomeSystemUpdate()` is executed (approximately 16% fewer times).

```
1 static void displayDataBusWrite( uint8_t dataBus )
2 {
3     displayPinWrite( DISPLAY_PIN_EN, OFF );
4     displayPinWrite( DISPLAY_PIN_D7, dataBus & 0b10000000 );
5     displayPinWrite( DISPLAY_PIN_D6, dataBus & 0b01000000 );
6     displayPinWrite( DISPLAY_PIN_D5, dataBus & 0b00100000 );
7     displayPinWrite( DISPLAY_PIN_D4, dataBus & 0b00010000 );
8     displayPinWrite( DISPLAY_PIN_D3, dataBus & 0b00001000 );
9     displayPinWrite( DISPLAY_PIN_D2, dataBus & 0b00000100 );
10    displayPinWrite( DISPLAY_PIN_D1, dataBus & 0b00000010 );
11    displayPinWrite( DISPLAY_PIN_D0, dataBus & 0b00000001 );
12    displayPinWrite( DISPLAY_PIN_EN, ON );
13    delay( 1 );
14    displayPinWrite( DISPLAY_PIN_EN, OFF );
15    delay( 1 );
16 }
```

Code 6.11 Implementation of the function `displayDataBusWrite()`.

In Code 6.12, the implementation of the function `displayCharPositionWrite()`, which was used in Code 6.4, is shown. This function has two parameters: the position in x and y coordinates. In line 3, there is a switch over `charPositionY`. Depending on the value of `charPositionY`, `displayCodeWrite()` is called using different parameters. The first parameter of `displayCodeWrite()` is used to indicate that it is an instruction that should be written to the instruction register. The second parameter is used to indicate that it is a “Set DDRAM address” instruction and the value that should be loaded in the DDRAM address.

For example, if `charPositionY` is 0 (line 4), the DDRAM address value is obtained as “DISPLAY_20x4_LINE1_FIRST_CHARACTER_ADDRESS + `charPositionX`”, where `DISPLAY_20x4_LINE1_FIRST_CHARACTER_ADDRESS` is defined as 0, as was shown in Code 6.6. If `charPositionY` is 1 (line 12), the DDRAM address value is obtained as “DISPLAY_20x4_LINE2_FIRST_CHARACTER_ADDRESS + `charPositionX`”, where `DISPLAY_20x4_LINE2_FIRST_CHARACTER_ADDRESS` is defined as 64, as was shown in Code 6.6.



NOTE: There are other ways to implement `displayCharPositionWrite()` that lead to shorter code. However, the implementation shown in Code 6.12 was chosen because the program code is easy to understand.

The other function of the `display` module used in Code 6.4 is `displayStringWrite()`. The implementation of this function is shown in Code 6.13. It has only one parameter, a pointer to a string (i.e., `char* str`). As was mentioned in previous chapters, a string is an array of characters ending with a null character ('\0'). When, for example, `displayStringWrite("Detected")` is written in the program code, the pointer points to the first element of “Detected”, in this case the character “D”. Then, in the `while` loop (line 3 of Code 6.13), the positions in the array are read one after the other, until the null element is found (remember that a null character is added at the end of an array of char when it is written between quotes, as in “Detected”).

In this way, `displayCodeWrite()` is called on line 4. In this case, the first parameter is `DISPLAY_RS_DATA` to indicate that the value in the data bus should be written to the data register. The second parameter is the character pointed by the pointer `str`. Thus, the corresponding character is written on the display. The pointer is incremented to the next position of the string on line 5. In this way, one after the other the characters of the string are written on the display until the null character ('\0') is found (that is, `*str` is equal to '\0').



NOTE: The `str++` operation in line 4 of Code 6.13 is the first time in the book that a mathematical operation has been made on a pointer. As discussed in Chapter 4, the modification of the value of a pointer should be done carefully, because otherwise improper access to a memory address can be made.

```
1 void displayCharPositionWrite( uint8_t charPositionX, uint8_t charPositionY )
2 {
3     switch( charPositionY ) {
4         case 0:
5             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
6                               DISPLAY_IR_SET_DDRAM_ADDR |
7                               ( DISPLAY_20x4_LINE1_FIRST_CHARACTER_ADDRESS +
8                                 charPositionX ) );
9             delay( 1 );
10            break;
11
12        case 1:
13            displayCodeWrite( DISPLAY_RS_INSTRUCTION,
14                               DISPLAY_IR_SET_DDRAM_ADDR |
15                               ( DISPLAY_20x4_LINE2_FIRST_CHARACTER_ADDRESS +
16                                 charPositionX ) );
17            delay( 1 );
18            break;
19
20        case 2:
21            displayCodeWrite( DISPLAY_RS_INSTRUCTION,
22                               DISPLAY_IR_SET_DDRAM_ADDR |
23                               ( DISPLAY_20x4_LINE3_FIRST_CHARACTER_ADDRESS +
24                                 charPositionX ) );
25            delay( 1 );
26            break;
27
28        case 3:
29            displayCodeWrite( DISPLAY_RS_INSTRUCTION,
30                               DISPLAY_IR_SET_DDRAM_ADDR |
31                               ( DISPLAY_20x4_LINE4_FIRST_CHARACTER_ADDRESS +
32                                 charPositionX ) );
33            delay( 1 );
34            break;
35    }
36 }
```

Code 6.12 Implementation of the function `displayCharPositionWrite()`.

```
1 void displayStringWrite( const char* str )
2 {
3     while (*str) {
4         displayCodeWrite(DISPLAY_RS_DATA, *str++);
5         str++;
6     }
7 }
```

Code 6.13 Implementation of the function `displayStringWrite()`.

Proposed Exercises

1. How can a string “Smart Home” be placed in the center of the fourth line of the display?
2. How can the symbol “°” be placed in the position ($x = 14, y = 0$) to indicate the degree sign?

Answers to the Exercises

1. The fourth line of the display corresponds to $y = 3$, and the string “Smart Home” has 10 characters; so, in order to be centered the string must be placed at $x = 5$. The following statements could be used:

```
displayCharPositionWrite ( 5,3 );
displayStringWrite( "Smart Home" );
```

2. The code for the “ \circ ” symbol in Table 6.1 is 223, so an array `char buffer[3]`; can be declared, and in line 15 of Code 6.4 the following statements can be used:

```
sprintf (buffer, "%cC", 223);
displayStringWrite( buffer );
```

Example 6.2: Use of a 4-Bit Mode to Send Commands and Data to the Display

Objective

Introduce the use of a character LCD display by means of a GPIO connection with fewer wires.

Summary of the Expected Behavior

The expected behavior is the same as in the previous example, the only difference being that a reduced number of connections is used between the display and the NUCLEO board, as shown in Figure 6.13.

Test the Proposed Solution on the Board

Import the project “Example 6.2” using the URL available in [3], build the project, and drag the `.bin` file onto the NUCLEO board. The behavior should be the same as that discussed in Example 6.1.

Discussion of the Proposed Solution

In Code 6.5 of Example 6.1, the configuration “DISPLAY_IR_FUNCTION_SET_8BITS” was used. By reading the datasheet of the character LCD display driver [2], the reader may notice that there is also a 4-bit mode available, which uses only the D4 to D7 pins of the display instead of using D0 to D7. In this example, the `display` module is modified in order to allow the use of both the 8-bit mode and the 4-bit mode connection.



NOTE: Only the functions that are modified are shown. All the other code remains the same.

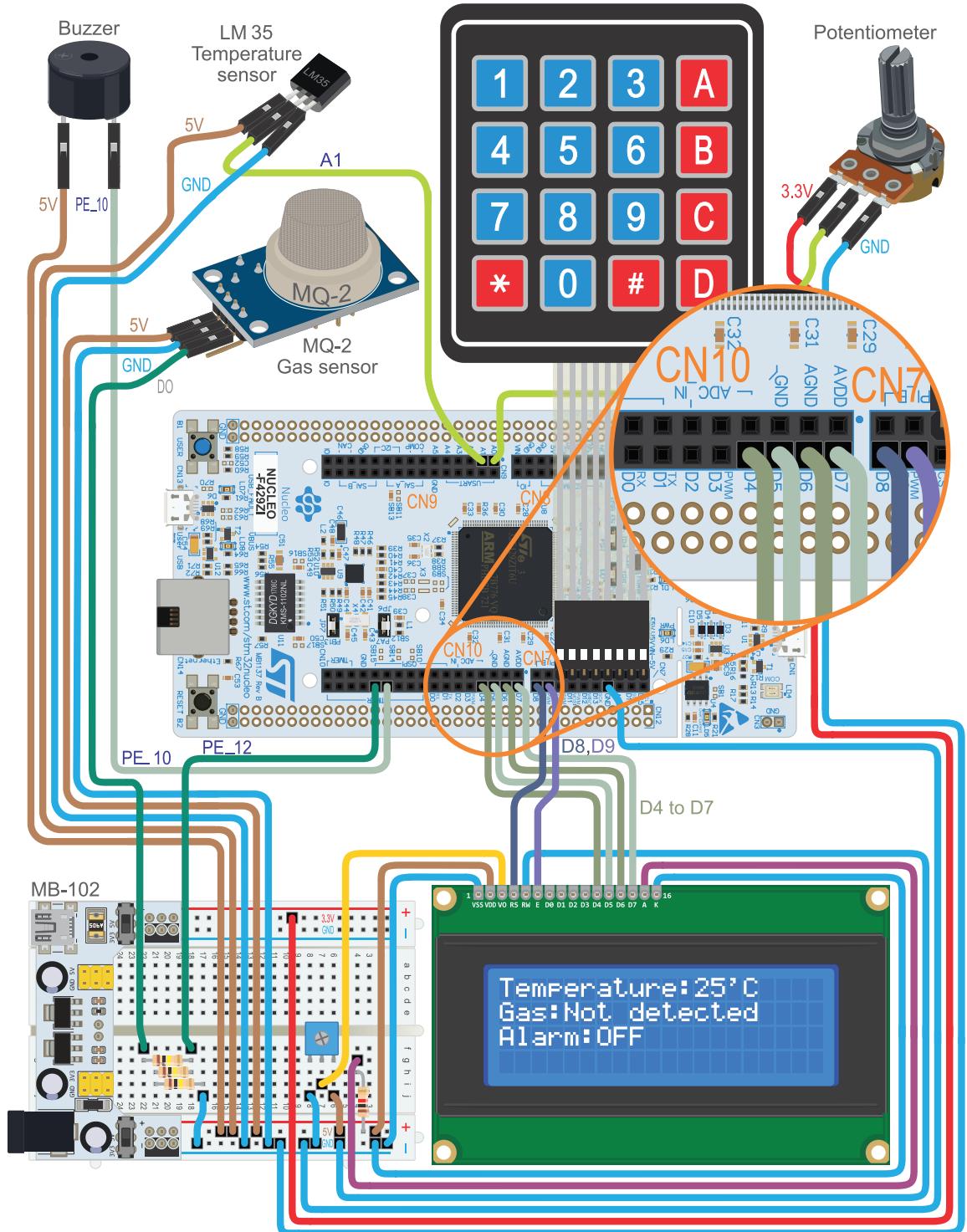


Figure 6.13 The smart home system connected to the character LCD display using 4-bit mode interface.

The transfer timing sequence of writing instructions when a 4-bit interface is configured is shown in Figure 6.14. It can be seen that first the four bits are transferred and then the last four bits are transferred. The only case where the transfer timing sequence shown in Figure 6.13 is not followed is at the beginning of the 4-bit interface initialization procedure, shown in Figure 6.15, that is described in [2]. It can be seen that the initialization procedure is very similar to the 8-bit interface initialization procedure that was introduced in Figure 6.11. The main difference is that now transfers are done using only the D4 to D7 pins of the LCD display and that the “Function Set” instruction is executed five times.

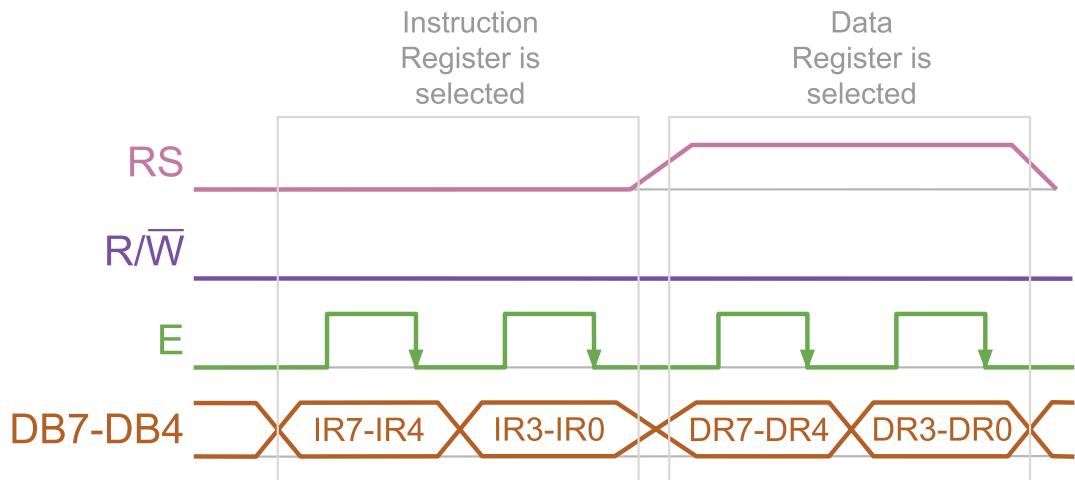


Figure 6.14 Transfer timing sequence of writing instructions when a 4-bit interface is configured.

Implementation of the Proposed Solution

In Code 6.14, the new implementation of *userInterfaceDisplayInit()* is shown. It can be seen on line 4 that *displayInit()* now has a parameter, which in this case is DISPLAY_CONNECTION_GPIO_4BITS.

```

1 static void userInterfaceDisplayInit()
2 {
3     displayInit( DISPLAY_CONNECTION_GPIO_4BITS );
4
5     displayCharPositionWrite( 0,0 );
6     displayStringWrite( "Temperature:" );
7
8     displayCharPositionWrite( 0,1 );
9     displayStringWrite( "Gas:" );
10
11    displayCharPositionWrite( 0,2 );
12    displayStringWrite( "Alarm:" );
13 }
14

```

Code 6.14 Implementation of the function *userInterfaceDisplayInit()*.

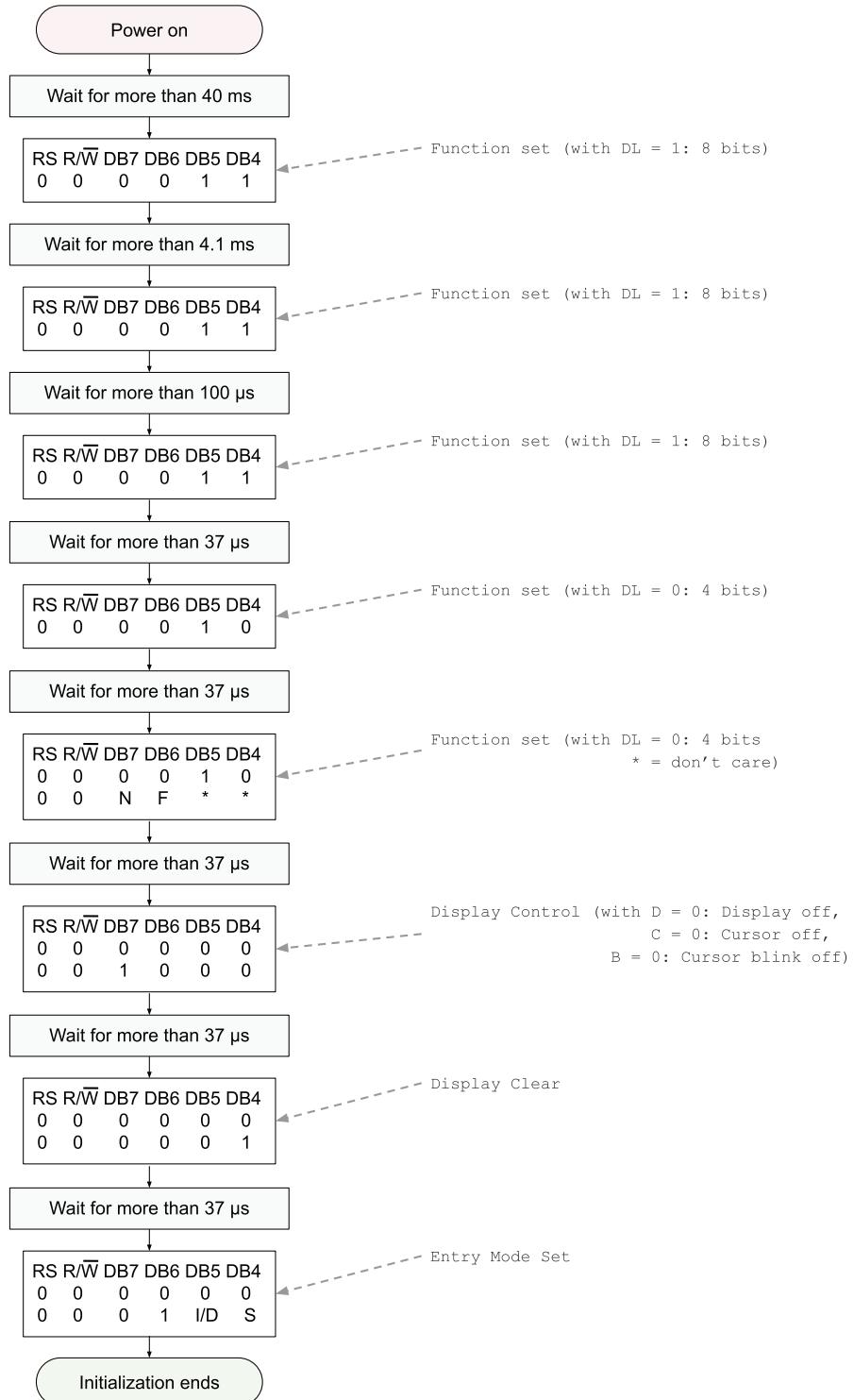


Figure 6.15 Initialization procedure of the graphic display when a 4-bit interface is used.

In Code 6.15 and Code 6.16, the new implementation of *displayInit()* is shown. On line 3 of Code 6.15, a static variable of the user-defined type *displayConnection_t* named *displayConnection* is declared. This type is defined in *display.h* and can have only two values: *DISPLAY_CONNECTION_GPIO_4BITS* and *DISPLAY_CONNECTION_GPIO_8BITS*.

On line 5, a new private global Boolean variable named *initial8BitCommunicationIsCompleted*, that is declared in *display.cpp*, is assigned false. In this way, it is indicated that the transfer timing sequence must be made according to Figure 6.11 (only one E pin pulse per data transfer and only one data bus set of values).

From lines 7 to 22, the program code remains the same as in the previous implementation of *displayInit()*. This is because the first three instructions on Figure 6.11 and Figure 6.15 are the same, the only difference being that in one case the D7 to D0 pins of the LCD display are connected to the NUCLEO board, while in the other case only the D7 to D4 pins of the LCD display are connected to the NUCLEO board.

On line 24, there is a switch statement over *display.connection*. In the case of *DISPLAY_CONNECTION_GPIO_8BITS*, the code between lines 25 and 32 is executed; it corresponds to the fourth instruction shown on Figure 6.11. In the case of *DISPLAY_CONNECTION_GPIO_4BITS*, the code between lines 34 and 48 is executed; it corresponds to the fourth and fifth instructions shown on Figure 6.15. It is important to note that in line 40, *initial8BitCommunicationIsCompleted* is assigned true. In this way, it is indicated that from now on the transfer should be following the timing sequence shown in Figure 6.14. Consequently, the fifth instruction shown in Figure 6.15 is transferred using two E pin pulses per display instruction.

The second part of *displayInit()*, which is shown in Code 6.16, is the same as in Code 6.5, because the last part of Figure 6.11 is equal to the last part of Figure 6.15. The only difference is how the instructions are sent to the display: in the first case the D7 to D0 pins are used, while in the second case only the D7 to D4 pins are used. The new implementation of *displayDataBusWrite()* deals with this.

Code 6.17 shows the new implementation of *displayDataBusWrite()*. Lines 3 to 7 are the same as in the previous implementation (Code 6.11). In line 8, there is a switch statement over *display.connection*. In the case of *DISPLAY_CONNECTION_GPIO_8BITS*, lines 10 to 13 are executed, which are the same as lines 8 to 11 of Code 6.11. In the case of *DISPLAY_CONNECTION_GPIO_4BITS*, the *if* statement on line 17 is evaluated. If *initial8BitCommunicationIsCompleted* is true (it is set true on line 40 of Code 6.15), a pulse on the E pin is generated (lines 18 to 21) and then the *DigitalOut* objects D7 to D4 are written with the values of DB3 to DB0 (lines 22 to 25). For example, on line 22 the statement is *displayPinWrite(DISPLAY_PIN_D7, dataBus & 0b00001000)*, which implies that the fourth bit from the left of *dataBus* (that is, DB3) is written into *DISPLAY_PIN_D7*.

Finally, in lines 30 to 33, a pulse is generated in the E pin. This pulse performs the transfer of the data, either the data corresponding to lines 4 to 13, or 22 to 25, depending on the value of *display.connection*.

```

1 void displayInit( displayConnection_t connection )
2 {
3     display.connection = connection;
4
5     initial8BitCommunicationIsCompleted = false;
6
7     delay( 50 );
8
9     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
10                     DISPLAY_IR_FUNCTION_SET |
11                     DISPLAY_IR_FUNCTION_SET_8BITS );
12    delay( 5 );
13
14    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
15                     DISPLAY_IR_FUNCTION_SET |
16                     DISPLAY_IR_FUNCTION_SET_8BITS );
17    delay( 1 );
18
19    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
20                     DISPLAY_IR_FUNCTION_SET |
21                     DISPLAY_IR_FUNCTION_SET_8BITS );
22    delay( 1 );
23
24    switch( display.connection ) {
25        case DISPLAY_CONNECTION_GPIO_8BITS:
26            displayCodeWrite( DISPLAY_RS_INSTRUCTION,
27                             DISPLAY_IR_FUNCTION_SET |
28                             DISPLAY_IR_FUNCTION_SET_8BITS |
29                             DISPLAY_IR_FUNCTION_SET_2LINES |
30                             DISPLAY_IR_FUNCTION_SET_5x8DOTS );
31            delay( 1 );
32            break;
33
34        case DISPLAY_CONNECTION_GPIO_4BITS:
35            displayCodeWrite( DISPLAY_RS_INSTRUCTION,
36                             DISPLAY_IR_FUNCTION_SET |
37                             DISPLAY_IR_FUNCTION_SET_4BITS );
38            delay( 1 );
39
40            initial8BitCommunicationIsCompleted = true;
41
42            displayCodeWrite( DISPLAY_RS_INSTRUCTION,
43                             DISPLAY_IR_FUNCTION_SET |
44                             DISPLAY_IR_FUNCTION_SET_4BITS |
45                             DISPLAY_IR_FUNCTION_SET_2LINES |
46                             DISPLAY_IR_FUNCTION_SET_5x8DOTS );
47            delay( 1 );
48            break;
49    }
}

```

Code 6.15 New implementation of the function `displayInit()` (Part 1/2).

```

1   displayCodeWrite( DISPLAY_RS_INSTRUCTION,
2                     DISPLAY_IR_DISPLAY_CONTROL |
3                     DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_OFF |
4                     DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
5                     DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
6   delay( 1 );
7
8   displayCodeWrite( DISPLAY_RS_INSTRUCTION,
9                     DISPLAY_IR_CLEAR_DISPLAY );
10  delay( 1 );
11
12  displayCodeWrite( DISPLAY_RS_INSTRUCTION,
13                     DISPLAY_IR_ENTRY_MODE_SET |
14                     DISPLAY_IR_ENTRY_MODE_SET_INCREMENT |
15                     DISPLAY_IR_ENTRY_MODE_SET_NO_SHIFT );
16  delay( 1 );
17
18  displayCodeWrite( DISPLAY_RS_INSTRUCTION,
19                     DISPLAY_IR_DISPLAY_CONTROL |
20                     DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_ON |
21                     DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
22                     DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
23  delay( 1 );
24 }
```

Code 6.16 New implementation of the function `displayInit()` (Part 2/2).

```

1 static void displayDataBusWrite( uint8_t dataBus )
2 {
3     displayPinWrite( DISPLAY_PIN_EN, OFF );
4     displayPinWrite( DISPLAY_PIN_D7, dataBus & 0b10000000 );
5     displayPinWrite( DISPLAY_PIN_D6, dataBus & 0b01000000 );
6     displayPinWrite( DISPLAY_PIN_D5, dataBus & 0b00100000 );
7     displayPinWrite( DISPLAY_PIN_D4, dataBus & 0b00010000 );
8     switch( display.connection ) {
9         case DISPLAY_CONNECTION_GPIO_8BITS:
10            displayPinWrite( DISPLAY_PIN_D3, dataBus & 0b00001000 );
11            displayPinWrite( DISPLAY_PIN_D2, dataBus & 0b00000100 );
12            displayPinWrite( DISPLAY_PIN_D1, dataBus & 0b00000010 );
13            displayPinWrite( DISPLAY_PIN_D0, dataBus & 0b00000001 );
14            break;
15
16        case DISPLAY_CONNECTION_GPIO_4BITS:
17            if ( initial8BitCommunicationIsCompleted == true ) {
18                displayPinWrite( DISPLAY_PIN_EN, ON );
19                delay( 1 );
20                displayPinWrite( DISPLAY_PIN_EN, OFF );
21                delay( 1 );
22                displayPinWrite( DISPLAY_PIN_D7, dataBus & 0b00001000 );
23                displayPinWrite( DISPLAY_PIN_D6, dataBus & 0b00000100 );
24                displayPinWrite( DISPLAY_PIN_D5, dataBus & 0b00000010 );
25                displayPinWrite( DISPLAY_PIN_D4, dataBus & 0b00000001 );
26            }
27            break;
28
29    }
30    displayPinWrite( DISPLAY_PIN_EN, ON );
31    delay( 1 );
32    displayPinWrite( DISPLAY_PIN_EN, OFF );
33    delay( 1 );
34 }
```

Code 6.17 New implementation of the function `displayDataBusWrite()`.



NOTE: There are other ways to implement `displayDataBusWrite()` that lead to shorter code. However, the implementation shown in Code 6.17 was chosen because the program code is easy to understand.

The new implementation of `displayPinWrite()` is shown in Code 6.18. In line 3, there is a switch statement over `display.connection`. In the case of `DISPLAY_CONNECTION_GPIO_8BITS`, the code from lines 5 to 18 is executed, which corresponds to the same code as in Code 6.8. In the case of `DISPLAY_CONNECTION_GPIO_4BITS`, the code between lines 21 and 30 is executed, which is very similar to the code from lines 6 to 18, the difference being that the `DigitalOut` objects `displayD0` to `displayD3` are never assigned a value.

```

1 static void displayPinWrite( uint8_t pinName, int value )
2 {
3     switch( display.connection ) {
4         case DISPLAY_CONNECTION_GPIO_8BITS:
5             switch( pinName ) {
6                 case DISPLAY_PIN_D0: displayD0 = value; break;
7                 case DISPLAY_PIN_D1: displayD1 = value; break;
8                 case DISPLAY_PIN_D2: displayD2 = value; break;
9                 case DISPLAY_PIN_D3: displayD3 = value; break;
10                case DISPLAY_PIN_D4: displayD4 = value; break;
11                case DISPLAY_PIN_D5: displayD5 = value; break;
12                case DISPLAY_PIN_D6: displayD6 = value; break;
13                case DISPLAY_PIN_D7: displayD7 = value; break;
14                case DISPLAY_PIN_RS: displayRS = value; break;
15                case DISPLAY_PIN_EN: displayEN = value; break;
16                case DISPLAY_PIN_RW: break;
17                default: break;
18            }
19            break;
20        case DISPLAY_CONNECTION_GPIO_4BITS:
21            switch( pinName ) {
22                case DISPLAY_PIN_D4: displayD4 = value; break;
23                case DISPLAY_PIN_D5: displayD5 = value; break;
24                case DISPLAY_PIN_D6: displayD6 = value; break;
25                case DISPLAY_PIN_D7: displayD7 = value; break;
26                case DISPLAY_PIN_RS: displayRS = value; break;
27                case DISPLAY_PIN_EN: displayEN = value; break;
28                case DISPLAY_PIN_RW: break;
29                default: break;
30            }
31            break;
32    }
33 }
```

Code 6.18 New implementation of the function `displayPinWrite()`.

Proposed Exercise

1. How can Code 6.14 be modified in order to use an 8-bit mode connection?

Answer to the Exercise

1. Line 3 should be modified as follows:

```
displayInit( DISPLAY_CONNECTION_GPIO_8BITS );
```



WARNING: If an 8-bit mode connection is set, then the eight data pins of the display (D0 to D7) should be connected, as shown in Figure 6.2.



NOTE: An 8-bit mode connection implies less time to transfer data to the display and code that is easier to read when compared with a 4-bit mode connection. However, given that data transfer time between the microcontroller and the display is not usually an issue, 4-bit mode may be more convenient due to the reduced number of connections.

6.2.3 Connect a Character LCD Display to the Smart Home System using the I2C Bus

In the previous sections, many GPIOs and cables were used to connect the character-based LCD display to the NUCLEO board. It might be more convenient to employ a setup that uses fewer cables to connect the character-based LCD display with the NUCLEO board. The proposed solution is shown in Figure 6.16. It can be seen that a module based on the PCF8574 8-bit I/O expander for I2C bus is used, which is described in [6]. This module provides 8 GPIO pins, which are controlled by means of a two-wire I2C bus connection, as summarized in Figure 6.17. The aim of this setup is to reduce the number of cables that are necessary to connect to the NUCLEO board to control the character-based LCD display.

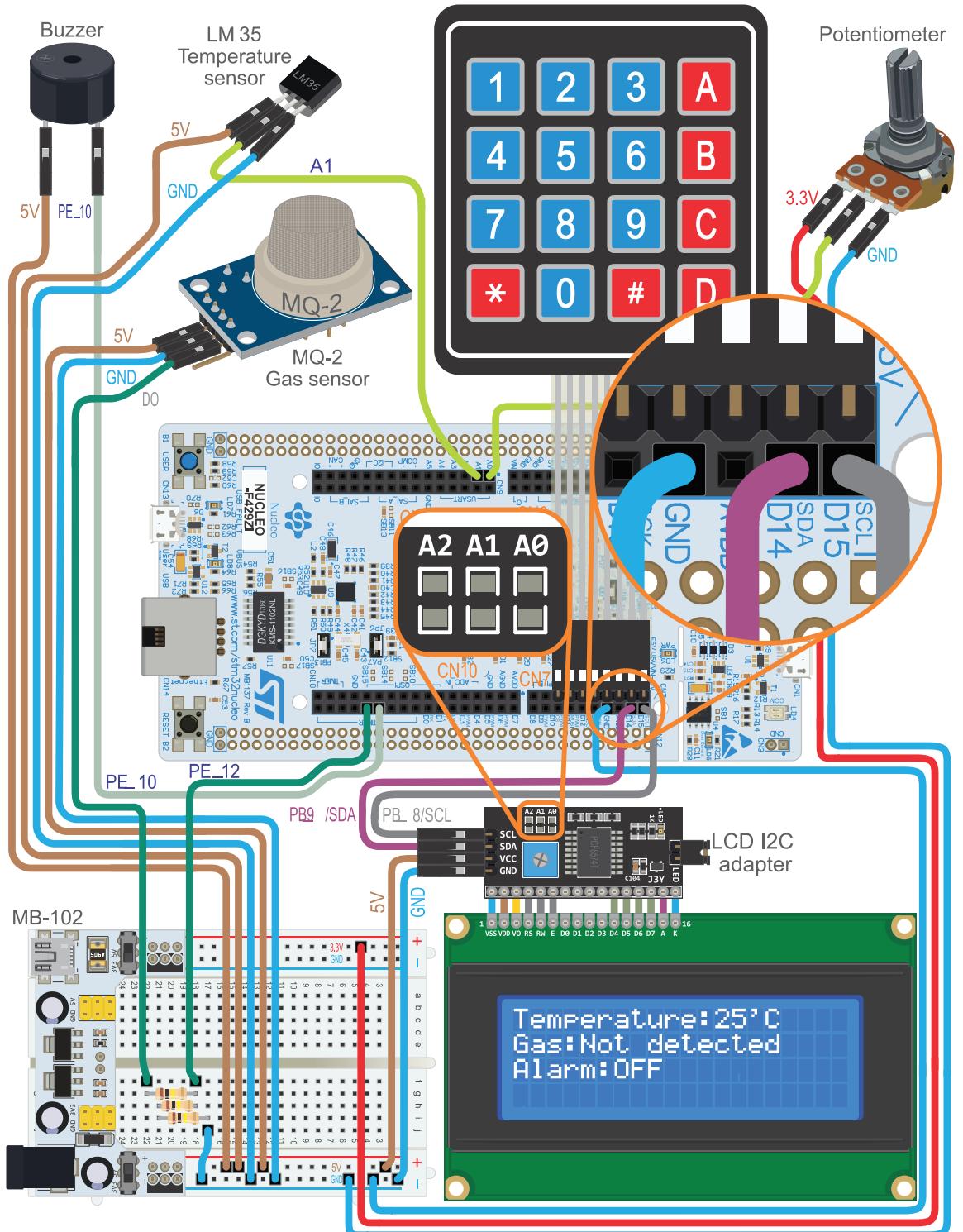


Figure 6.16 The smart home system is now connected to the character LCD display using the I2C bus.

The I²C bus is used to implement a serial communication between the NUCLEO board and the module based on the PCF8574. The PCF8574 LCD module receives, by means of the I²C bus, the commands and data that the NUCLEO board delivers to the character-based LCD display and places those bits in its own GPIOs. These are connected to the character LCD display, as shown in Figure 6.17.



NOTE: The jumper JP1 is used to connect or disconnect the anode of the LCD from the 5 V supply provided in the VCC pin of the PCF8574. The functionality of the A0, A1, and A2 pads is discussed below.

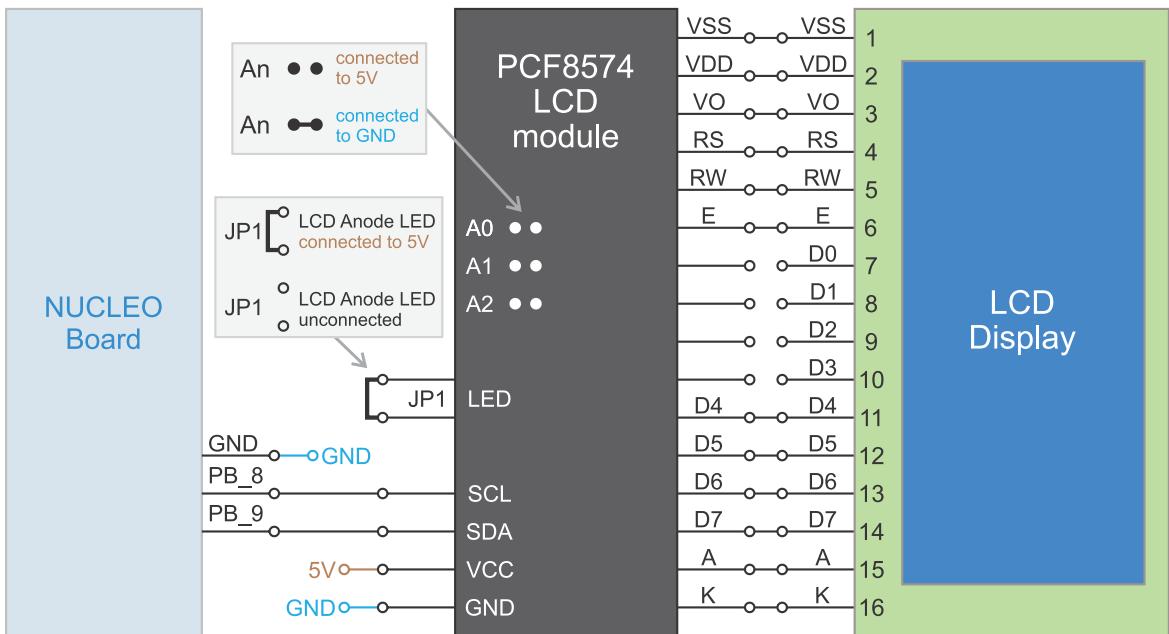


Figure 6.17 Diagram of the connections between the character LCD display and the NUCLEO board using the I²C bus.

A simplified block diagram of the PCF8574, together with its connections to the display, is shown in Figure 6.18. The state of D7, D6, D5, D4, A, E, R/W, and RS is controlled using the I²C bus. For example, if the binary value 0b10001001 is written into the PCF8574, then D7, A, and RS will be in high state, while D6, D5, D4, E, and R/W will be in low state. The pins A0, A1, and A2 are used to set the address of the PCF8574 module, as shown in Table 6.5. If those pins are left unconnected, as in Figure 6.16, they are in high state because of pull-up resistors located in the module, and then the I²C bus 8-bit write address 78 is configured.

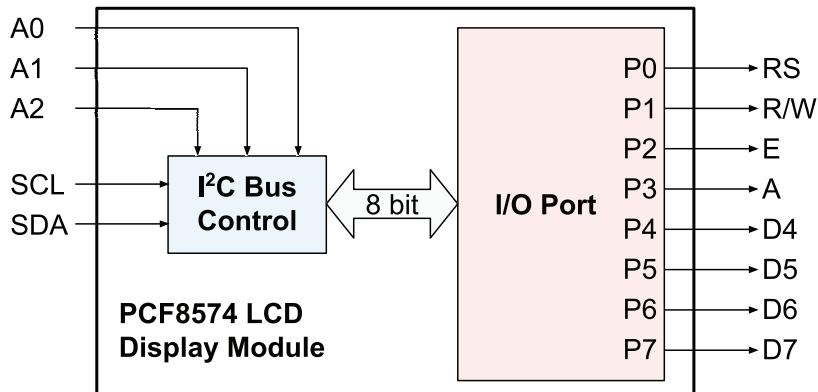


Figure 6.18 Simplified block diagram of the PCF8574 together with its connections to the LCD display.



NOTE: The concept of “I²C bus 8-bit addresses” is used because addresses are defined in this way in [7]. However, the way in which addresses are defined in the I²C bus is discussed in more detail in section 6.2.4, and it is explained that it might be more appropriate to talk about 7-bit addresses, as they are defined in the corresponding standard, as can be seen in [8].

Table 6.5 Address reference of the PCF8574 module. The addresses used in the proposed setup are highlighted.

Inputs			I ² C Bus 8-bit Write Address	I ² C Bus 8-bit Read Address
A2	A1	A0		
Low	Low	Low	64	65
Low	Low	High	66	67
Low	High	Low	68	69
Low	High	High	70	71
High	Low	Low	72	73
High	Low	High	74	75
High	High	Low	76	77
High	High	High	78	79

The reader might notice in Figure 6.16 that the cables and connections are simpler than in Figure 6.2 and Figure 6.13 and that the resistor and the potentiometer have been removed from the breadboard. The PCF8574 LCD module provides a potentiometer that can be used to adjust the contrast of the character LCD display.



WARNING: Other modules based on the PCF8574 are available on the market, as shown in Figure 6.19. Those modules are not convenient for this application because they do not include the potentiometer and resistor that are necessary to connect the character LCD display and, therefore, require more elements.

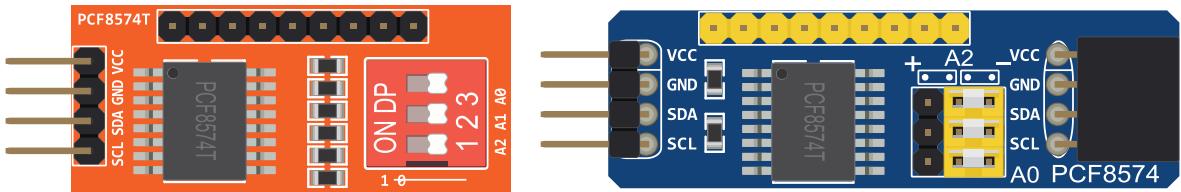


Figure 6.19 Examples of other PCF8574 modules that do not include the potentiometer and the resistors.



TIP: If the reader has two 20×4 character LCD displays, the character LCD display connected as shown in Figure 6.13 can be kept, and a second character LCD display may be connected as shown in Figure 6.16. Only one display will be active, depending on the program code, but it will be easier to compare how they work.

To test if the character LCD display is working, the *.bin* file of the program “Subsection 6.2.3” should be downloaded from the URL available in [3] and dragged onto the NUCLEO board. After power on, the most pertinent information about the smart home system should be shown on the character LCD display.

6.2.4 Fundamentals of the Inter-Integrated Circuit (I2C) Communication Protocol

The I2C bus that is used in the proposed setup is described in [8]. This bus has some similarities and differences to the UART communication that was introduced in Chapter 2. Firstly, on each device there is only one pin that is used to exchange data with other devices, called SDA (*Serial Data Line*), while in the UART serial port each device has a TxD pin to transmit data and an RxD pin to receive data. Secondly, there is a pin called SCL (*Serial Clock Line*), which is used to establish a common clock signal used to control the timing of the data interchange. In this way, the I2C bus establishes a *synchronous communication* (because the clock signal is delivered), and it is not necessary to agree the transmission rate in advance, as in the UART interface, which does not have a clock signal and, therefore, is based on *asynchronous communication*.

A third point, which is very important to highlight, is that UART connection only allows a point-to-point connection between two devices, while an I2C bus allows up to 127 devices to be connected together using only the two connections, SDA and SCL. This idea is illustrated in Figure 6.20, where two pull-up resistors are included because the I2C bus standard establishes that the SDA and SCL outputs are open drain.

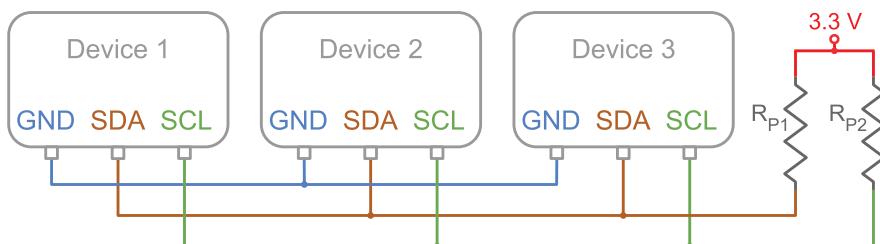


Figure 6.20 Example of a typical I2C bus connection between many devices.

In order to implement the connection of multiple devices using only two wires, a method to unequivocally identify each device must be implemented. In this way, the manager can send a message to any device, and it can be guaranteed that the destination device will recognize that the message is for it alone.

The address of each device is configured by means of a fixed base address plus an optional offset that is modifiable by means of wired connections. In this way, the PCF8574 module can be configured with any 8-bit address in the range of 64/65 to 78/79. The three connections shown in Figure 6.16 on the PCF8574 module (A0, A1, and A2) are used for this purpose, as was shown in Table 6.5.

When a device is not transmitting, it does not establish either a low or high state in its SDA and SCL pins. If no device is transmitting, the SDA and SCL lines are in high state, because of the pull-up resistors (R_p1 and R_p2) that can be seen in Figure 6.20. Therefore, a manager that wants to transmit data establishes a start bit condition on the bus by setting SDA to a low state when the signal in SCL is high, as can be seen in Figure 6.21. The same figure also shows how the stop bit condition is established.



Figure 6.21 Example of I2C bus start and stop conditions.

The first message that the transmitting device sends is the address of the device for which the message is intended. This message is depicted in Figure 6.22. The transmitting device establishes, one after the other, the seven bits of the address of the device for which the message is intended (A6 to A0). This is followed by an R/W bit that indicates if the message is a read or a write operation (it is high if it is a read operation and low if it is a write operation). The ACK (acknowledge) bit (shown in blue) is established by the destination device to confirm that it has received the message.



Figure 6.22 Example of a typical I2C bus address message.

Figure 6.23 shows the typical sequence of an I2C communication. Firstly, the device that starts the communication, called the *manager* device, establishes the start sequence, followed by the address message corresponding to another I2C device, called the *subordinate*. Note that the SCL signal is generated by the manager. In Figure 6.23, the first R/W bit indicates that a read operation is to be made. The subordinate acknowledges in the ninth clock pulse (shown in blue). Then, the manager

indicates which register of the subordinate it wants to read using D7 to D0. Note that this is followed by another start bit without any prior stop bit, after which the manager repeats the address of the subordinate, using the R/W bit to indicate that a read operation is being made. After that, the subordinate writes the 8 bits (indicated in blue) corresponding to the register data following the SCL pulses that are established by the manager. Finally, the manager generates the ACK bit (indicated in brown) and the stop bit.

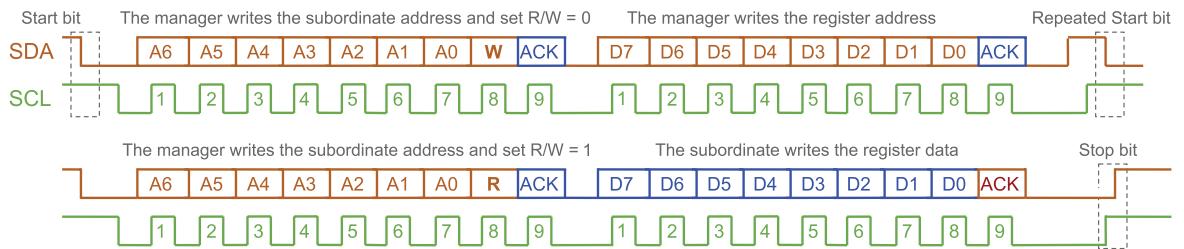


Figure 6.23 Example of a typical I2C bus communication.



NOTE: In a read operation, if the subordinate is not ready to send the data it is allowed to hold the SCL line low. The manager should wait until the SCL line goes high before continuing.

NOTE: The I2C bus allows more than one manager on the bus, but only one manager can use the bus at any one time. For this purpose there is an established arbitration mechanism that is not covered in this book. For further details, the reader is referred to [8].

The examples below will show how to implement I2C communications in order to control the character display using the PCF8574. Given that only write operations will be used, the sequence will be simpler than the one shown in Figure 6.23, and the subordinate will never hold the SCL line low.

Figure 6.24 shows the sequence to write the port pins value of the PCF8574 LCD module. The first four bits of the address, 1000, are fixed and cannot be changed. The next three bits, 111, correspond to A2, A1, and A0, as explained above. The last bit, 0, indicates that it is a write operation. In this way, the 8-bit address value 0b10001110 is obtained, which corresponds to 78 (in decimal notation). Lastly, it can be seen in Figure 6.24 that the port pin values are written (P7-P0). In this way, the display pins D7-D4, A, E, R/W, and RS are established.



Figure 6.24 Example of a writing operation to the PCF8574.



NOTE: 8-bit addresses are obtained considering as a whole the bits A6 to A0 (7 bits) and the R/W bit. It will be seen in the example that considering 8-bit addresses (as was shown in Table 6.5) simplifies the code. However, in most I2C literature, addresses are defined using only bits A6 to A0 (without considering the R/W bit) and, therefore, addresses have 7 bits. In this way, up to 127 different addresses can be used ($2^7 - 1$).

Example 6.3: Control the Character LCD Display by means of the I2C Bus

Objective

Introduce the usage of the I2C bus.

Summary of the Expected Behavior

The expected behavior is exactly the same as in the previous example (Example 6.2), although the connection with the character LCD display has been changed from GPIOs to the I2C bus.

Test the Proposed Solution on the Board

Import the project “Example 6.3” using the URL available in [3], build the project, and drag the `.bin` file onto the NUCLEO board. Follow the same steps as indicated in the section “Test the Proposed Solution on the Board” of Example 6.1. The present temperature, gas detector, and alarm state should be shown on the display.

Discussion of the Proposed Solution

The proposed solution is based on changes introduced in the *display* module. This time, the functions are modified in order to allow the selection of the communication type with the display, the available options being 8-bit GPIO, 4-bit GPIO, or I2C bus by means of the PCF8574 LCD module. All the other characteristics remain the same, as will be shown in the code below. In this way, the concept of a *Hardware Abstraction Layer* is introduced.

Implementation of the Proposed Solution

The new implementation of the function `userInterfaceDisplayInit()` is shown in Code 6.19. It can be seen that the only change is on line 3, where the first parameter is used to indicate how to establish communication with the display. All the other functions and parameters remain the same as in Code 6.3, although the communication with the display has changed. This is possible thanks to the hardware abstraction approach used in the design of the functions of the *display* module.

```

1 static void userInterfaceDisplayInit()
2 {
3     displayInit( DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER );
4
5     displayCharPositionWrite ( 0,0 );
6     displayStringWrite( "Temperature:" );
7
8     displayCharPositionWrite ( 0,1 );
9     displayStringWrite( "Gas:" );
10
11    displayCharPositionWrite ( 0,2 );
12    displayStringWrite( "Alarm:" );
13 }
```

Code 6.19 New implementation of the function *userInterfaceDisplayInit()*.

The implementation of the function *userInterfaceDisplayUpdate()* is shown in Code 6.20. The reader should note that the code remains exactly the same as in Code 6.4, although the interface with the display has changed. Again, this is possible thanks to the hardware abstraction approach followed in the design of the functions of the *display* module. In this way, the maintainability of the code is improved, as well as the possibility of using this same code in other projects, even if other interfaces are used in order to communicate with the display.

```

1 static void userInterfaceDisplayUpdate()
2 {
3     static int accumulatedDisplayTime = 0;
4     char temperatureString[3] = "";
5
6     if( accumulatedDisplayTime >=
7         DISPLAY_REFRESH_TIME_MS ) {
8
9         accumulatedDisplayTime = 0;
10
11        sprintf(temperatureString, "%.0f", temperatureSensorReadCelsius());
12        displayCharPositionWrite ( 12,0 );
13        displayStringWrite( temperatureString );
14        displayCharPositionWrite ( 14,0 );
15        displayStringWrite( "'C" );
16
17        displayCharPositionWrite ( 4,1 );
18
19        if ( gasDetectorStateRead() ) {
20            displayStringWrite( "Detected      " );
21        } else {
22            displayStringWrite( "Not Detected" );
23        }
24
25        displayCharPositionWrite ( 6,2 );
26
27        if ( sirenStateRead() ) {
28            displayStringWrite( "ON      " );
29        } else {
30            displayStringWrite( "OFF" );
31        }
32
33    } else {
34        accumulatedDisplayTime =
35            accumulatedDisplayTime + SYSTEM_TIME_INCREMENT_MS;
36    }
37 }
```

Code 6.20 Implementation of the function *userInterfaceDisplayUpdate()*.

In Code 6.21, the new #defines that are used in *display.cpp* are shown. In line 1, DISPLAY_PIN_A_PCF8574 is defined as 3. In lines 3 and 4, the pins PB_9 and PB_8 used for the I2C connection are defined as I2C1_SDA and I2C1_SCL, respectively. In line 6, PCF8574_I2C_BUS_8BIT_WRITE_ADDRESS is defined as 78.

```
1 #define DISPLAY_PIN_A_PCF8574 3
2
3 #define I2C1_SDA PB_9
4 #define I2C1_SCL PB_8
5
6 #define PCF8574_I2C_BUS_8BIT_WRITE_ADDRESS 78
```

Code 6.21 Declaration of new private #defines in *display.cpp*.

Code 6.22 shows the declaration of the new public global object that is used for the I2C bus communication, as it is declared in *display.cpp*.

```
1 I2C i2cPcf8574( I2C1_SDA, I2C1_SCL );
```

Code 6.22 Declaration of the public global object used to implement the I2C bus communication.

A new private data type, shown in Code 6.23, is declared in *display.cpp* in order to implement the control of the LCD pins using the PCF8574 LCD module.

```
1 typedef struct{
2     int address;
3     char data;
4     bool displayPinRs;
5     bool displayPinRw;
6     bool displayPinEn;
7     bool displayPinA;
8     bool displayPinD4;
9     bool displayPinD5;
10    bool displayPinD6;
11    bool displayPinD7;
12 } pcf8574_t;
```

Code 6.23 New private data type used to implement the control of the LCD pins using the PCF8574 LCD module.

A private variable of type *pcf8574_t* is declared in *display.cpp*, as shown in Code 6.24.

```
1 static pcf8574_t pcf8574;
```

Code 6.24 Declaration of a private variable of type *pcf8574_t*.

The implementation of *displayInit()* is shown in Code 6.25 and Code 6.26. In line 3 of Code 6.25, *display.connection* is assigned and in line 5 it is evaluated to establish if it is equal to DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER. If so, *pcf8574.address* and *pcf8574.data* are

assigned (lines 6 and 7), `i2cPcf8574` frequency is configured as 100,000 Hz (line 8), and the display anode is set to high (line 9).

The remaining lines of Code 6.25 and Code 6.26 are as in the previous version of `displayInit()`, except line 42 of Code 6.25, which is new. The reader might notice that the case of `DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER` is treated in the same way as `DISPLAY_CONNECTION_GPIO_4BITS`. This is due to the fact that they involve the same data transfer, despite data being transferred from the NUCLEO board to the character LCD display using the I2C bus in one setup, and GPIOs in the other setup.

For the same reason, `displayDataBusWrite()` (Code 6.27) is the same as in the previous implementation, except line 17, where the case `DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER` has been added.

In Code 6.28, the new implementation of `displayPinWrite()` is shown. Line 32 starts the case for `DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER`. Firstly (line 33), it is assessed if `value` is true. If so, the corresponding field of the `pcf8574` is set to ON between lines 35 and 43. Otherwise, the corresponding pin is set to OFF between lines 48 and 56.

In line 59, `pcf8574.data` is assigned with `0b00000000`. Then, one after the other, the pins fields of `pcf8574` are evaluated. If they are ON, then the corresponding bit of `pcf8574.data` is turned on. For example, in line 60 it is assessed if `pcf8574.displayPinRs` is true. If so, `pcf8574.data` is set to `0b00000001` by means of the OR bitwise operation `pcf8574.data |= 0b00000001`.

Finally, in line 65 the corresponding value is transferred using `i2cPcf8574.write(pcf8574.address, &pcf8574.data, 1)`. The first parameter is the address, the second parameter is the data to be transferred (which must be preceded by the reference operator (`&`)), and the third parameter (1) is used to indicate that only one byte of data is to be transferred.

```

1 void displayInit( displayConnection_t connection )
2 {
3     display.connection = connection;
4
5     if( display.connection == DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER ) {
6         pcf8574.address = PCF8574_I2C_BUS_8BIT_WRITE_ADDRESS;
7         pcf8574.data = 0b00000000;
8         i2cPcf8574.frequency(100000);
9         displayPinWrite( DISPLAY_PIN_A_PCF8574,  ON );
10    }
11
12    initial8BitCommunicationIsCompleted = false;
13
14    delay( 50 );
15
16    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
17                      DISPLAY_IR_FUNCTION_SET |
18                      DISPLAY_IR_FUNCTION_SET_8BITS );
19    delay( 5 );
20
21    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
22                      DISPLAY_IR_FUNCTION_SET |
23                      DISPLAY_IR_FUNCTION_SET_8BITS );

```

```

24     delay( 1 );
25
26     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
27                         DISPLAY_IR_FUNCTION_SET |
28                         DISPLAY_IR_FUNCTION_SET_8BITS );
29     delay( 1 );
30
31     switch( display.connection ) {
32         case DISPLAY_CONNECTION_GPIO_8BITS:
33             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
34                             DISPLAY_IR_FUNCTION_SET |
35                             DISPLAY_IR_FUNCTION_SET_8BITS |
36                             DISPLAY_IR_FUNCTION_SET_2LINES |
37                             DISPLAY_IR_FUNCTION_SET_5x8DOTS );
38             delay( 1 );
39             break;
40
41         case DISPLAY_CONNECTION_GPIO_4BITS:
42         case DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER:
43             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
44                             DISPLAY_IR_FUNCTION_SET |
45                             DISPLAY_IR_FUNCTION_SET_4BITS );
46             delay( 1 );
47
48             initial8BitCommunicationIsCompleted = true;
49
50             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
51                             DISPLAY_IR_FUNCTION_SET |
52                             DISPLAY_IR_FUNCTION_SET_4BITS |
53                             DISPLAY_IR_FUNCTION_SET_2LINES |
54                             DISPLAY_IR_FUNCTION_SET_5x8DOTS );
55             delay( 1 );
56             break;
57     }

```

Code 6.25 New implementation of the function `displayInit()` (Part 1/2).

```

1     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
2                     DISPLAY_IR_DISPLAY_CONTROL |
3                     DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_OFF |
4                     DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
5                     DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
6     delay( 1 );
7
8     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
9                     DISPLAY_IR_CLEAR_DISPLAY );
10    delay( 1 );
11
12    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
13                     DISPLAY_IR_ENTRY_MODE_SET |
14                     DISPLAY_IR_ENTRY_MODE_SET_INCREMENT |
15                     DISPLAY_IR_ENTRY_MODE_SET_NO_SHIFT );
16    delay( 1 );
17
18    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
19                     DISPLAY_IR_DISPLAY_CONTROL |
20                     DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_ON |
21                     DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
22                     DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
23    delay( 1 );
24 }

```

Code 6.26 New implementation of the function `displayInit()` (Part 2/2).

```

1 static void displayDataBusWrite( uint8_t dataBus )
2 {
3     displayPinWrite( DISPLAY_PIN_EN, OFF );
4     displayPinWrite( DISPLAY_PIN_D7, dataBus & 0b10000000 );
5     displayPinWrite( DISPLAY_PIN_D6, dataBus & 0b01000000 );
6     displayPinWrite( DISPLAY_PIN_D5, dataBus & 0b00100000 );
7     displayPinWrite( DISPLAY_PIN_D4, dataBus & 0b00010000 );
8     switch( display.connection ) {
9         case DISPLAY_CONNECTION_GPIO_8BITS:
10            displayPinWrite( DISPLAY_PIN_D3, dataBus & 0b00001000 );
11            displayPinWrite( DISPLAY_PIN_D2, dataBus & 0b00000100 );
12            displayPinWrite( DISPLAY_PIN_D1, dataBus & 0b00000010 );
13            displayPinWrite( DISPLAY_PIN_D0, dataBus & 0b00000001 );
14            break;
15
16        case DISPLAY_CONNECTION_GPIO_4BITS:
17        case DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER:
18            if ( initial8BitCommunicationIsCompleted == true ) {
19                displayPinWrite( DISPLAY_PIN_EN, ON );
20                delay( 1 );
21                displayPinWrite( DISPLAY_PIN_EN, OFF );
22                delay( 1 );
23                displayPinWrite( DISPLAY_PIN_D7, dataBus & 0b00001000 );
24                displayPinWrite( DISPLAY_PIN_D6, dataBus & 0b00000100 );
25                displayPinWrite( DISPLAY_PIN_D5, dataBus & 0b00000010 );
26                displayPinWrite( DISPLAY_PIN_D4, dataBus & 0b00000001 );
27            }
28            break;
29
30    }
31    displayPinWrite( DISPLAY_PIN_EN, ON );
32    delay( 1 );
33    displayPinWrite( DISPLAY_PIN_EN, OFF );
34    delay( 1 );
35 }

```

Code 6.27 New implementation of the function `displayDataBusWrite()`.

```

1 static void displayPinWrite( uint8_t pinName, int value )
2 {
3     switch( display.connection ) {
4         case DISPLAY_CONNECTION_GPIO_8BITS:
5             switch( pinName ) {
6                 case DISPLAY_PIN_D0: displayD0 = value; break;
7                 case DISPLAY_PIN_D1: displayD1 = value; break;
8                 case DISPLAY_PIN_D2: displayD2 = value; break;
9                 case DISPLAY_PIN_D3: displayD3 = value; break;
10                case DISPLAY_PIN_D4: displayD4 = value; break;
11                case DISPLAY_PIN_D5: displayD5 = value; break;
12                case DISPLAY_PIN_D6: displayD6 = value; break;
13                case DISPLAY_PIN_D7: displayD7 = value; break;
14                case DISPLAY_PIN_RS: displayRS = value; break;
15                case DISPLAY_PIN_EN: displayEN = value; break;
16                case DISPLAY_PIN_RW: break;
17                default: break;
18            }
19            break;
20        case DISPLAY_CONNECTION_GPIO_4BITS:
21            switch( pinName ) {
22                case DISPLAY_PIN_D4: displayD4 = value; break;
23                case DISPLAY_PIN_D5: displayD5 = value; break;
24                case DISPLAY_PIN_D6: displayD6 = value; break;

```

```

25             case DISPLAY_PIN_D7: displayD7 = value; break;
26             case DISPLAY_PIN_RS: displayRS = value; break;
27             case DISPLAY_PIN_EN: displayEN = value; break;
28             case DISPLAY_PIN_RW: break;
29             default: break;
30         }
31     break;
32     case DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER:
33     if ( value ) {
34         switch( pinName ) {
35             case DISPLAY_PIN_D4: pcf8574.displayPinD4 = ON; break;
36             case DISPLAY_PIN_D5: pcf8574.displayPinD5 = ON; break;
37             case DISPLAY_PIN_D6: pcf8574.displayPinD6 = ON; break;
38             case DISPLAY_PIN_D7: pcf8574.displayPinD7 = ON; break;
39             case DISPLAY_PIN_RS: pcf8574.displayPinRs = ON; break;
40             case DISPLAY_PIN_EN: pcf8574.displayPinEn = ON; break;
41             case DISPLAY_PIN_RW: pcf8574.displayPinRw = ON; break;
42             case DISPLAY_PIN_A_PCF8574: pcf8574.displayPinA = ON; break;
43             default: break;
44         }
45     } else {
46         switch( pinName ) {
47             case DISPLAY_PIN_D4: pcf8574.displayPinD4 = OFF; break;
48             case DISPLAY_PIN_D5: pcf8574.displayPinD5 = OFF; break;
49             case DISPLAY_PIN_D6: pcf8574.displayPinD6 = OFF; break;
50             case DISPLAY_PIN_D7: pcf8574.displayPinD7 = OFF; break;
51             case DISPLAY_PIN_RS: pcf8574.displayPinRs = OFF; break;
52             case DISPLAY_PIN_EN: pcf8574.displayPinEn = OFF; break;
53             case DISPLAY_PIN_RW: pcf8574.displayPinRw = OFF; break;
54             case DISPLAY_PIN_A_PCF8574: pcf8574.displayPinA = OFF; break;
55             default: break;
56         }
57     }
58     pcf8574.data = 0b00000000;
59     if ( pcf8574.displayPinRs ) pcf8574.data |= 0b00000001;
60     if ( pcf8574.displayPinRw ) pcf8574.data |= 0b00000010;
61     if ( pcf8574.displayPinEn ) pcf8574.data |= 0b00000100;
62     if ( pcf8574.displayPinA ) pcf8574.data |= 0b00001000;
63     if ( pcf8574.displayPinD4 ) pcf8574.data |= 0b00010000;
64     if ( pcf8574.displayPinD5 ) pcf8574.data |= 0b00100000;
65     if ( pcf8574.displayPinD6 ) pcf8574.data |= 0b01000000;
66     if ( pcf8574.displayPinD7 ) pcf8574.data |= 0b10000000;
67     i2c_pcf8574.write( pcf8574.address, &pcf8574.data, 1 );
68     break;
69   }
70 }
71 }
```

Code 6.28 New implementation of the function `displayPinWrite()`.

Proposed Exercise

- How can a second display be connected using the same I2C bus?

Answer to the Exercise

- A second PCF8574 module should be connected using the signals SDA and SCL of the same bus, and a different address should be configured for this PCF8574 module, changing the configuration of A2, A1, and A0 (as shown in Figure 6.16). Then, when using this module, that address must be used. Note that using this approach, up to eight character-based LCD displays can be connected to the same I2C bus.

6.2.5 Connect a Graphical LCD Display to the Smart Home System using the SPI Bus

In the previous sections, a character-based LCD display was connected to the NUCLEO board using GPIOs and the I2C bus. In this section, a 128×64 pixel graphical LCD display is connected using the SPI bus, as shown in Figure 6.25. The connections that must be made are summarized in Figure 6.26.

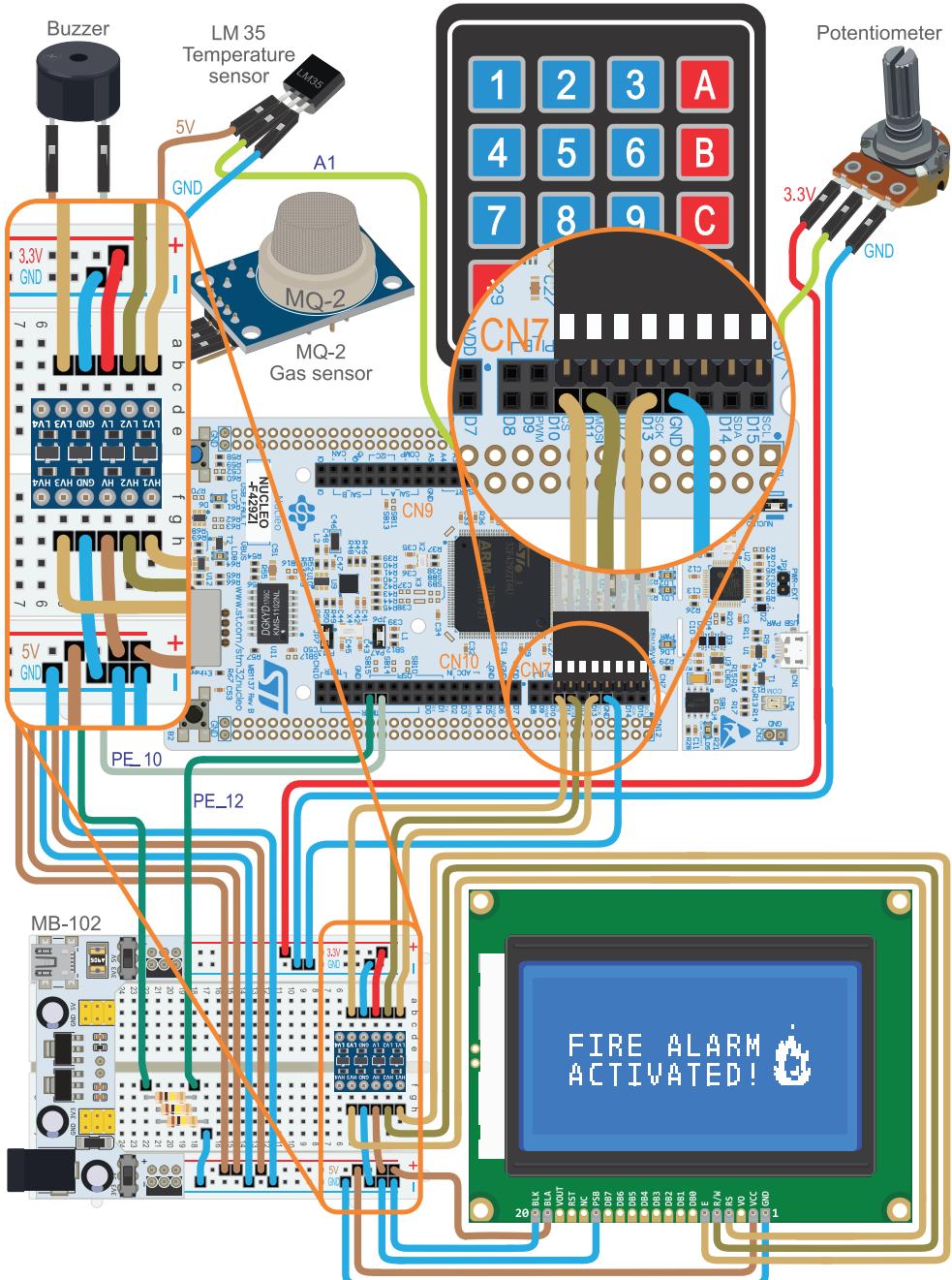


Figure 6.25 The smart home system is now connected to the graphical LCD display using the SPI bus.

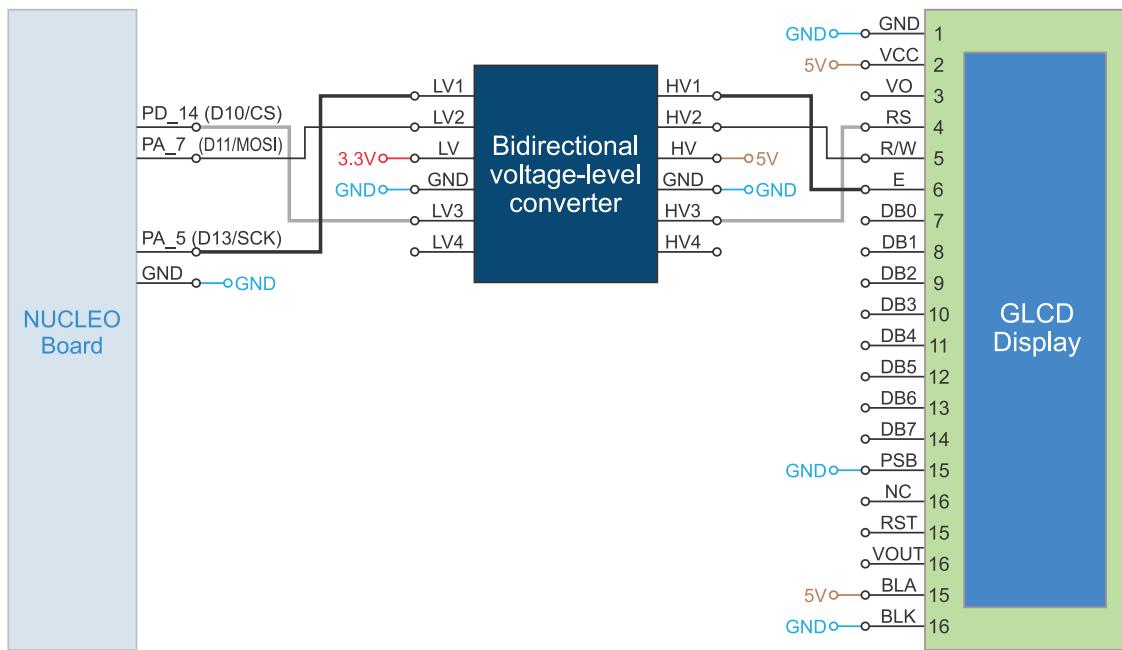


Figure 6.26 Diagram of the connections between the graphical LCD display and the NUCLEO board using the SPI bus.



TIP: The display (or displays) that were previously connected as shown in Figure 6.13 and Figure 6.16 can be kept, and the graphical LCD display connected as shown in Figure 6.25. Only one display will be active at a time, depending on the program code, but it will be easier to compare how they work.

The 4-bit voltage-level converter, which is connected as shown in Figure 6.25, is necessary because the NUCLEO board provides 3.3 V outputs, while the graphical LCD display expects to receive 5 V signals on its E, R/W, and RS pins. The 4-bit voltage-level converter is specially designed to solve this problem.



NOTE: In this case, a voltage divider, as used to connect the MQ-2 Gas Sensor with the NUCLEO board, cannot be used to solve the problem, because the voltage does not need to be attenuated, but augmented.

NOTE: The graphical LCD display can be configured to receive 3.3 V signals, but this requires soldering some SMD (surface-mount device) components, which is beyond the scope of this book.

To test if the graphical LCD display is working, the *.bin* file of the program “Subsection 6.2.5” should be downloaded from the URL available in [3] and dragged onto the NUCLEO board. After power on, the most pertinent information of the smart home system should be shown on the graphical LCD display.

6.2.6 Basics Principles of Graphical LCD Displays

The graphical LCD display module that is used [9] is based on the ST7920 LCD controller [10]. This controller provides all the functionality provided by the character-based LCD display that was used in the previous examples, plus additional functionality, which is possible thanks to the 128×64 pixel arrangement of the graphical LCD display. Every individual pixel is addressable; hence, the display offers graphic capabilities (in this case, monochrome).

The graphical LCD display can be configured in character mode and in graphic mode. If character mode is selected, the LCD display behaves as a 16×4 display that follows the DDRAM map shown in Figure 6.27, where each address stores two characters and each character has 8×16 dots. For example, address 0 stores one 8×16 dot character that is printed in the position indicated as 0_L (left) and another 8×16 dot character that is printed in the position indicated as 0_R (right). This is because the ST7920 LCD controller is optimized for Chinese characters, which have a square shape (e.g., 水 or 光) and, therefore, occupy two display positions for a single character (e.g., 0_L and 0_R). Note that 16×4 by 8×16 results in 128×64 pixels.

0_L	0_R	1_L	1_R	2_L	2_R	3_L	3_R	4_L	4_R	5_L	5_R	6_L	6_R	7_L	7_R
16_L	16_R	17_L	17_R	18_L	18_R	19_L	19_R	20_L	20_R	21_L	21_R	22_L	22_R	23_L	23_R
8_L	8_R	9_L	9_R	10_L	10_R	11_L	11_R	12_L	12_R	13_L	13_R	14_L	14_R	15_L	15_R
24_L	24_R	25_L	25_R	26_L	26_R	27_L	27_R	28_L	28_R	29_L	29_R	30_L	30_R	31_L	31_R

Figure 6.27 Addresses corresponding to each of the positions of a graphical LCD display in character mode.

It is possible to write a single Latin character (e.g., "a" or "b") in the left half of a position (for example, 0_L , 1_L , etc.). For that purpose, the corresponding DDRAM address must be set (for example, 0, 1, etc.) and then the character transferred. However, it is not possible to write a Latin character only on the right half of a position (for example, 0_R , 1_R , etc.). To write a character in the right half of a position, first a character must be written in the left half of the position and then the desired character must be written in the right half. For example, to write "Gas:" at the beginning of the second row, the address 16 must be written into the DDRAM and then, one after the other, the characters "G", "a", "s", ":" are transferred. To write "Not detected" just next to "Gas:", address 18 must be set and the characters "N", "o", "t", " ", "D", "e", "t", "e", "c", "t", "e", "d" transferred.



NOTE: In the ST7920 datasheet, “high” and “low” are used in place of “left” and “right”, respectively.

NOTE: In previous examples, the coordinate (4,1) was used to write “Not detected” in the DDRAM address 68. This address was calculated as 4 added to the first character address of line 2 (64). For the reasons explained above, in order to write “Not detected” in the same display position, the coordinate (4,1) is used in the examples below, and the corresponding address (18) is calculated as the first character address of line 2 (16) increased by the x coordinate (4) divided by 2. In this way, 18 is obtained as “ $16 + 4/2$ ”.

The instructions to initialize the graphical LCD display and to transfer the characters are summarized in Table 6.6. They are almost the same as the instructions that were used in previous sections to control the character LCD display. The differences are highlighted in blue. It can be seen that a bit is not used to set the number of lines (N). There is a bit used to set the instruction set (RE); this bit must be set to 0 during the initialization in order to select the basic instruction set. The bit G is used to activate graphic mode.

Table 6.6 Summary of the graphical LCD display instructions that are used in this chapter.

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift.	1.52 ms
Display control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor (B).	37 µs
Function set	0	0	0	0	1	DL	*	RE	G	*	Sets interface data length (DL), instruction set (RE), and graphic display (G).	37 µs
Set DDRAM address	0	0	1	A6	A5	A4	A3	A2	A1	A0	Sets DDRAM address.	37 µs
Write data to DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Writes data into DDRAM.	37 µs
I/D = 1: Increment, I/D = 0: Decrement S = 1: Accompany display shift, S = 0: Don't accompany display shift, D = 1: Display on, D = 0: Display off C = 1: Cursor on, C = 0: Cursor off B = 1: Cursor blink on, B = 0: Cursor blink off										DL = 1: 8 bits, RE = 1: Extended, G = 1: Graphics on, A6 ... A0 = Address * = don't care A6 ... A0 = Address D7 ... D0 = Data	DL = 0: 4 bits RE = 0: Basic (instruction set) G = 0: Graphics off	

The PSB pin (Parallel/Serial Bus configuration) of the graphical LCD display is used to select its interface. If PSB is connected to 5 V, then 8/4-bit parallel bus mode is selected, and the data transfer is made exactly as in the character LCD display (Figure 6.11 and Figure 6.13). If PSB is connected to GND (as in Figure 6.25), then a serial communication interface based on the SPI (Serial Peripheral Interface) bus is selected.

In the serial interface mode, the RS, R/W, and E pins play a different role than the ones discussed in the previous sections. This is summarized in Table 6.7.

Table 6.7 Connections of the graphical LCD display used in this book when the serial bus option is selected.

Pin label	Pin functionality in serial mode	Description
RS	CS	Chip select (high: chip enable, low: chip disable)
R/W	SID	Serial data input
E	SCLK	Serial clock

The data transfer is made following the timing diagram shown in Figure 6.28. CS must be high during the data transfer. The SCLK signal is used to implement the clock of the transmission (in a similar way as the SCL signal of the I2C bus). The SID signal is used to transfer the data to the LCD display. First, a synchronizing bit string (0b11111) is sent, followed by the state of RW and the state of RS. Then, a 0 state is sent. Next, D7 to D4 are sent, followed by four zeros. Lastly, D3 to D0 are sent, followed by four zeros.

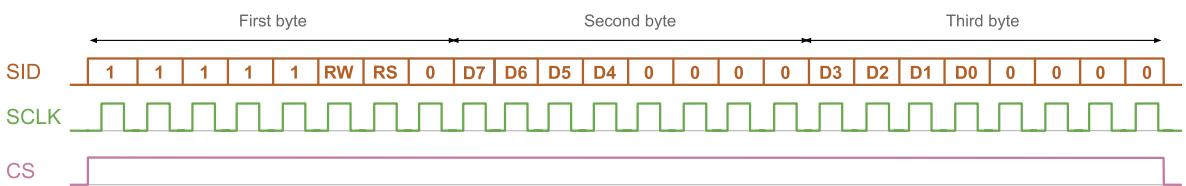


Figure 6.28 Transfer timing sequence of the graphical LCD display when the serial mode is configured.

As has been mentioned, the G bit is used to activate graphic mode. RE and G cannot be altered in the same instruction, so RE is changed first and G later. The examples below show how this is done.

Once the display is in graphic mode, the structure of the data transfer completely changes, and instead of DDRAM, the Graphic Display RAM (GDRAM) is used. The way in which data loaded into the GDRAM is shown in the display is very particular and is closely related to the way in which data is organized in Figure 6.27.

Data organization in Figure 6.27 suggests that it can be thought of as a 32×2 character display, where each character has 8×16 bits, and whose two rows are “cut” at the middle and reorganized as a 16×4 display. This is actually the case, because the ST7920 has 32 common signal driver pins and 64 segment signal driver pins. To drive a 64×128 dot matrix LCD panel, the ST7920 controls two ST7921 chips, as shown in Figure 6.29. Each ST7921 is capable of driving 96 segments. In this way, a

ST7920 is used to control a 32×256 dot matrix LCD. In the graphical LCD display module used in this chapter, the dots of this matrix are arranged in two 32×128 dot layouts, one over the other, as shown in Figure 6.30.

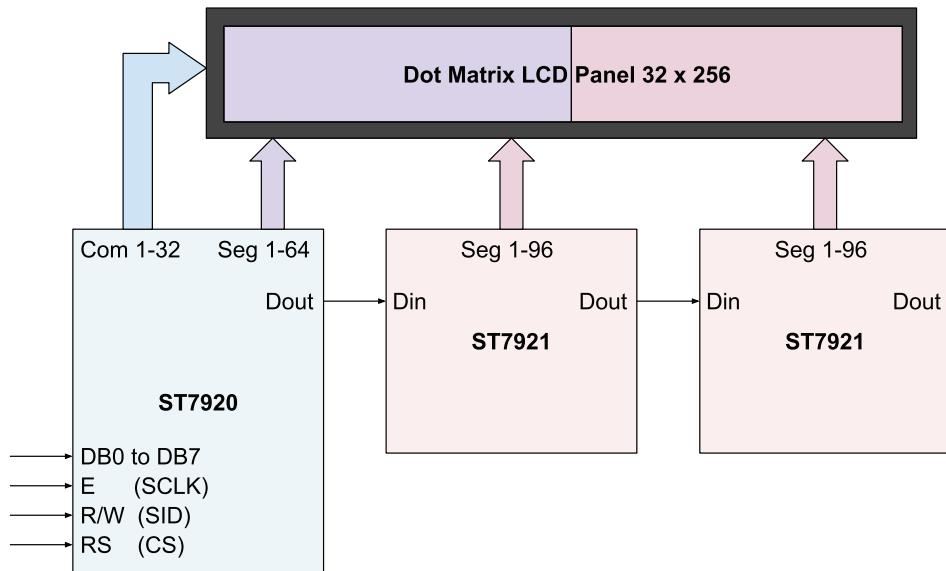


Figure 6.29 Simplified block diagram of an ST7920 and two ST7921 used to drive a 32×256 dot matrix LCD panel.

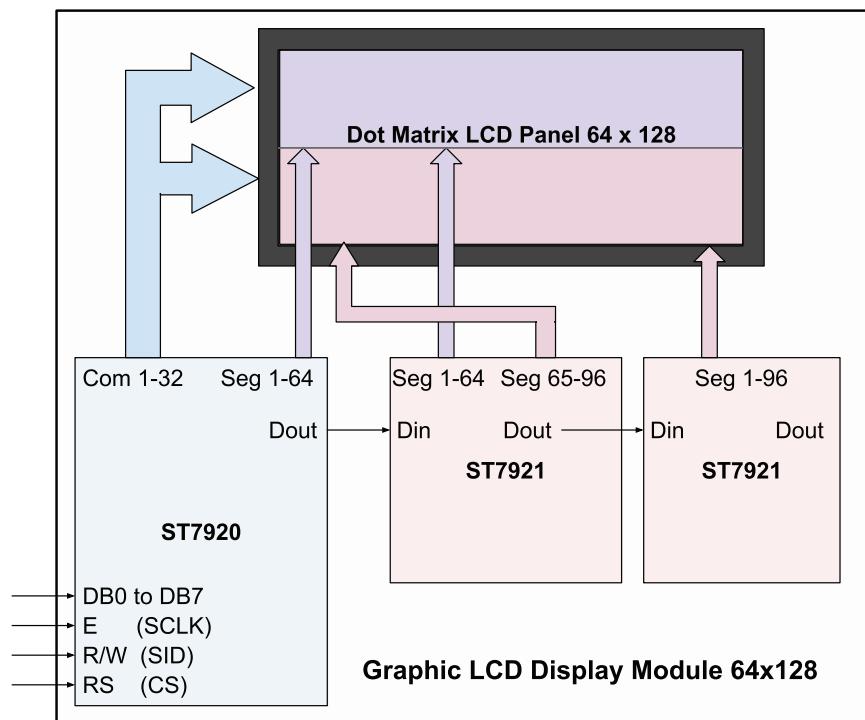


Figure 6.30 Simplified block diagram of an ST7920 and two ST7921 used to drive a 64×128 dot matrix LCD panel.

For this reason, the correspondence between the GDRAM addresses and the display pixels is as shown in Figure 6.31. Each GDRAM address stores 16 bits and is identified by a horizontal and a vertical address. To load the content of a given GDRAM address, the transfer procedure is as follows:

1. Set vertical address (Y) for GDRAM
2. Set horizontal address (X) for GDRAM
3. Write the bits b15 to b8 to GDRAM (first byte)
4. Write the bits b7 to b0 to GDRAM (second byte)

The “Set DDRAM address” instruction code shown in Table 6.6 is used to set the vertical and horizontal address, and the “Write data to DDRAM” instruction code shown in Table 6.6 is used to write the GDRAM bits.

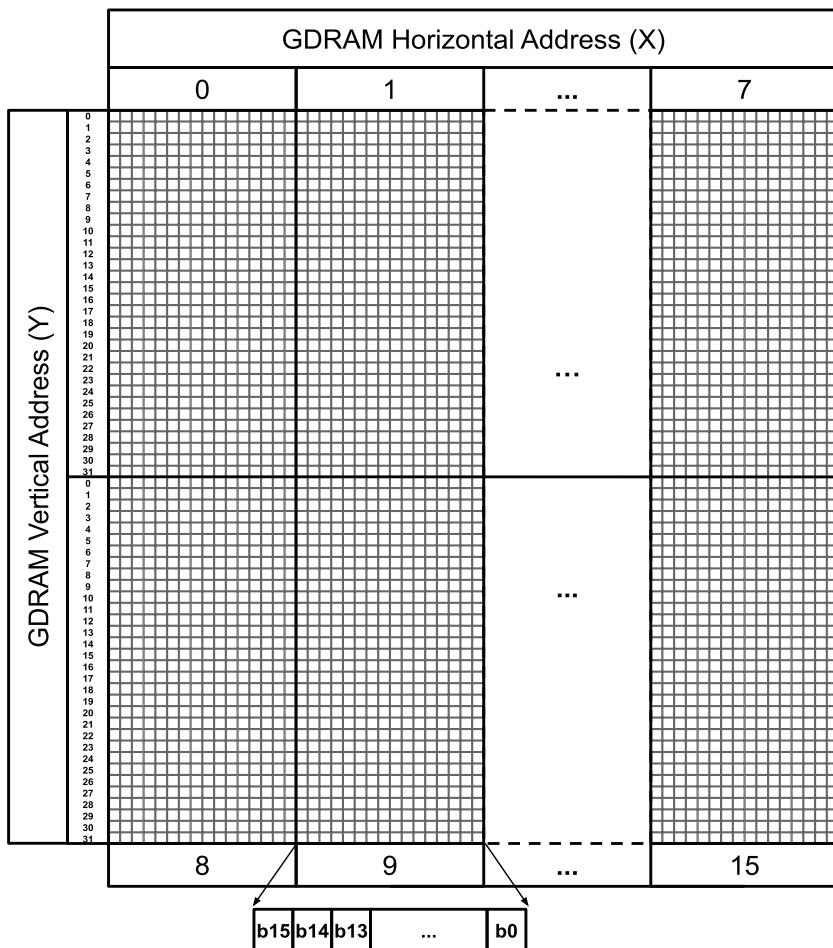


Figure 6.31 Diagram of the correspondence between the GDRAM addresses and the display pixels.



NOTE: The ST7920 includes other functionality that is not discussed here for space reasons and because meaningful programs can be made using the functionality that has been presented in this section, as is shown in the examples below.

NOTE: The HD44780 used in previous sections has 16 common signal driver pins and 40 segment signal driver pins, as shown in [2]. Also in [2], it is shown how the connections are made inside character LCD modules in order to control LCD displays having different layouts using a single HD44780.

6.2.7 Fundamentals of the Serial Peripheral Interface (SPI) Communication Protocol

The transfer timing sequence shown in Figure 6.28 is a special (reduced) implementation of the SPI bus [11]. The SPI bus is implemented by means of four signals, as shown in Table 6.8. It can be seen that the graphical LCD display has completely different pin designations than the signals shown in Table 6.8. The CS, SID, and SCLK signals in Table 6.7 correspond to the SS, MOSI, and SCLK signals, respectively.

Table 6.8 Signals of the SPI bus.

Signal name	Function
SS (Subordinate Select)	The manager sets this to low to select only one subordinate at a time.
SCLK (Clock)	The manager drives this signal, which is common to all the subordinates.
MOSI (Manager Output Subordinate Input)	The manager sends data to one subordinate at a time using this signal.
MISO (Manager Input Subordinate Output)	The manager receives data from one subordinate at a time using this signal.

It can also be seen that the graphical LCD display does not have a signal equivalent to the MISO signal (i.e., the graphical LCD is not able to output data to the NUCLEO board when the serial bus configuration option is used). The SPI signals that are provided by the NUCLEO board are summarized in Table 6.9.

Table 6.9 Summary of the NUCLEO board pins that are used to implement the SPI bus communication.

NUCLEO board	SPI bus signal
PD_14	CS
PA_7	MOSI
PA_5	SCK

In Figure 6.32, a typical SPI connection between a manager and several subordinates can be seen. There is only one SCLK signal, only one MOSI signal, and only one MISO signal, but as many SS signals as subordinates. In this way, the manager controls which subordinate is active on the bus. It should be noted that there can be only one manager in an SPI bus network.

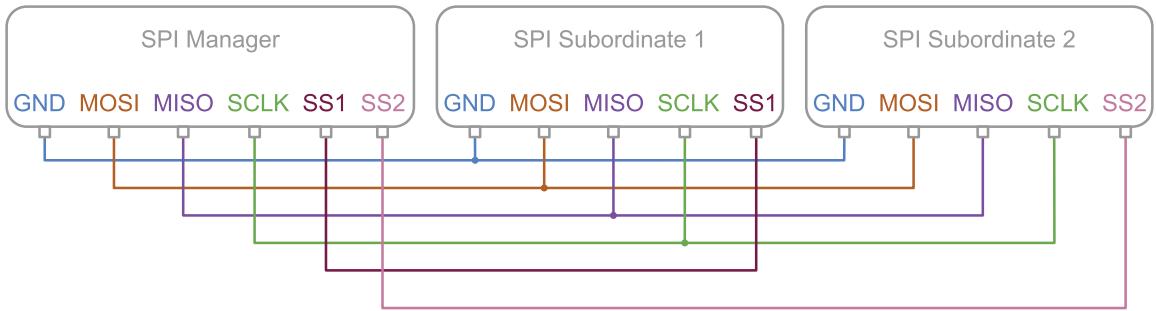


Figure 6.32 Example of a typical SPI bus connection between many devices.



NOTE: If a single subordinate device is used, the SS pin may be fixed to logic low, as shown in Figure 6.25.

Finally, it should be noted that, unlike the I₂C bus, the SPI bus does not have a start bit, a stop bit, or an ACK bit. Moreover, there is no definition for how many bits there are in a message, although 8-bit messages are the most common. There is also no unique option for the *clock polarity* and *clock phase* with respect to the data; thus, there are four possible situations, each of which is known as an SPI mode, as shown in Figure 6.33. The corresponding SPI mode should be configured, as will be shown in the examples below.

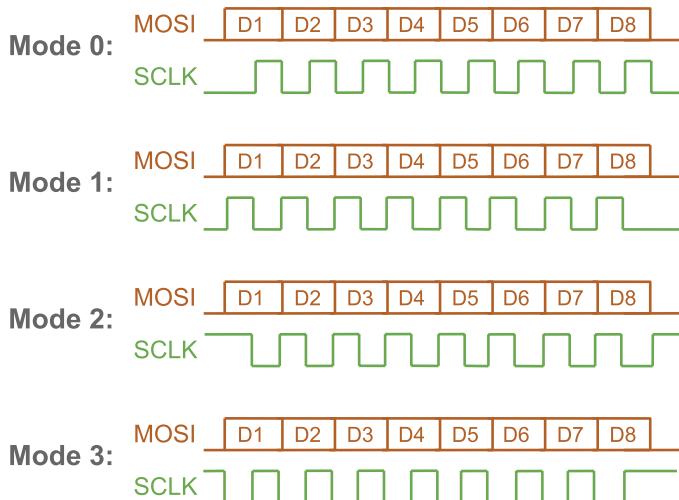


Figure 6.33 Diagram of the four possible SPI modes, depending on the clock polarity and clock phase.



NOTE: In Figure 6.33, only the MOSI signal and the SCLK signal are shown. However, once SPI mode is configured, the MISO signal must respect the same clock polarity and phase as the MOSI signal.

Example 6.4: Control the Graphical LCD Display by means of the SPI Bus

Objective

Introduce the use of the SPI bus.

Summary of the Expected Behavior

The expected behavior is exactly the same as in Examples 6.1, 6.2, and 6.3, although a graphical LCD display is now being used and that connection with the display is implemented using the SPI bus.

Test the Proposed Solution on the Board

Import the project “Example 6.4” using the URL available in [3], build the project, and drag the .bin file onto the NUCLEO board. Follow the same steps as indicated in the section “Test the Proposed Solution on the Board” of Example 6.1. The present temperature, gas detector, and alarm state should be shown on the display.

Discussion of the Proposed Solution

The functions of the *display* module are modified in order to allow communication with a graphical LCD display by means of the SPI bus. All the other characteristics of the functions remain the same, as will be shown. In this way, the concept of a *Hardware Abstraction Layer* is applied one more time.



NOTE: The graphical features of the graphical LCD display are not used in this example. Those features are introduced in Example 6.5.

Implementation of the Proposed Solution

The new implementation of the function *userInterfaceDisplayInit()* is shown in Code 6.29. It can be seen that the function *displayInit()* on line 3 has been changed in order to include the display type and the type of connection. It can also be seen that all the other functions and parameters from lines 10 to 17 remain the same as in Code 6.19, although the communication with the display has changed, as well as the type of display being used.

```
1 static void userInterfaceDisplayInit()
2 {
3     displayInit( DISPLAY_TYPE_GLCD_ST7920, DISPLAY_CONNECTION_SPI );
4
5     displayCharPositionWrite ( 0,0 );
6     displayStringWrite( "Temperature:" );
7
8     displayCharPositionWrite ( 0,1 );
9     displayStringWrite( "Gas:" );
10
11    displayCharPositionWrite ( 0,2 );
12    displayStringWrite( "Alarm:" );
13 }
```

Code 6.29 New implementation of the function *userInterfaceDisplayInit()*.

The public data types to implement the parameters used in line 3 of Code 6.29 are declared in *display.h*, as shown in Code 6.30. The data type *displayType_t* has two possible values, DISPLAY_TYPE_LCD_HD44780 and DISPLAY_TYPE_GLCD_ST7920. The data type *displayConnection_t* (lines 6 to 11) was used in previous examples. In this example, the value DISPLAY_CONNECTION_SPI is included. Using these data types, the struct *display_t* is declared (lines 13 to 16), having two fields, *connection* and *type*.

The new private #defines that are declared in *display.cpp* are shown in Code 6.31. The corresponding values were discussed in section 6.2.6. SPI1_MISO is defined because it is required by the SPI object, but it is left unconnected. ST7920_SPI_SYNCHRONIZING_BIT_STRING is defined as 0b11111000 and will be used to implement the first five bits, shown in Figure 6.28. By means of the definitions shown in lines 13 to 17, the state of the RS and RW bits shown in Figure 6.28 will be implemented.

```

1  typedef enum {
2      DISPLAY_TYPE_LCD_HD44780,
3      DISPLAY_TYPE_GLCD_ST7920,
4  } displayType_t;
5
6  typedef enum {
7      DISPLAY_CONNECTION_GPIO_4BITS,
8      DISPLAY_CONNECTION_GPIO_8BITS,
9      DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER,
10     DISPLAY_CONNECTION_SPI
11 } displayConnection_t;
12
13 typedef struct {
14     displayConnection_t connection;
15     displayType_t type;
16 } display_t;

```

Code 6.30 Public data types defined in *display.h*.

```

1  #define SPI1_MOSI PA_7
2  #define SPI1_MISO PA_6
3  #define SPI1_SCK PA_5
4  #define SPI1_CS PD_14
5
6  #define DISPLAY_ST7920_LINE1_FIRST_CHARACTER_ADDRESS 0
7  #define DISPLAY_ST7920_LINE2_FIRST_CHARACTER_ADDRESS 16
8  #define DISPLAY_ST7920_LINE3_FIRST_CHARACTER_ADDRESS 8
9  #define DISPLAY_ST7920_LINE4_FIRST_CHARACTER_ADDRESS 24
10
11 #define ST7920_SPI_SYNCHRONIZING_BIT_STRING 0b11111000
12
13 #define ST7920_SPI_RS_INSTRUCTION 0b0000000000
14 #define ST7920_SPI_RS_DATA 0b0000000010
15
16 #define ST7920_SPI_RW_WRITE 0b0000000000
17 #define ST7920_SPI_RW_READ 0b0000000100

```

Code 6.31 New private defines that are declared in *display.cpp*.

Two new public objects are declared, as shown in Code 6.32. DigitalOut object `spiSt7920ChipSelect` is used to generate the CS signal, according to Figure 6.28. The SPI object allows the implementation of the SCLK and SID signals, as shown in the same figure.

```
1 DigitalOut spiSt7920ChipSelect(SPI1_CS);  
2 SPI spiSt7920(SPI1_MOSI, SPI1_MISO, SPI1_SCK);
```

Code 6.32 New public objects that are declared in *display.cpp*.



NOTE: The SPI object could also be defined as `SPI spiSt7920(SPI1_MOSI, SPI1_MISO, SPI1_SCK, SPI1_CS)`, and the `DigitalOut` object `spiSt7920ChipSelect(SPI1_CS)` would not be necessary. However, when using that definition of the `SPI` object, only one SPI device can be connected to the SPI bus, while defining Chip Select pin as a `DigitalOut` object means that many devices can be connected to the same SPI bus.

The implementation of the function `userInterfaceDisplayUpdate()` used in this example is exactly the same as in Examples 6.1, 6.2, and 6.3. This fact is quite remarkable, considering that in this example a graphical LCD display is being used and that the communication with the display is implemented by means of an SPI bus connection. In this way, it can be appreciated how powerful a Hardware Abstraction Layer (HAL) can be in terms of code reusability and maintainability. For this reason, the HAL must be carefully designed.

The HAL implemented in this chapter is shown in Figure 6.34. It can be seen that the *display* module allows the use of different types of connections without changing the program code. In this way, a graphical display in character mode connected using the SPI bus can be controlled using the same functions (i.e., `displayCharPositionWrite()` and `displayStringWrite()`) as the character display connected using either 4-bit GPIO, 8-bit GPIO, or I2C bus.

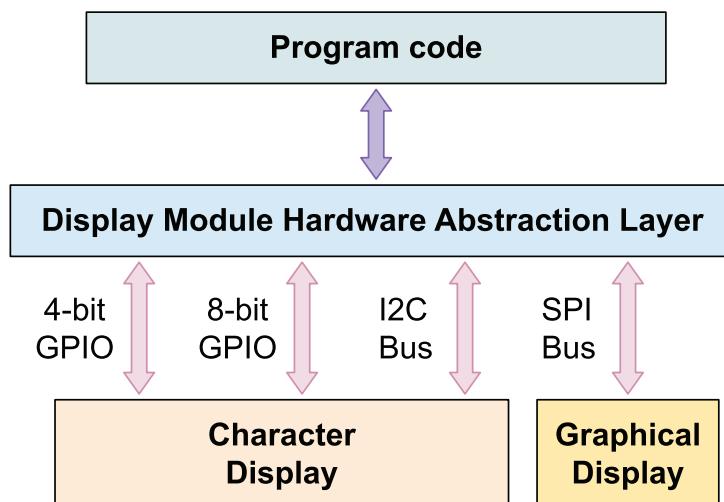


Figure 6.34 Diagram of the Hardware Abstraction Layer that is implemented in the *display* module.

In this context, for the sake of completeness, the program code should be written in such a way that it becomes clear what every function will do according to the selected type of display and connection. For this reason, the lines shown in Code 6.33 are included at the end of *displayPinWrite()* and *displayDataBusWrite()* to make clear that no action will be made by those functions if they are called when *display.connection* is set to **DISPLAY_CONNECTION_SPI**. This is because when the SPI bus connection is used, the only way to write information in the display is using the scheme introduced in Figure 6.28 (i.e., it is not possible to write into a single pin or to write only to the data bus).

```

1     case DISPLAY_CONNECTION_SPI:
2         break;

```

Code 6.33 Lines added to *displayPinWrite()* and *displayDataBusWrite()*.

The new implementation of the function *displayInit()* is shown in Code 6.34 and Code 6.35. In line 3 of Code 6.34, the value of the parameter type is set to *display.type*, and on line 4, the value of the parameter *connection* is assigned to *display.connection*. The new lines in Code 6.34 are lines 13 to 16 and line 39. In line 14, *spiSt7920* is configured with 8 bits per transfer and mode 3, as discussed in Figure 6.33. In line 15, the SPI object is configured to 1,000,000 Hz (1 MHz). Line 39 indicates that the SPI initialization implies the same instructions as the 8-bit parallel mode. Code 6.35 is exactly the same as in previous examples.

```

1 void displayInit( displayType_t type, displayConnection_t connection )
2 {
3     display.type = type;
4     display.connection = connection;
5
6     if( display.connection == DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER ) {
7         pcf8574.address = PCF8574_I2C_BUS_8BIT_WRITE_ADDRESS;
8         pcf8574.data = 0b00000000;
9         i2cPcf8574.frequency(100000);
10        displayPinWrite( DISPLAY_PIN_A_PCF8574, ON );
11    }
12
13    if( display.connection == DISPLAY_CONNECTION_SPI ) {
14        spiSt7920.format( 8, 3 );
15        spiSt7920.frequency( 1000000 );
16    }
17
18    initial8BitCommunicationIsCompleted = false;
19
20    delay( 50 );
21
22    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
23                      DISPLAY_IR_FUNCTION_SET |
24                      DISPLAY_IR_FUNCTION_SET_8BITS );
25    delay( 5 );
26
27    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
28                      DISPLAY_IR_FUNCTION_SET |
29                      DISPLAY_IR_FUNCTION_SET_8BITS );
30    delay( 1 );
31
32    displayCodeWrite( DISPLAY_RS_INSTRUCTION,

```

```

33             DISPLAY_IR_FUNCTION_SET |
34             DISPLAY_IR_FUNCTION_SET_8BITS );
35     delay( 1 );
36
37     switch( display.connection ) {
38         case DISPLAY_CONNECTION_GPIO_8BITS:
39         case DISPLAY_CONNECTION_SPI:
40             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
41                             DISPLAY_IR_FUNCTION_SET |
42                             DISPLAY_IR_FUNCTION_SET_8BITS |
43                             DISPLAY_IR_FUNCTION_SET_2LINES |
44                             DISPLAY_IR_FUNCTION_SET_5x8DOTS );
45             delay( 1 );
46             break;
47
48         case DISPLAY_CONNECTION_GPIO_4BITS:
49         case DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER:
50             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
51                             DISPLAY_IR_FUNCTION_SET |
52                             DISPLAY_IR_FUNCTION_SET_4BITS );
53             delay( 1 );
54
55             initial8BitCommunicationIsCompleted = true;
56
57             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
58                             DISPLAY_IR_FUNCTION_SET |
59                             DISPLAY_IR_FUNCTION_SET_4BITS |
60                             DISPLAY_IR_FUNCTION_SET_2LINES |
61                             DISPLAY_IR_FUNCTION_SET_5x8DOTS );
62             delay( 1 );
63             break;
64     }

```

Code 6.34 New implementation of the function displayInit() (Part 1/2).

```

1     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
2                     DISPLAY_IR_DISPLAY_CONTROL |
3                     DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_OFF |
4                     DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
5                     DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
6     delay( 1 );
7
8     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
9                     DISPLAY_IR_CLEAR_DISPLAY );
10    delay( 1 );
11
12    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
13                     DISPLAY_IR_ENTRY_MODE_SET |
14                     DISPLAY_IR_ENTRY_MODE_SET_INCREMENT |
15                     DISPLAY_IR_ENTRY_MODE_SET_NO_SHIFT );
16    delay( 1 );
17
18    displayCodeWrite( DISPLAY_RS_INSTRUCTION,
19                     DISPLAY_IR_DISPLAY_CONTROL |
20                     DISPLAY_IR_DISPLAY_CONTROL_DISPLAY_ON |
21                     DISPLAY_IR_DISPLAY_CONTROL_CURSOR_OFF |
22                     DISPLAY_IR_DISPLAY_CONTROL_BLINK_OFF );
23    delay( 1 );
24 }

```

Code 6.35 New implementation of the function displayInit() (Part 2/2).

In Code 6.36, the new implementation of *displayCodeWrite()* is shown. The new code is between lines 15 and 30, where the case of DISPLAY_CONNECTION_SPI is addressed. On line 16, *spi.lock()* is used to acquire exclusive access to the SPI bus. On line 17, *spiSt7920ChipSelect* is set to ON. Line 18 assesses if the code corresponds to an instruction. If so, lines 19 to 21 are used to write the first byte to the SPI bus, following the format that was introduced in Figure 6.28. Otherwise, lines 23 to 25 are executed, to transfer the first byte with the RS bit set to 1 because ST7920_SPI_RS_DATA is defined as 0b000000010.

Line 26 is used to send the second byte using “*dataBus & 0b11110000*”. In this way, only the D7 to D4 bits of *dataBus* are transferred, followed by four zeros. Line 27 is used to send the third byte by means of “(*dataBus* << 4) & 0b11110000”. The <<4 bitwise operation is used to shift the bits of *dataBus* four positions to the left. In this way, the D3 to D0 bits are placed in the first part of the byte, as shown in the third byte of Figure 6.28. Line 28 is used to set *spiSt7920ChipSelect* to OFF. Finally, on line 29, *spiSt7920.unlock()* is used to release exclusive access to this SPI bus.

In Code 6.37 and Code 6.38, the implementation of *displayCharPositionWrite()* is shown. In line 3, an *if* statement is used to evaluate if *display.type* is DISPLAY_TYPE_LCD_HD44780. If so, the *switch* statement of line 4 is executed, which has the same code as in previous examples. Line 9 of Code 6.38 evaluates if *display.type* is DISPLAY_TYPE_GLCD_ST7920; in that case, the *switch* statement of line 10 is executed. It is important to note that *charPositionX/2* is used in lines 15, 23, 31, and 39. This is due to the reasons explained regarding Figure 6.27.



NOTE: In this example, the implementation of *displayStringWrite()* is not shown, since this function was not modified in this example. The modifications introduced in *displayPinWrite()* and *displayDataBusWrite()* were already discussed in Code 6.33.

```

1 static void displayCodeWrite( bool type, uint8_t dataBus )
2 {
3     switch( display.connection ) {
4         case DISPLAY_CONNECTION_GPIO_8BITS:
5         case DISPLAY_CONNECTION_GPIO_4BITS:
6         case DISPLAY_CONNECTION_I2C_PCF8574_IO_EXPANDER:
7             if ( type == DISPLAY_RS_INSTRUCTION )
8                 displayPinWrite( DISPLAY_PIN_RS, DISPLAY_RS_INSTRUCTION );
9             else
10                 displayPinWrite( DISPLAY_PIN_RS, DISPLAY_RS_DATA );
11             displayPinWrite( DISPLAY_PIN_RW, DISPLAY_RW_WRITE );
12             displayDataBusWrite( dataBus );
13             break;
14
15         case DISPLAY_CONNECTION_SPI:
16             spiSt7920.lock();
17             spiSt7920ChipSelect = ON;
18             if ( type == DISPLAY_RS_INSTRUCTION )
19                 spiSt7920.write( ST7920_SPI_SYNCHRONIZING_BIT_STRING |
20                                 ST7920_SPI_RW_WRITE |
21                                 ST7920_SPI_RS_INSTRUCTION );
22             else
23                 spiSt7920.write( ST7920_SPI_SYNCHRONIZING_BIT_STRING |
24                                 ST7920_SPI_RW_WRITE |

```

```

25                     ST7920_SPI_RS_DATA );
26     spiSt7920.write( dataBus & 0b11110000 );
27     spiSt7920.write( (dataBus<<4) & 0b11110000 );
28     spiSt7920ChipSelect = OFF;
29     spiSt7920.unlock();
30     break;
31 }
32 }
```

Code 6.36 New implementation of the function `displayCodeWrite()`.

```

1 void displayCharPositionWrite( uint8_t charPositionX, uint8_t charPositionY )
2 {
3     if( display.type == DISPLAY_TYPE_LCD_HD44780 ) {
4         switch( charPositionY ) {
5             case 0:
6                 displayCodeWrite( DISPLAY_RS_INSTRUCTION,
7                     DISPLAY_IR_SET_DDRAM_ADDR |
8                     ( DISPLAY_20x4_LINE1_FIRST_CHARACTER_ADDRESS +
9                         charPositionX ) );
10            delay( 1 );
11            break;
12
13         case 1:
14             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
15                 DISPLAY_IR_SET_DDRAM_ADDR |
16                 ( DISPLAY_20x4_LINE2_FIRST_CHARACTER_ADDRESS +
17                     charPositionX ) );
18            delay( 1 );
19            break;
20
21         case 2:
22             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
23                 DISPLAY_IR_SET_DDRAM_ADDR |
24                 ( DISPLAY_20x4_LINE3_FIRST_CHARACTER_ADDRESS +
25                     charPositionX ) );
26            delay( 1 );
27            break;
28     }
29 }
```

Code 6.37 New implementation of the function `displayCharPositionWrite()` (Part 1/2).

```

1         case 3:
2             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
3                 DISPLAY_IR_SET_DDRAM_ADDR |
4                 ( DISPLAY_20x4_LINE4_FIRST_CHARACTER_ADDRESS +
5                     charPositionX ) );
6             delay( 1 );
7             break;
8     }
9 } else if( display.type == DISPLAY_TYPE_GLCD_ST7920 ) {
10    switch( charPositionY ) {
11        case 0:
12            displayCodeWrite( DISPLAY_RS_INSTRUCTION,
13                DISPLAY_IR_SET_DDRAM_ADDR |
14                ( DISPLAY_ST7920_LINE1_FIRST_CHARACTER_ADDRESS +
15                    charPositionX/2 ) );
16            delay( 1 );
17            break;
18    }
19 }
```

```

19         case 1:
20             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
21                               DISPLAY_IR_SET_DDRAM_ADDR |
22                               ( DISPLAY_ST7920_LINE2_FIRST_CHARACTER_ADDRESS +
23                                 charPositionX/2 ) );
24             delay( 1 );
25             break;
26
27         case 2:
28             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
29                               DISPLAY_IR_SET_DDRAM_ADDR |
30                               ( DISPLAY_ST7920_LINE3_FIRST_CHARACTER_ADDRESS +
31                                 charPositionX/2 ) );
32             delay( 1 );
33             break;
34
35         case 3:
36             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
37                               DISPLAY_IR_SET_DDRAM_ADDR |
38                               ( DISPLAY_ST7920_LINE4_FIRST_CHARACTER_ADDRESS +
39                                 charPositionX/2 ) );
40             delay( 1 );
41             break;
42     }
43 }
44 }
```

Code 6.38 New implementation of the function `displayCharPositionWrite()` (Part 2/2).

Proposed Exercise

1. How should `displayInit()` be called if a character LCD display is used with a 4-bit mode connection?

Answer to the Exercise

1. The function call should be made as follows:

```
displayInit ( DISPLAY_TYPE_LCD_HD44780, DISPLAY_CONNECTION_GPIO_4BITS );
```

Example 6.5: Use of the Graphic Capabilities of the Graphical LCD Display

Objective

Introduce the use of the graphic mode of the graphical LCD display.

Summary of the Expected Behavior

The expected behavior is similar to the previous examples, the difference being that when the alarm is activated the display is changed to graphic mode, and an animation of a fire burning, together with a "FIRE ALARM ACTIVATED!" legend, is shown. When the alarm is deactivated, the display is configured again to character mode and its behavior returns to the behavior of the previous examples.

Test the Proposed Solution on the Board

Import the project “Example 6.5” using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Follow the same steps as indicated in the section “Test the Proposed Solution on the Board” of Example 6.1. The present temperature, gas detector, and alarm states should be shown on the display. When the alarm is activated, an animation composed of the four images shown in Figure 6.35 should be displayed.

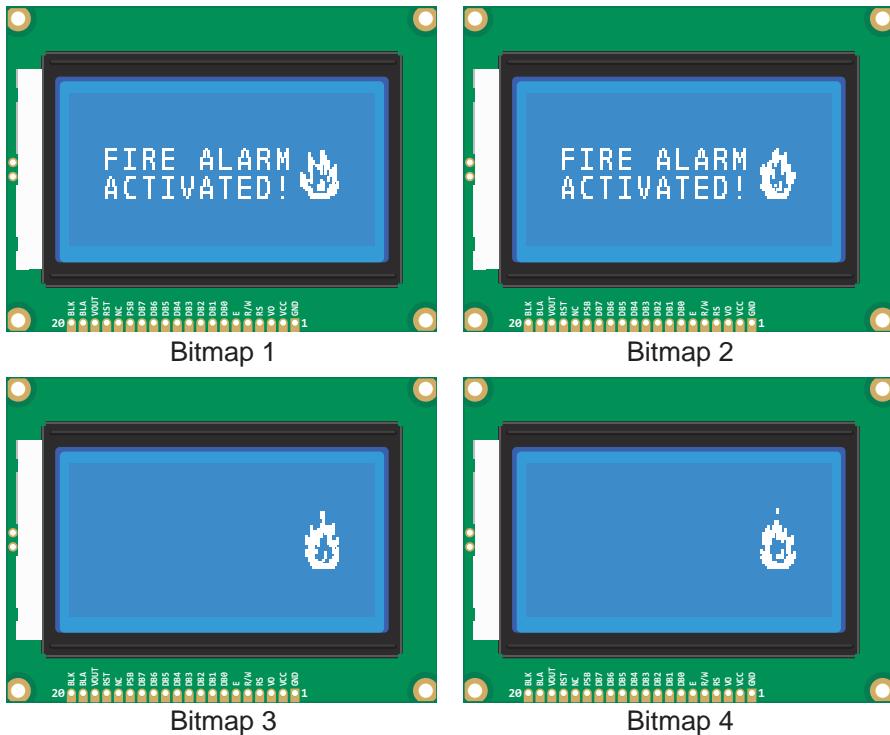


Figure 6.35 Frames of the animation that are shown when the alarm is activated.

Discussion of the Proposed Solution

In this example, the functions *displayModeWrite()* and *displayBitmapWrite()* are included. The former allows selecting between character mode and graphic mode, while the latter is used to send a bitmap to the graphical LCD display.

Implementation of the Proposed Solution

In Table 6.10, the sections where lines were added to the file *user_interface.cpp* are summarized. It can be seen that the file *GLCD_fire_alarm.h* has been included. This file has the bitmaps shown in Figure 6.35. There are also two new *#defines*, which are used to control the refresh time of the display: *DISPLAY_REFRESH_TIME_REPORT_MS*, which replaces the previous *#define* (removed), as shown in Table 6.11, and *DISPLAY_REFRESH_TIME_ALARM_MS*, which is used when the animation is shown.

Table 6.10 also shows the declaration of a new data type, *displayState_t*, which is used to control the display, as will be shown below. Three private global variables are declared, as shown in Table 6.10. Finally, four private functions are declared, which will be explained below.

Table 6.10 Sections in which lines were added to user_interface.cpp.

Section or function	Lines that were added
Libraries	#include "GLCD_fire_alarm.h"
Definitions	#define DISPLAY_REFRESH_TIME_REPORT_MS 1000 #define DISPLAY_REFRESH_TIME_ALARM_MS 300
Declaration of private data types	typedef enum{ DISPLAY_ALARM_STATE, DISPLAY_REPORT_STATE } displayState_t;
Declaration and initialization of private global variables	static displayState_t displayState = DISPLAY_REPORT_STATE; static int displayAlarmGraphicSequence = 0; static int displayRefreshTimeMs = DISPLAY_REFRESH_TIME_REPORT_MS;
Declarations (prototypes) of private functions	static void userInterfaceDisplayReportStateInit(); static void userInterfaceDisplayReportStateUpdate(); static void userInterfaceDisplayAlarmStateInit(); static void userInterfaceDisplayAlarmStateUpdate();

Table 6.11 Sections in which lines were removed from user_interface.cpp.

Section or function	Lines that were removed
Definitions	#define DISPLAY_REFRESH_TIME_MS 1000

In Code 6.39, the new public data type *displayMode_t* is shown. It has two valid values, DISPLAY_MODE_CHAR and DISPLAY_MODE_GRAPHIC. This data type is incorporated into the *display_t*, as shown in Code 6.39.

```

1  typedef enum {
2      DISPLAY_MODE_CHAR,
3      DISPLAY_MODE_GRAPHIC
4  } displayMode_t;
5
6  typedef struct {
7      displayConnection_t connection;
8      displayType_t type;
9      displayMode_t mode;
10 } display_t;

```

Code 6.39 New private data types declared in display.h.

In Code 6.40, the new #defines that are introduced in *display.cpp* are shown. DISPLAY_IR_SET_GDRAM_ADDR is used to establish the GDRAM address. The other #defines are used to implement the instructions shown in blue in Table 6.6.

```

1 #define DISPLAY_IR_SET_GDRAM_ADDR 0b10000000
2
3 #define DISPLAY_IR_FUNCTION_SET_EXTENDED_INSTRUCCION_SET 0b000000100
4 #define DISPLAY_IR_FUNCTION_SET_BASIC_INSTRUCCION_SET 0b000000000
5 #define DISPLAY_IR_FUNCTION_SET_GRAPHIC_DISPLAY_ON 0b000000010
6 #define DISPLAY_IR_FUNCTION_SET_GRAPHIC_DISPLAY_OFF 0b000000000

```

Code 6.40 Declaration of new private #defines in *display.cpp*.

In Code 6.41, the new implementation of the function *userInterfaceDisplayUpdate()* is shown. It is now divided into two different states. In the case of DISPLAY_REPORT_STATE, the behavior is similar to the previous examples, except that the Alarm ON state is not shown. In order to implement this functionality, the new function *userInterfaceDisplayReportStateUpdate()* is used.

When the siren is active, *userInterfaceDisplayAlarmStateInit()* is called (line 15), which changes the state of the variable *displayState* to DISPLAY_ALARM_STATE and makes the display change to graphic mode.

When the siren is not activated, the function *userInterfaceDisplayReportStateInit()* is called (line 23). As discussed below, this function changes the state of the variable *displayState* to DISPLAY_REPORT_STATE and also makes the display change to character mode.

Finally, note that *displayRefreshTimeMs* is configured in *userInterfaceDisplayAlarmStateInit()* and *userInterfaceDisplayReportStateInit()*, because different refresh time values are used depending on the state of the display (DISPLAY_ALARM_STATE or DISPLAY_REPORT_STATE).

```

1 static void userInterfaceDisplayUpdate()
2 {
3     static int accumulatedDisplayTime = 0;
4
5     if( accumulatedDisplayTime >=
6         displayRefreshTimeMs ) {
7
8         accumulatedDisplayTime = 0;
9
10        switch ( displayState ) {
11            case DISPLAY_REPORT_STATE:
12                userInterfaceDisplayReportStateUpdate();
13
14                if ( sirenStateRead() ) {
15                    userInterfaceDisplayAlarmStateInit();
16                }
17                break;
18
19            case DISPLAY_ALARM_STATE:
20                userInterfaceDisplayAlarmStateUpdate();
21
22                if ( !sirenStateRead() ) {
23                    userInterfaceDisplayReportStateInit();
24                }
25                break;

```

```

26         default:
27             userInterfaceDisplayReportStateInit();
28             break;
29     }
30 }
31 } else {
32     accumulatedDisplayTime =
33         accumulatedDisplayTime + SYSTEM_TIME_INCREMENT_MS;
34 }
35 }
36 }
```

Code 6.41 New implementation of the function `userInterfaceDisplayUpdate()`.

The function `userInterfaceDisplayInit()`, shown in Code 6.42, is simpler than in the previous examples, as some parts of the initialization are made in `userInterfaceDisplayReportStateInit()`. Note that in the previous examples the strings “Temperature:”, “Gas:”, and “Alarm:” were written once at the beginning and not modified again. In this example, those strings should be output whenever the display returns to the DISPLAY_REPORT_STATE.

In this example, the implementation of `displayInit()` is not shown, as there is only one new line, which is used to set the new field `mode` of the variable `display` to DISPLAY_MODE_CHAR. The display is always initialized in character mode, and `displayModeWrite()` should be used to change to graphic mode.

```

1 static void userInterfaceDisplayInit()
2 {
3     displayInit( DISPLAY_TYPE_GLCD_ST7920, DISPLAY_CONNECTION_SPI );
4     userInterfaceDisplayReportStateInit();
5 }
```

Code 6.42 New implementation of the function `userInterfaceDisplayInit()`.

In Code 6.43, the new function `userInterfaceDisplayReportStateInit()` is shown. It has some statements that were previously in the function `userInterfaceDisplayInit()`. This function also configures the display state, the refresh time, and the display to char mode, which is established by means of `displayModeWrite()` on line 6. It can be seen that there is also a new function, `displayClear()`, which is used to clear everything that might be written on the display.

```

1 static void userInterfaceDisplayReportStateInit()
2 {
3     displayState = DISPLAY_REPORT_STATE;
4     displayRefreshTimeMs = DISPLAY_REFRESH_TIME_REPORT_MS;
5
6     displayModeWrite( DISPLAY_MODE_CHAR );
7
8     displayClear();
9
10    displayCharPositionWrite( 0,0 );
11    displayStringWrite( "Temperature:" );
12
13    displayCharPositionWrite( 0,1 );
14    displayStringWrite( "Gas:" );
15
16    displayCharPositionWrite( 0,2 );
17    displayStringWrite( "Alarm:" );
18 }
19 }
```

Code 6.43 Implementation of the function `userInterfaceDisplayReportStateInit()`.

The public function *displayClear()* is implemented in *display.cpp*, as shown in Code 6.44. In line 3, *displayCodeWrite()* is used to send the “Display Clear” instruction to the display. In line 5, a 2-millisecond delay is implemented.

```
1 void displayClear( void )
2 {
3     displayCodeWrite( DISPLAY_RS_INSTRUCTION,
4                       DISPLAY_IR_CLEAR_DISPLAY );
5     delay( 2 );
6 }
```

Code 6.44 Implementation of the function *displayClear()*.

In Code 6.45, *userInterfaceDisplayReportStateUpdate()* is shown. It mostly includes code that previously was in *userInterfaceDisplayUpdate()*.

```
1 static void userInterfaceDisplayReportStateUpdate()
2 {
3     char temperatureString[3] = "";
4
5     sprintf(temperatureString, "% .0f", temperatureSensorReadCelsius());
6     displayCharPositionWrite( 12,0 );
7     displayStringWrite( temperatureString );
8     displayCharPositionWrite( 14,0 );
9     displayStringWrite( "'C" );
10
11    displayCharPositionWrite( 4,1 );
12
13    if ( gasDetectorStateRead() ) {
14        displayStringWrite( "Detected      " );
15    } else {
16        displayStringWrite( "Not Detected" );
17    }
18    displayCharPositionWrite( 6,2 );
19    displayStringWrite( "OFF" );
20 }
```

Code 6.45 Implementation of the function *userInterfaceDisplayReportStateUpdate()*.

The implementation of *userInterfaceDisplayAlarmStateInit()* is shown in Code 6.46. This function initializes the display by means of configuring the state of the display to DISPLAY_ALARM_STATE, changing the refresh time, clearing the display, and changing to graphic mode. The variable *displayAlarmGraphicSequence*, which is related to the animation that is displayed, is also initialized.

```
1 static void userInterfaceDisplayAlarmStateInit()
2 {
3     displayState = DISPLAY_ALARM_STATE;
4     displayRefreshTimeMs = DISPLAY_REFRESH_TIME_ALARM_MS;
5
6     displayClear();
7
8     displayModeWrite( DISPLAY_MODE_GRAPHIC );
9
10    displayAlarmGraphicSequence = 0;
11 }
```

Code 6.46 Implementation of the function *userInterfaceDisplayAlarmStateInit()*.

The images that comprise the fire burning animation are declared in a new filename, *GLCD_fire_alarm.h*, which is now included in the *display* module. The most relevant lines of this file are shown in Code 6.47 and Code 6.48. Code 6.47 shows the declaration of the array *GLCD_ClearScreen*. It has 1024 elements in correspondence with the 8 columns, each having two bytes, and 64 rows that were introduced in Figure 6.31 (i.e., $1024 = 8 \times 2 \times 64$). All its elements are declared as zero using hexadecimal notation (0x00). For the sake of brevity, only some of the elements are shown.

In Code 6.48, the declaration of *GLCD_fire_alarm* is shown. It has four parts, each having 1024 elements, as is indicated in line 1. For the sake of brevity, only some of the elements are shown with the aim of illustrating that each of the four images that compose the fire burning animation is defined using a different set of bytes.

```

1  uint8_t GLCD_ClearScreen[1024] = {
2  0x00, 0x00,
3  0x00, 0x00,
4  ...
5  0x00, 0x00,
6  };

```

Code 6.47 Summary of the content of *GLCD_fire_alarm.h* (Part 1/2).

```

1  uint8_t GLCD_fire_alarm[4][1024] = {
2  {
3  0x00, 0x00,
4  ...
5  0x00, 0x01, 0xF1, 0xF1, 0xE1, 0xF0, 0x00, 0x41, 0x00, 0x41, 0xE1, 0x10, 0x00, 0x00,
6  ...
7  0x00, 0x00,
8  },
9  {
10  0x00, 0x00,
11  ...
12  0x00, 0x01, 0x00, 0x41, 0x11, 0x00, 0xA1, 0x00, 0xA1, 0x11, 0x10, 0x00, 0xC0,
13  ...
14  0x00, 0x00,
15  },
16  {
17  0x00, 0x00,
18  ...
19  0x00, 0x00, 0x04, 0x7E, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
20  ...
21  0x00, 0x00,
22  },
23  {
24  0x00, 0x00,
25  ...
26  0x00, 0x00, 0x1F, 0x79, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
27  ...
28  0x00, 0x00,
29  }
30  };

```

Code 6.48 Summary of the content of *GLCD_fire_alarm.h* (Part 2/2).

The function *userInterfaceDisplayAlarmStateUpdate()*, shown in Code 6.49, writes each of the four images that comprise the fire burning animation to the LCD shown in Figure 6.35. It can be seen that the function *displayBitmapWrite()* is used to write each of the images, which are defined in the *GLCD_fire_alarm.h* file.

```

1 static void userInterfaceDisplayAlarmStateUpdate()
2 {
3     switch( displayAlarmGraphicSequence ){
4         case 0:
5             displayBitmapWrite( GLCD_fire_alarm[0] );
6             displayAlarmGraphicSequence++;
7             break;
8         case 1:
9             displayBitmapWrite( GLCD_fire_alarm[1] );
10            displayAlarmGraphicSequence++;
11            break;
12        case 2:
13            displayBitmapWrite( GLCD_fire_alarm[2] );
14            displayAlarmGraphicSequence++;
15            break;
16        case 3:
17            displayBitmapWrite( GLCD_fire_alarm[3] );
18            displayAlarmGraphicSequence = 0;
19            break;
20        default:
21            displayBitmapWrite( GLCD_ClearScreen );
22            displayAlarmGraphicSequence = 1;
23            break;
24    }
25 }
```

Code 6.49 Implementation of the function *userInterfaceDisplayAlarmStateUpdate()*.

In Code 6.50, the implementation of the function *displayBitmapWrite()* of the *display* module is shown. The *for* loop on line 4 is used to increase the y coordinate of the image. If y is less than 32 (line 5), the top half of the screen is drawn using the *for* loop in lines 6 to 17. In extended instruction mode, vertical and horizontal coordinates must be specified before sending data. The vertical coordinate of the screen is specified first (line 7), then the horizontal coordinate of the screen is specified (line 10). On lines 13 and 15, the upper and lower bytes are sent to the coordinate (as shown in Figure 6.31). If y is not less than 32, then the operation is very similar, but the addresses are changed as shown in lines 20 and 23 of Code 6.50, following the ideas that were introduced in Figure 6.31.

```

1 void displayBitmapWrite( uint8_t* bitmap )
2 {
3     uint8_t x, y;
4     for( y=0; y<64; y++ ) {
5         if ( y < 32 ) {
6             for( x = 0; x < 8; x++ ) {
7                 displayCodeWrite( DISPLAY_RS_INSTRUCTION,
8                     DISPLAY_IR_SET_GDRAM_ADDR |
9                     y );
10                displayCodeWrite( DISPLAY_RS_INSTRUCTION,
11                    DISPLAY_IR_SET_GDRAM_ADDR |
12                    x );
13                displayCodeWrite( DISPLAY_RS_DATA,
```

```

14                     bitmap[16*y + 2*x] );
15             displayCodeWrite(DISPLAY_RS_DATA,
16                     bitmap[16*y + 2*x+1] );
17         }
18     } else {
19         for( x = 0; x < 8; x++ ) {
20             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
21                             DISPLAY_IR_SET_GDRAM_ADDR |
22                             (y-32) );
23             displayCodeWrite( DISPLAY_RS_INSTRUCTION,
24                             DISPLAY_IR_SET_GDRAM_ADDR |
25                             (x+8) );
26             displayCodeWrite(DISPLAY_RS_DATA,
27                             bitmap[16*y + 2*x]);
28             displayCodeWrite(DISPLAY_RS_DATA,
29                             bitmap[16*y + 2*x+1]);
30         }
31     }
32 }
33 }
```

Code 6.50 Implementation of the function *displayBitmapWrite()*.

NOTE: The function *displayBitmapWrite()* involves sending hundreds of bytes to the graphical display, which interferes with the time management of the strobe light and the siren. In this way, when gas and over temperature are detected, the time off and on of the strobe light and the siren is not always 100 ms as expected. This problem will be addressed in the next chapters as new concepts are introduced.

In Code 6.51, the implementation of *displayModeWrite()* is shown. On line 3, the *mode* parameter is compared to DISPLAY_MODE_GRAPHIC. If they are equal, the display is configured to graphic mode by means of the statements between lines 5 and 15. Otherwise, if *mode* is equal to DISPLAY_MODE_CHAR, then the display is configured to character mode by means of the statements in lines 17 to 22.

```

1 void displayModeWrite( displayMode_t mode )
2 {
3     if ( mode == DISPLAY_MODE_GRAPHIC )
4     {
5         displayCodeWrite( DISPLAY_RS_INSTRUCTION,
6                         DISPLAY_IR_FUNCTION_SET |
7                         DISPLAY_IR_FUNCTION_SET_8BITS |
8                         DISPLAY_IR_FUNCTION_SET_EXTENDED_INSTRUCCION_SET );
9         delay(1);
10        displayCodeWrite( DISPLAY_RS_INSTRUCTION,
11                        DISPLAY_IR_FUNCTION_SET |
12                        DISPLAY_IR_FUNCTION_SET_8BITS |
13                        DISPLAY_IR_FUNCTION_SET_EXTENDED_INSTRUCCION_SET |
14                        DISPLAY_IR_FUNCTION_SET_GRAPHIC_DISPLAY_ON );
15        delay(1);
16    } else if ( mode == DISPLAY_MODE_CHAR ) {
17        displayCodeWrite( DISPLAY_RS_INSTRUCTION,
18                        DISPLAY_IR_FUNCTION_SET |
19                        DISPLAY_IR_FUNCTION_SET_8BITS |
20                        DISPLAY_IR_FUNCTION_SET_BASIC_INSTRUCCION_SET |
21                        DISPLAY_IR_FUNCTION_SET_GRAPHIC_DISPLAY_OFF );
22        delay(1);
23    }
24 }
```

Code 6.51 Implementation of the function *displayModeWrite()*.

Proposed Exercise

1. How can a blank screen be added at the end of the animation shown when the alarm is activated?

Answer to the Exercise

1. The function `userInterfaceDisplayAlarmStateUpdate()` should be modified as shown in Code 6.52.

```
1 static void userInterfaceDisplayAlarmStateUpdate( )
2 {
3     switch( displayAlarmGraphicSequence ){
4         case 0:
5             displayBitmapWrite( GLCD_fire_alarm[0] );
6             displayAlarmGraphicSequence++;
7             break;
8         case 1:
9             displayBitmapWrite( GLCD_fire_alarm[1] );
10            displayAlarmGraphicSequence++;
11            break;
12        case 2:
13            displayBitmapWrite( GLCD_fire_alarm[2] );
14            displayAlarmGraphicSequence++;
15            break;
16        case 3:
17            displayBitmapWrite( GLCD_fire_alarm[3] );
18            displayAlarmGraphicSequence++;
19            break;
20        case 4:
21            displayCommandWrite( GLCD_ClearScreen );
22            delay(2);
23            displayAlarmGraphicSequence = 0;
24            break;
25        default:
26            displayBitmapWrite( GLCD_ClearScreen );
27            displayAlarmGraphicSequence = 1;
28            break;
29    }
30 }
```

Code 6.52 Implementation of the function `userInterfaceDisplayAlarmStateUpdate()`.

6.3 Under the Hood

6.3.1 Comparison between UART, SPI, and I2C

A comparison between the UART (which was introduced in chapter 2) and the SPI and I2C buses that were introduced in this chapter is shown in Table 6.12. It should be noted that all these communication interfaces require a wired connection between the devices. In Chapters 10 and 11, wireless communications will be introduced.

Table 6.12 Comparison between UART, SPI, and I2C.

	UART	SPI	I2C
Connectivity characteristics	Point-to-point connection (GND, TxD, and RxD connections)	Difficult to connect many devices (GND, SCLK, MOSI, MISO, SS)	Easy to chain many devices (GND, SCL, and SDA)
Maximum devices	2	Not defined (usually less than 10)	127
Maximum distance	Highest (up to 50 feet / 15 meters)	Lowest (up to 10 feet / 3 meters)	Medium (up to 33 feet / 10 meters)
Maximum data rate	Lowest (up to 460 kbps)	Highest (up to 20 Mbps)	Medium (up to 3.4 Mbps)
Number of managers	None	One	One or more
Parity bit	Available	No	No
Acknowledge bit	No	No	Yes
Advantages	Simplicity	Fastest of all these alternatives	Only two wires are required
Disadvantages	Can only connect two devices	Requires multiple SS wires	Slower when compared to SPI



NOTE: The values shown in Table 6.12 might not be available in some devices and/or might not be attainable in real-life implementations.



WARNING: There is usually a trade-off between distance and data rate. For example, the maximum length of an I2C link is about 1 meter at 100 kbps and about 10 meters at 10 kbps.

Proposed Exercises

- What bus would seem to be the most appropriate for a wired connection of 20 sensors to three microcontrollers?
- A 1 kbps data rate sensor is placed 12 meters away from the microcontroller. Which bus best suits this situation?
- Which bus seems most appropriate to connect a 10 GB SD memory card to a microcontroller?

Answers to the Exercises

- According to Table 7.3, the most appropriate bus seems to be I2C. The data rate of the sensors should be checked.
- Given that there is only one sensor connected at a low data rate and considering the large distance, UART is most appropriate.

3. In this case, the data rate is critical, while the distance is very short, so SPI is the most appropriate, as will be seen in Chapter 9.

6.4 Case Study

6.4.1 LCD Usage in Mbed-Based Projects

In this chapter, a character-based LCD display and a graphical LCD display were connected to the NUCLEO board using 4-bit and 8-bit modes, the I2C bus, and the SPI bus. In Figure 6.36, some examples of other systems based on Mbed that make use of LCD displays are shown.



Figure 6.36 Examples of other systems based on Mbed that make use of LCD displays.

The system on the left of Figure 6.36 is a solar charge controller that makes use of a character-based LCD display [12]. It is interesting to note that it is provided with a matrix keypad, and its information can be accessed by means of a smartphone application. In Chapter 10, the smart home system will be configured with a BLE connection and a smartphone app.

The system on the right of Figure 6.36 is a game console with a graphical LCD display [13]. The game console is the first example in this book where the power supply is a set of batteries. Power consumption becomes a critical issue in this type of system.

Proposed Exercises

1. What is the resolution of the graphical LCD display that is used in the game console? How does this resolution compare to the resolution of the graphical LCD used in the smart home system?

2. What batteries are used by the game console? How long will these batteries last if the current consumption is about 70 mA and the batteries are rated as 600 mAh?

Answers to the Exercises

1. In one of the images available in [13] it can be seen that the resolution of the LCD graphic display of the game console is 220×176 . This resolution is greater than the resolution of the LCD graphic display of the smart home system (128×64).
2. The game console uses Li-Po batteries, according to one of the images available in [13]. In that image, it is also indicated that the battery life is about eight to ten hours. Considering a current consumption of 70 mA, a battery rated as 600 mAh will last for about 8 hours ($600 \text{ mAh} / 70 \text{ mA}$).

References

- [1] “16x2 LCD Module_Pinout, Diagrams, Description & Datasheet” Accessed July 9, 2021.
<https://components101.com/displays/16x2-lcd-pinout-datasheet>
- [2] “HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver)”. Accessed July 9, 2021.
<https://www.sparkfun.com/datasheets/LCD/HD44780.pdf>
- [3] “GitHub - armBookCodeExamples/Directory”. Accessed July 9, 2021.
<https://github.com/armBookCodeExamples/Directory/>
- [4] “NUCLEO-F429ZI | Mbed”. Accessed July 9, 2021.
<https://os.mbed.com/platforms/ST-Nucleo-F429ZI/>
- [5] “ASCII | Wikipedia”. Accessed July 9, 2021.
<https://en.wikipedia.org/wiki/ASCII>
- [6] “I2C Serial Interface Adapter Module for LCD”. Accessed July 9, 2021.
<https://components101.com/modules/i2c-serial-interface-adapter-module>
- [7] “PCF8574 Remote 8-Bit I/O Expander for I2C Bus”. Accessed July 9, 2021.
<https://www.ti.com/lit/ds/symlink/pcf8574.pdf>
- [8] “Addressing - I2C Bus”. Accessed July 9, 2021.
<https://www.i2c-bus.org/addressing/>
- [9] “ST7290 GLCD Pinout, Features, Interfacing & Datasheet”. Accessed July 9, 2021.
<https://components101.com/displays/st7290-graphical-lcd>
- [10] “ST7920 Chinese Fonts built in LCD controller/driver”. Accessed July 9, 2021.
<https://pdf1.alldatasheet.es/datasheet-pdf/view/326219/SITRONIX/ST7920.html>

[11] "KeyStone Architecture | Serial Peripheral Interface (SPI)". Accessed July 9, 2021.

<https://www.ti.com/lit/ug/sprugp2a/sprugp2a.pdf>

[12] "Solar Charge Controller | Mbed". Accessed July 9, 2021.

<https://os.mbed.com/built-with-mbed/solar-charge-controller/>

[13] "Game Console | Mbed". Accessed July 9, 2021.

<https://os.mbed.com/built-with-mbed/game-console/>