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.
- Develop programs that implement finite-state machines 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. It is common to start by drawing the diagram of the FSM, analyze and adjust the behavior of the system using the diagram. Only when the behavior is as expected is the corresponding code implemented.

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 more than 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, the aim is to allow the use of numeric codes for clearing the alarm, and to improve the functionality of the smart home system by logging the alarm events. A keypad is proposed to replace buttons D2 to D7 to allow for numeric codes with a simple user interface. The previous smart home functions will be retained and 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. This is shown in Figure 4.1.

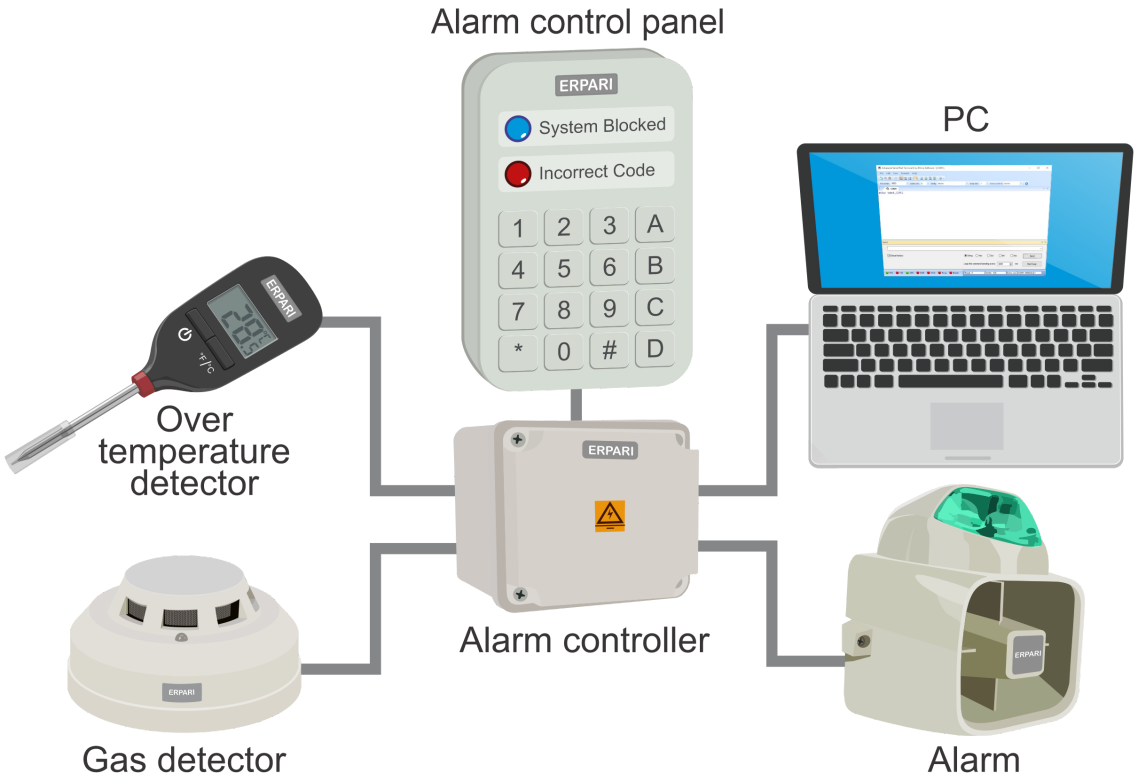

Figure 4.1. The smart home system is now connected to a matrix keypad.

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 in previous chapters to D4–D7 by the matrix keypad. The button connected to D2 will be replaced by the B1 USER button. In this way in the setup used in this chapter no buttons are connected to the NUCLEO board, as can be seen in Figure 4.2.

Note: The buttons connected to D2 and D4-D7 can be removed from the breadboard or can be left there, because they will not interfere with the system functionality. The advantage of removing the buttons is that the number of connections and elements is reduced. The advantage of not removing the buttons is that the examples introduced in previous chapters can be tried again, because the buttons are still connected.

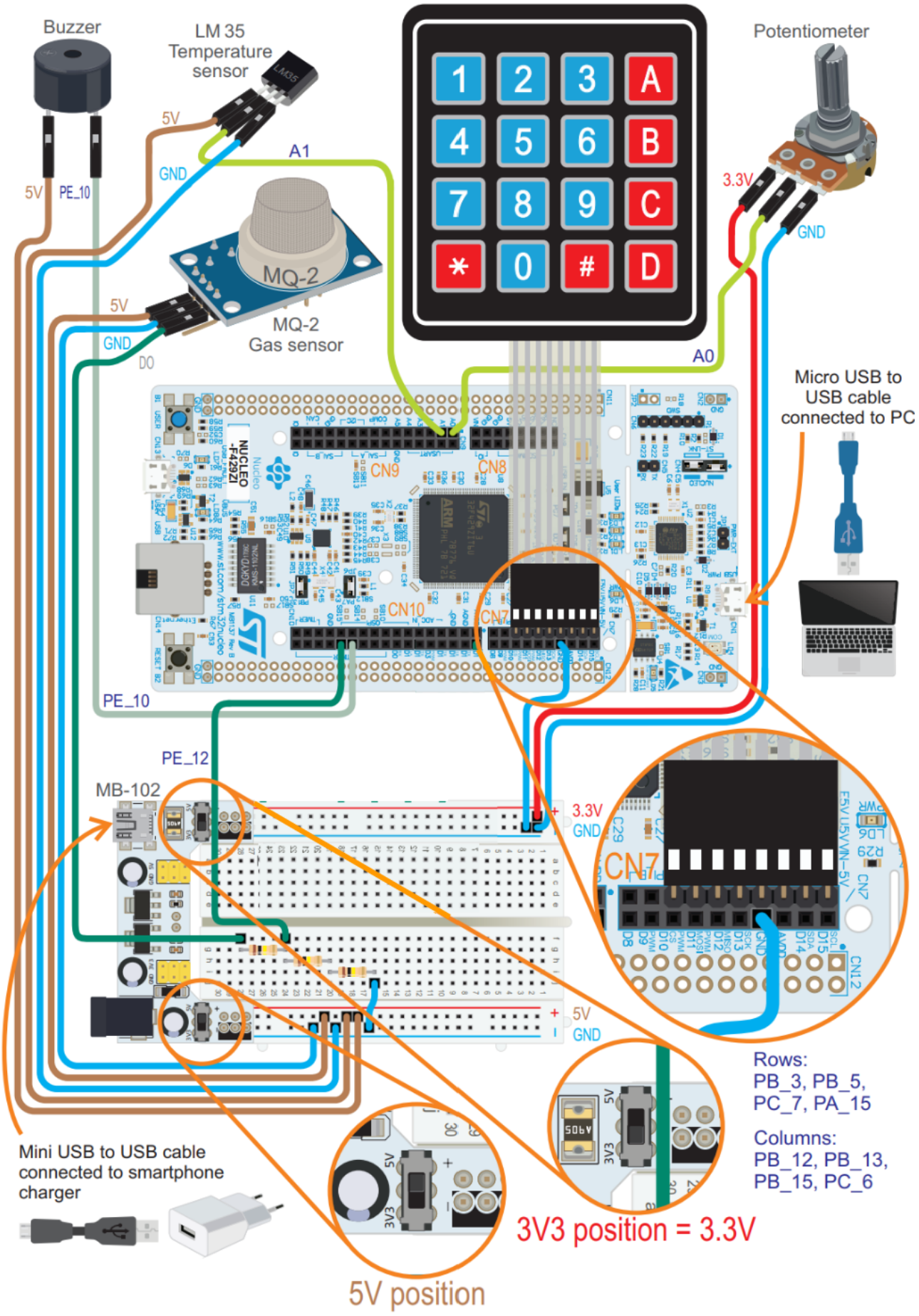

Figure 4.2. The smart home system is now connected to a matrix keypad.

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.

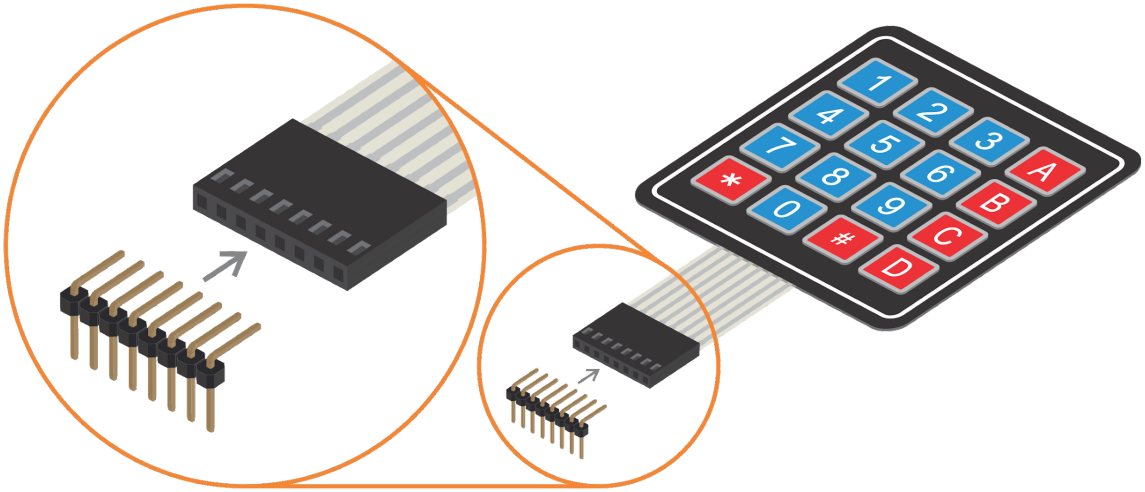

Figure 4.3. Detail showing how to prepare the matrix keypad connector using a pin header.

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).

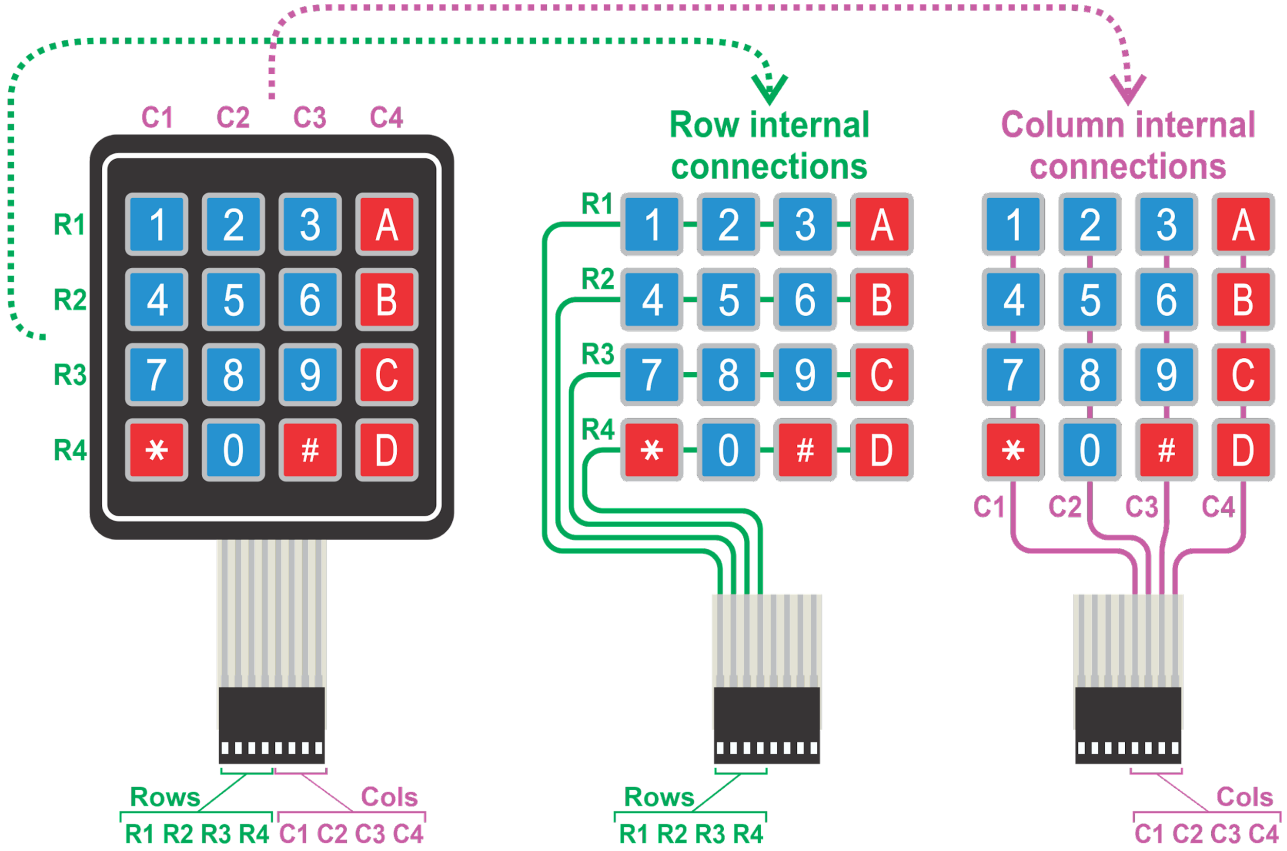

Figure 4.4. Diagram of the connections of the matrix keypad.

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.

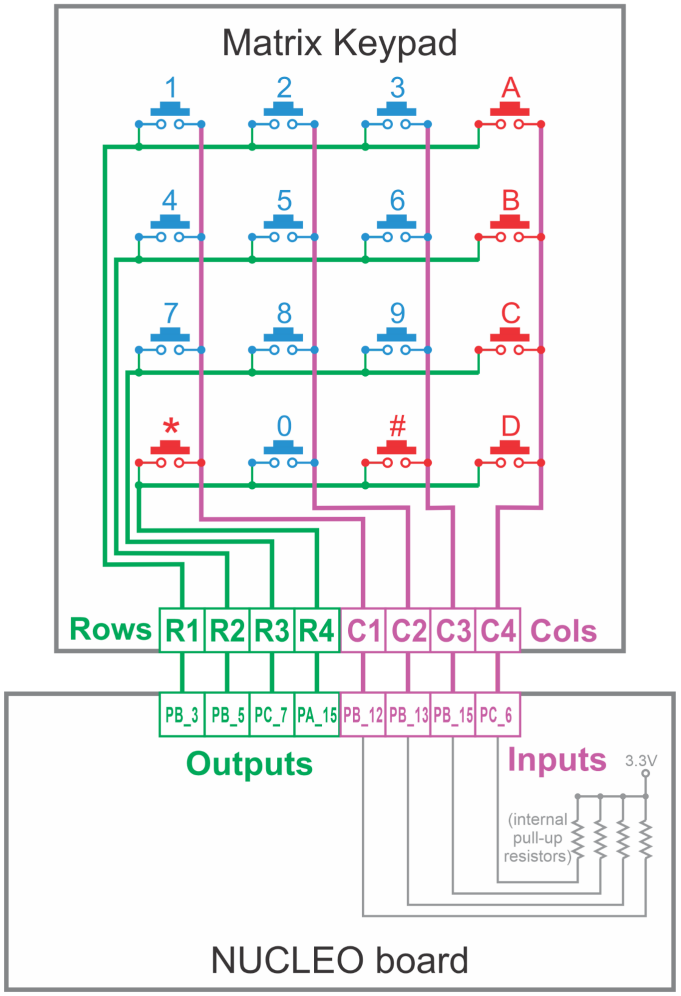

Figure 4.5. Diagram of the connections between the matrix keypad and the NUCLEO board.

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.

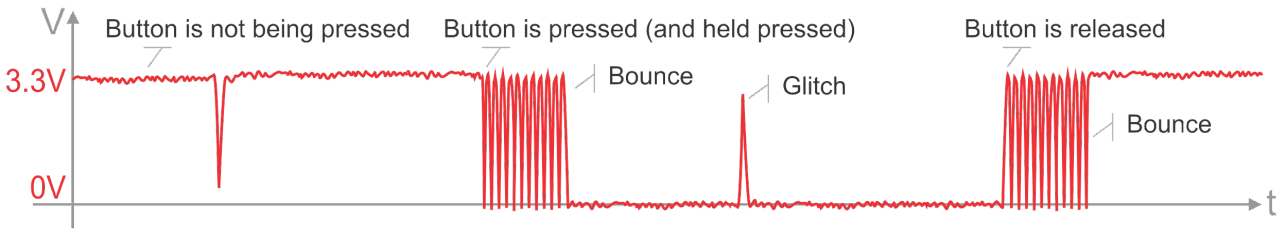

Figure 4.6. Voltage signal over time for a given button, including typical glitches and bounces.

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.

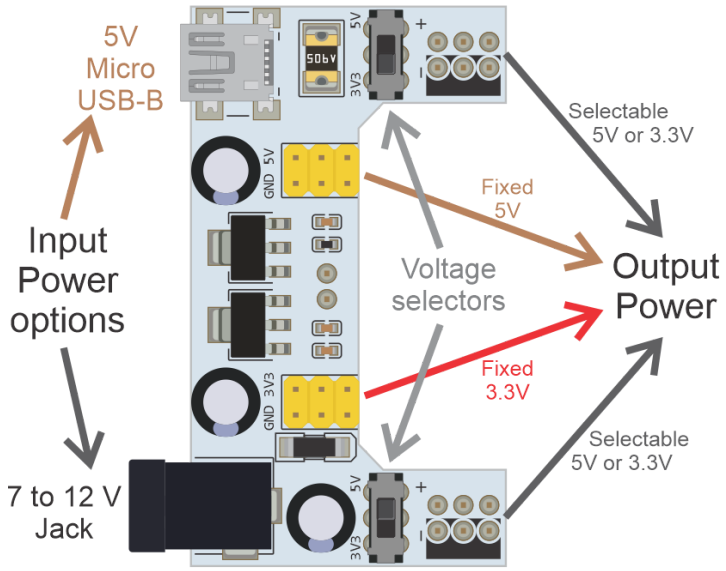

Figure 4.7. Diagram of the MB102 module.

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, FSMs diagrams and the implementation of FSMs.

Summary of the expected behavior

To turn off the incorrect code LED the Enter button ("#" key of the keypad) must be double pressed (pressed twice consecutively). This replaces entering a code simultaneously pressing the buttons connected to D4-D7 (A+B+C+D) as implemented in Chapter 1. The Alarm Test button is now implemented using the B1 USER button of the NUCLEO board.

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 (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. Double press the Enter button ("#" key of the keypad). The Alarm LED should turn off.

> **Note:** In this example by pressing key "#" on the keypad the Alarm LED will turn off, no matter if other keys were pressed in the keypad or not. The Incorrect Code LED will only turn on if an incorrect code is entered using command "4" in the PC keyboard.

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").

This is shown in Figure 4.8, where the voltage at a given button over time is shown with a red line, following the diagram that was introduced in section 4.2.1. The signal is sampled regularly at the rate given by TIME_INCREMENT_MS, as indicated in Figure 4.8. Initially (at t_0) the button is released, the voltage is 3.3 V and the finite-state machine (FSM) is in the BUTTON_UP state (as indicated in 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 after waiting for DEBOUNCE_BUTTON_TIME_MS it is determined that it was not an actual change in the button state .

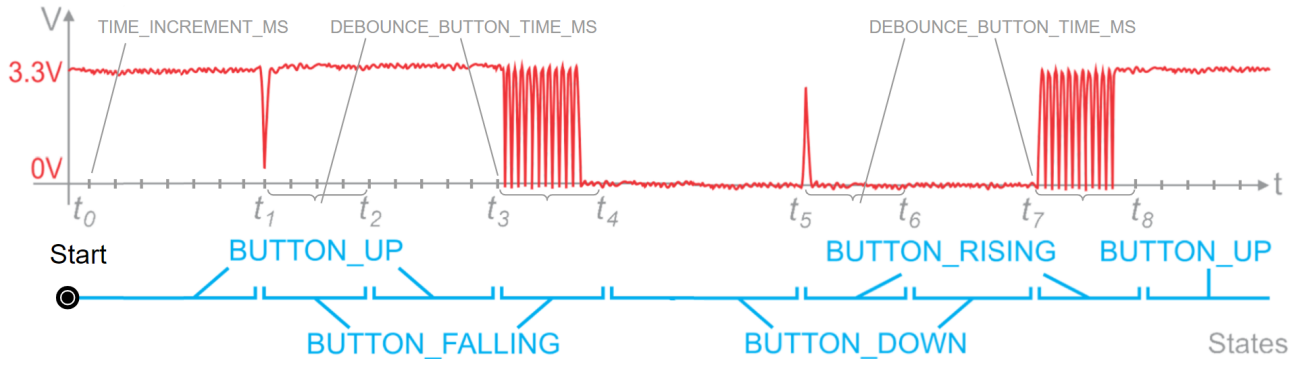

Figure 4.8. Voltage signal over time for a given button as it is pressed or released.

At t_3, the bounce in the voltage signal is 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 gets back to BUTTON_DOWN at t_6.

At t_7, there is a transition from pressed to released, which is accompanied by a bounce in the signal. It can be seen that the FSM state changed to BUTTON_RISING. After the DEBOUNCE_BUTTON_TIME_MS, at t_8, the signal is stable at 3.3 V, and, therefore the FSM state changes to BUTTON_UP.

Note: As shown in Figure 4.8, the transition from being "up" to being "down" (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.

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 4.9 shows a state diagram for the FSM discussed above Figure 4.8. This is the FSM that will be implemented in this Example 4.1.

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.

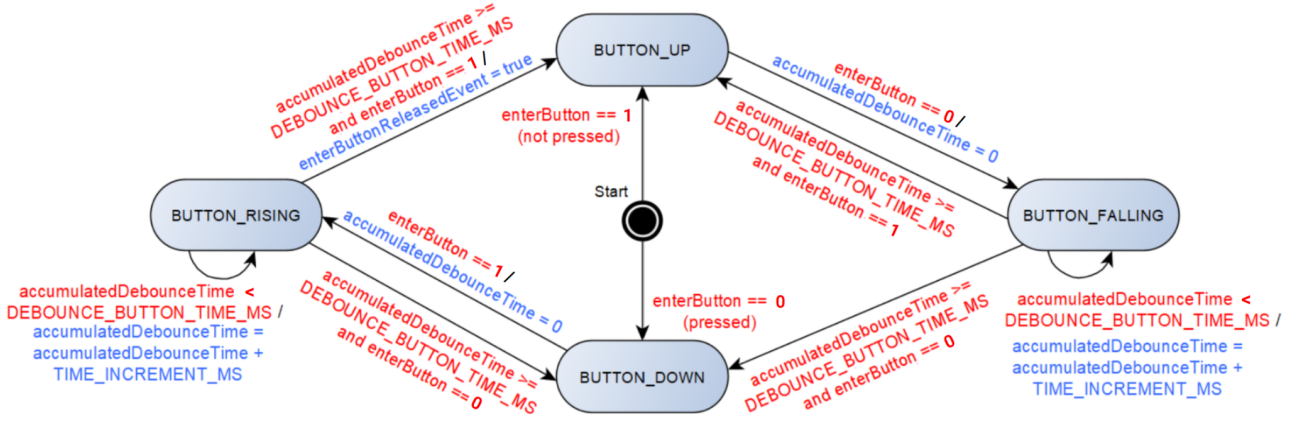

Figure 4.9. Diagram of the finite-state machine implemented in Example 4.1.

The four states of the FSM that are implemented in this example 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 1, implying that the button is initially not pressed) or to BUTTON_DOWN (if *enterButton* is 0, implying that the button is initially pressed). This will be implemented in this Example 4.1 in a function named *debounceButtonInit()*. 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 1 (implying that the button is initially not pressed) and, therefore, the first state is BUTTON_UP. By referring back to Figure 4.9, if the button is pressed ("enterButton == 0") there will be a transition from the BUTTON_UP state to the BUTTON_FALLING state, and *accumulatedDebounceTime* will be set to zero. Recall that the condition that triggers a given transition is indicated in red color, while the actions performed during the transition are indicated in blue color.

Once in the BUTTON_FALLING state, Figure 4.9 can be analyzed to determine all the possible transitions. It can be seen that in order to have a transition, *accumulatedDebounceTime* has to reach DEBOUNCE_BUTTON_TIME_MS. In that situation, if *enterButton* is 0 (i.e., the button is pressed), the transition is to the BUTTON_DOWN state, and if *enterButton* is 1 (i.e., the button is not pressed), the transition is to the BUTTON_UP state. While in BUTTON_FALLING state, *accumulatedDebounceTime* is incremented by TIME_INCREMENT_MS every 10 ms. This is indicated by the re-entering arrow below BUTTON_FALLING.

The only possible transition from BUTTON_DOWN is to BUTTON_RISING. This transition takes place if *enterButton* == 1, as is shown in Figure 4.9.

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. The only difference is that in the BUTTON_FALLING state, the variable *enterButtonReleasedEvent()* is set to true. This 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 “Mealy machine”. It is beyond the scope of this book to introduce other representations of FSMs.

Tip: It is common to start by drawing the diagram of the FSM, analyze and adjust the behavior of the system using the diagram. Only when the behavior is as expected is the corresponding code implemented. This process is shown by means of a complete example in Chapter 12.

Implementation of the proposed solution

Table 4.2 shows the objects that were modified from Example 3.5. The Alarm Test button is now the B1 USER button and the Enter button is now PB_15, in order to implement the reading of the “#”, as can be seen on Figure 4.5.

Table 4.2. *Declaration of public global objects that were modified from Example 3.5.*

Declaration in Example 3.5	Declaration in Example 4.1
`DigitalIn alarmTestButton(D2);` `DigitalIn enterButton(BUTTON1);`	`DigitalIn alarmTestButton(BUTTON1);` `DigitalIn enterButton(PB_15);`

Table 4.3 shows the DEBOUNCE_BUTTON_TIME_MS #define that is used to implement the debounce of the Enter 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 4.3. 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.

Following Table 4.3, a DigitalOut object named *outputRow4* was added to enable a low state on row R4 (see Figure 4.5) which is necessary to assess if the key “#” was pressed or not, by means of reading PB_15. In this way the Enter button reading is implemented in this example.

The variable *accumulatedDebounceButtonTime* declared as shown in Table 4.3, 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 *lm35ReadingsArrayInit()* was introduced and discussed in the proposed exercise of Example 3.5 and therefore is not discussed here again. The functions *debounceButtonInit()* and *debounceButtonUpdate()* will be used to implement the debounce.

The setup of the pull up on *enterButton* (see Figure 4.5) and a call to the function *debounceButtonInit()* are included in the function *inputsInit()*. Finally, *outputRow4* is set to LOW in *outputsInit()*.

Table 4.3. *Sections in which lines were added to Example 3.5.*

Section	Lines that were added
Defines	`#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 objects	`DigitalOut outputRow4(PA_15);`

Declaration and initialization of public global variables	`int accumulatedDebounceButtonTime = 0;`
	`int numberOfEnterButtonReleasedEvents = 0;`
	`buttonState_t enterButtonState;`

Declarations (prototypes) of public functions	`void lm35ReadingsArrayInit();`
	`void debounceButtonInit();`
	`bool debounceButtonUpdate();`

`void inputsInit()`	`enterButton.mode(PullUp);` `debounceButtonInit();`
`void outputsInit()`	`outputRow4 = LOW;`

Table 4.4 shows the lines that were removed from Example 3.5. The variable *buttonsPressed* was removed because no code is entered using the buttons in this example. For the same reason the objects *aButton*, *bButton*, *cButton* and *dButton* were removed. Their corresponding pull down settings were removed from *inputsInit()*, as well as pull down settings on *alarmTestButton*, because the B1 USER button is already connected to an external pull-down resistor placed in the NUCLEO board.

Given that in this example no code will be entered using the buttons nor the keypad, the function *areEqual()* was removed.

Table 4.4. *Sections in which lines were removed from Example 3.5.*

Section	Lines that were removed
Declaration and initialization of public global variables	`int buttonsPressed[NUMBER_OF_KEYS] = { 0, 0, 0, 0 };`

Declaration and initialization of public global objects	`DigitalIn aButton(D4);`
	`DigitalIn bButton(D5);`
	`DigitalIn cButton(D6);`
	`DigitalIn dButton(D7);`

Declarations (prototypes) of public functions	`areEqual()`

Implementations of public functions	`bool areEqual()`
	`{`
	`  int i;`
	
	`  for (i = 0; i < NUMBER_OF_KEYS; i++) {`
	`  if (codeSequence[i] !=`
	`  buttonsPressed[i]) {`
	`  return false;`
	`  }`
	`  }`
	
	`  return true;`
	`}`

`void inputsInit()`	`alarmTestButton.mode(PullDown);`
	`aButton.mode(PullDown);`
	`bButton.mode(PullDown);`
	`cButton.mode(PullDown);`
	`dButton.mode(PullDown);`

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 not being pressed (line 3), *enterButtonState* is set to BUTTON_UP (line 4). If it is pressed, *enterButtonState* is set to BUTTON_DOWN (line 6).

```
1 void debounceButtonInit()
2 {
3     if( enterButton == 1) {
4         enterButtonState = BUTTON_UP;
5     } else {
6         enterButtonState = BUTTON_DOWN;
7     }
8 }
```

Code 4.1. *Details of the function debounceButtonInit().*

In Code 4.2, the new implementation of the function *alarmActivationUpdate()* is shown. On line 3 it is assessed if *numberOfIncorrectCodes* is less than 5, because alarm deactivation should be available only if less than five incorrect codes were entered. Despite that no codes are entered in this example by means of the keypad, incorrect codes might be entered by means of the command “4” using the PC keyboard.

On line 4 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, the variable *numberOfEnterButtonReleasedEvents* is increased by one. Line 8 evaluates whether there were two or more enter button released events, and if so *numberOfEnterButtonReleasedEvents* is set to zero and *alarmState* is set to OFF.. This implements the double-press functionality for the Enter button. On line 10 *incorrectCodeLed* is set to OFF (remember that an incorrect code can be entered using the PC), and *numberOfIncorrectCodes* is assigned zero to reset the count.

Finally, on line 14 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              numberOfEnterButtonReleasedEvents++;
7              if( numberOfEnterButtonReleasedEvents >= 2 ) {
8                  numberOfEnterButtonReleasedEvents = 0;
9                  alarmState = OFF;
10                 incorrectCodeLed = OFF;
11                 numberOfIncorrectCodes = 0;
12             }
13         }
14     } else {
15         systemBlockedLed = ON;
16     }
17 }
```

Code 4.2. *Modifications introduced in the function alarmDeactivationUpdate().*

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 as a double-press.

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 false, 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 true, 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 == 1* 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 == 0), if (enterButton == false), if (!enterButton).

```
 1 bool debounceButtonUpdate()
 2 {
 3     bool enterButtonReleasedEvent = false;
 4     switch( enterButtonState ) {
 5
 6     case BUTTON_UP:
 7         if( enterButton == 0 ) {
 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 == 0 ) {
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 == 1 ) {
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 == 1 ) {
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. Recall that in Figure 4.8, 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 exercises

1) How can the code be modified in order to properly debounce a button with a bouncing time of about 400 ms?

Answers to the exercises

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 be used to enter the code to deactivate the alarm.

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 (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. 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. The corresponding diagram is shown in Figure 4.10.

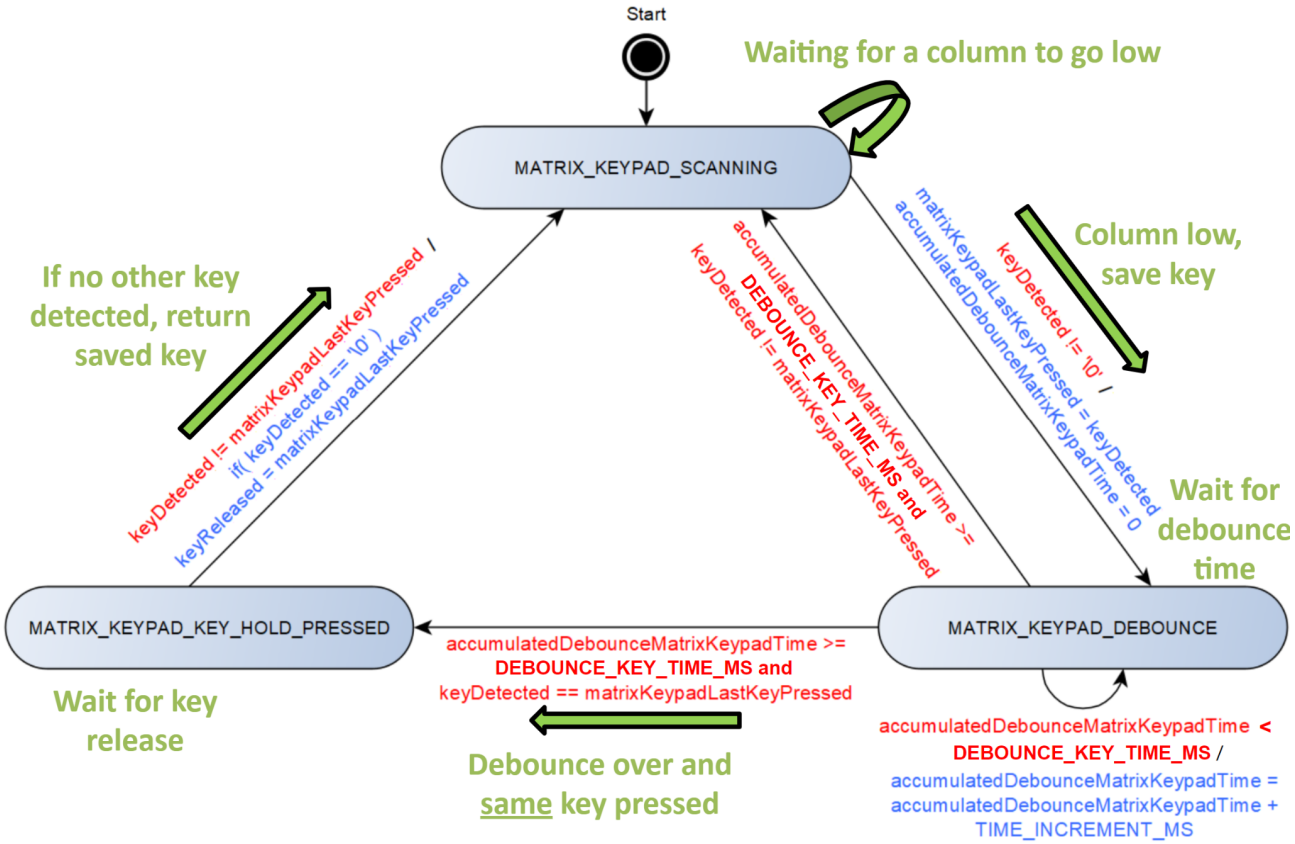

Figure 4.10. Diagram of the finite-state machine implemented in Example 4.2. Some comments added in green.

The first state is MATRIX_KEYPAD_SCANNING. If a key is pressed (*keyDetected* != ‘\0’), the corresponding reading is assigned to *matrixKeypadLastKeyPressed*, *accumulatedDebounceMatrixKeypadTime* set to 0, and the new state of the FSM is MATRIX_KEYPAD_DEBOUNCE. The FSM remains in that state until *accumulatedDebounceMatrixKeypadTime* reaches DEBOUNCE_BUTTON_TIME_MS. Then, if *keyDetected* == *matrixKeypadLastKeyrPressed*, the transition is to MATRIX_KEYPAD_KEY_HOLD_PRESSED.

Otherwise, the transition is to MATRIX_KEYPAD_SCANING. In MATRIX_KEYPAD_KEY_HOLD_PRESSED the FSM remains until *keyDetected* != *matrixKeypadLastKeyrPressed*, then if *keyDetected* = '\0, then *keyReleased* = *matrixKeypadLastKeyPressed*

Implementation of the proposed solution

In Table 4.5 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. Also, a variable of the user-defined type *matrixKeypadState_t* is declared as *matrixKeypadState*. Finally, an array of integer *keysPressed* is declared to store the keys that were pressed, and a integer variable *numberOfHashKeyReleasedEvents* is declared to count for the "#" that were pressed.

In the section "Declarations (prototypes) of public functions," four new functions are declared: *matrixKeypadInit()*, *matrixKeypadScan()*, *matrixKeypadUpdate()* and *areEqual()*. These functions are explained in the example.

Finally, Table 4.5 shows that a call to the function *matrixKeypadInit()* is included into the function *inputsInit()*.

Table 4.5. *Sections and functions in which lines were added to Example 4.1*.

Section or function	Lines that were added
Defines	`#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;`
	`int keysPressed[NUMBER_OF_KEYS] = { 0, 0, 0, 0 };`
	`int numberOfHashKeyReleasedEvents = 0;`

Declarations (prototypes) of public functions	`void matrixKeypadInit();`
	`char matrixKeypadScan();`
	`char matrixKeypadUpdate();`
	`bool areEqual();`

Function inputsInit()	`matrixKeypadInit();`

In Table 4.6 it is shown that DEBOUNCE_BUTTON_TIME_MS was renamed to DEBOUNCE_KEY_TIME_MS. Finally, all the code that was introduced in Example 4.1 to read the Enter button is now deleted, because the "#" key will be read using the functions *matrixKeypadScan()* and *matrixKeypadUpdate()* that are discussed below. The lines that were removed are summarized in Table 4.7.

Table 4.6. *Declaration of public global objects that were modified from Example 4.1.*

Declaration in Example 4.1	Declaration in Example 4.2
`#define DEBOUNCE_BUTTON_TIME_MS 40`	`#define DEBOUNCE_KEY_TIME_MS 40`

Table 4.7. *Sections in which lines were removed from Example 4.2*

Section	Lines that were removed

Declaration and initialization of public global variables	`int accumulatedDebounceButtonTime = 0;`
	`int numberOfEnterButtonReleasedEvents = 0;`
	`buttonState_t enterButtonState;`

Declaration and initialization of public global objects	`DigitalIn enterButton(PB_15);` `DigitalOut outputRow4(PA_15);`

Declaration of public data types	`typedef enum {`
	`  BUTTON_UP,`
	`  BUTTON_DOWN,`
	`  BUTTON_FALLING,`
	`  BUTTON_RISING`
	`} buttonState_t;`

Declarations (prototypes) of public functions	`void debounceButtonInit()` `bool debounceButtonUpdate()`
Implementations of public functions	`void debounceButtonInit()` `bool debounceButtonUpdate()`
`void inputsInit()`	`enterButton.mode(PullUp);` `outputRow4 = LOW;`

In Code 4.4 the implementation of *areEqual()* is shown. It is almost the same as the function *areEqual()* discussed in Chapter 2, the only difference being that now *keysPressed* is used in the comparison.

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

Code 4.4. *Details of the function areEqual().*

In Code 4.5, the new implementation of the function *alarmDeactivationUpdate()* is shown. The changes begin on line 4, where a variable called *keyReleased* is defined and assigned the returned value of the function *matrixKeypadUpdate()*. On line 5 there is a switch over *keyReleased*; if *keyReleased* is equal to \0 (line 6), it implies that no key was pressed, and the break on line 7 is executed. if the key pressed is equal to "A", "B", "C", or "D", then the corresponding position of the array *buttonsPressed* is set to 1 (lines 8 to 19).

On line 20 it is determined if the "#" key was pressed on the matrix keypad. If so, it is assessed if *incorrectCodeLed* is on. In that case *numberOfHashKeyReleasedEvents* is increased by 1. If there were two or more hash key released events (line 23), then *incorrectCodeLed* is set to OFF and *numberOfHashKeyReleasedEvents* is set to zero. Also from lines 26 to 29 all of the positions of *keysPressed* are assigned 0.

Note: If assignments from lines 26 to 29 were not done then it may happen that after entering an incorrect code, for example only the “A” key, followed by another incorrect code, for example only the key “B”, then in the next attempt if for example only the key “#” were pressed the code will be considered correct.

If *incorrectCodeLed* is off, then it is assessed if *alarmState* is true. If so, the function areEqual() is used to evaluate *keysPressed* and *codeSequence* are equal. In that case *alarmState* is set to OFF and *numberOfIncorrectCodes* is set to zero. Also,all the positions of *keysPressed* are assigned 0 (lines 36 to 39).

Note: If assignments from lines 36 to 39 were not done it may happen that after entering a correct code and turning off the Alarm, then if the Alarm turns on again it can be turned off by pressing only “#” key.

If the entered code is not correct, *incorrectCodeLed* is turned on (line 41) and *numberOfIncorrectCodes* is increased by one (line 42). Finally, if *numberOfIncorrectCodes* is 5 or more, *systemBlockedLed* is turned on (line 48).

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 *keysPressed* array. For example, if “A” is pressed, a “1” is stored in *keysPressed*[0], and if “B” is pressed, a “1” is stored in *keysPressed*[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.

```
void alarmDeactivationUpdate()
{
    if ( numberOfIncorrectCodes < 5 ) {
        char keyReleased = matrixKeypadUpdate();
        switch (keyReleased) {
        case '\0':
            break;
        case 'A':
            keysPressed[0] = 1;
            break;
        case 'B':
            keysPressed[1] = 1;
            break;
        case 'C':
            keysPressed[2] = 1;
            break;
        case 'D':
            keysPressed[3] = 1;
            break;
        case '#':
            if( incorrectCodeLed ) {
                numberOfHashKeyReleasedEvents++;
                if( numberOfHashKeyReleasedEvents >= 2 ) {
                    incorrectCodeLed = OFF;
                    numberOfHashKeyReleasedEvents = 0;
                    keysPressed[0] = 0;
                    keysPressed[1] = 0;
                    keysPressed[2] = 0;
                    keysPressed[3] = 0;
                }
            } else {
                if ( alarmState ) {
                    if ( areEqual() ) {
                        alarmState = OFF;
                        numberOfIncorrectCodes = 0;
                        keysPressed[0] = 0;
                        keysPressed[1] = 0;
                        keysPressed[2] = 0;
                        keysPressed[3] = 0;
                    } else {
                        incorrectCodeLed = ON;
                        numberOfIncorrectCodes++;
                    }
                }
            }
        }
    } else {
        systemBlockedLed = ON;
    }
}
```

Code 4.5. *Details of the function alarmDeactivationUpdate().*

Code 4.6 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 4.6. *Details of the function matrixKeypadInit().*

The implementation of the function *matrixKeypadScan()* is shown in Code 4.7. 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 (corresponding to R1-R4 in Figure 4.4) 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 4.7. 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 4.7. *Details of the function matrixKeypadScan().*

Code 4.8 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 4.8. *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 4.5 as described previously.

Proposed exercises

1) What should be adapted in the code if a keypad having five rows and five columns is to be used?

Answers to the exercises

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 in 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 4.8 shows the variable *matrixKeypadCodeIndex* that is declared in this example. The variable *matrixKeypadCodeIndex* will be used to keep track of the buttons that are pressed on the matrix keypad.

In Table 4.9, 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’.

Note: *codeSequence* and *keysPressed* 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 4.8. *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;`

Table 4.9. *Defines, variables names and variable initializations that were modified from Example 4.2.*

Declaration in Example 4.2	Declaration in Example 4.3
`int buttonBeingCompared = 0;`	`int keyBeingCompared = 0;`
`int codeSequence[NUMBER_OF_KEYS] = { 1, 1, 0, 0 };`	`char codeSequence[NUMBER_OF_KEYS] = { '1', '8', '0', '5' };`
`int buttonsPressed[NUMBER_OF_KEYS] = { 0, 0, 0, 0 };`	`char keyPressed[NUMBER_OF_KEYS] = { '0', '0', '0', '0' }`

Code 4.9 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 five. 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 *keysPressed* 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 4.9 are identical to the corresponding lines of Code 4.2 except for the addition of lines 19 and 30 that assigns 0 to *matrixKeypadCodeIndex*.

```
 1 void alarmDeactivationUpdate()
 2 {
 3     if ( numberOfIncorrectCodes < 5 ) {
 4         char keyReleased = matrixKeypadUpdate();
 5         if( keyReleased != '\0' && keyReleased != '#' ) {
 6             keysPressed[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                     keysPressed[0] = '0';
21                     keysPressed[1] = '0';
22                     keysPressed[2] = '0';
23                     keysPressed[3] = '0';
24                 }
25             } else {
26                 if ( alarmState ) {
27                     if ( areEqual() ) {
28                         alarmState = OFF;
29                         numberOfIncorrectCodes = 0;
30                         matrixKeypadCodeIndex = 0;
31                         keysPressed[0] = '0';
32                         keysPressed[1] = '0';
33                         keysPressed[2] = '0';
34                         keysPressed[3] = '0';
35                     } else {
36                         incorrectCodeLed = ON;
37                         numberOfIncorrectCodes++;
38                     }
39                 }
40             }
41         }
42     } else {
43         systemBlockedLed = ON;
44     }
45 }
```

Code 4.9. *Details of the function alarmDeactivationUpdate().*

In Code 4.10, 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 12; *incorrectCode* is set to true (line 13) if one of the keys does not match the code sequence. Line 11 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 39 is used to get the keys and store them in *codeSequence*. Line 41 is used to inform the user that the new code has been configured.

```
case '4':
    uartUsb.write( "Please enter the four digits numeric code ", 42 );
    uartUsb.write( "to deactivate the alarm: ", 25 );

    incorrectCode = false;

    for ( keyBeingCompared = 0;
          keyBeingCompared < NUMBER_OF_KEYS;
          keyBeingCompared++) {
        uartUsb.read( &receivedChar, 1 );
        uartUsb.write( "*", 1 );
        if ( codeSequence[keyBeingCompared] != receivedChar ) {
            incorrectCode = true;
        }
    }

    if ( incorrectCode == false ) {
        uartUsb.write( "\r\nThe code is correct\r\n\r\n", 25 );
        alarmState = OFF;
        incorrectCodeLed = OFF;
        numberOfIncorrectCodes = 0;
    } else {
        uartUsb.write( "\r\nThe code is incorrect\r\n\r\n", 27 );
        incorrectCodeLed = ON;
        numberOfIncorrectCodes++;
    }
    break;

case '5':
    uartUsb.write( "Please enter the new four digits numeric code ", 46 );
    uartUsb.write( "to deactivate the alarm: ", 25 );

    for ( keyBeingCompared = 0;
          keyBeingCompared < NUMBER_OF_KEYS;
          keyBeingCompared++) {
        uartUsb.read( &receivedChar, 1 );
        uartUsb.write( "*", 1 );
        codeSequence[keyBeingConfigured] = receivedChar;
    }

    uartUsb.write( "\r\nNew code generated\r\n\r\n", 24 );
    break;
```

Code 4.10. *Lines that were modified in the function uartTask().*

Proposed exercises

1) How can the code be modified in order to use a three-digit code?

Answers to the exercises

1) In the arrays *codeSequence* and *keysPressed* 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 100 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 in 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 4.10 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 100, 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 4.10 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 4.10. *Sections in which lines were added to Example 4.3.*

Section	Lines that were added
Defines	`#define EVENT_MAX_STORAGE 100` `#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 4.11, 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 4.11. 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 4.11. *Details of the struct tm.*

Member	Type	Meaning	Range
tm_sec	int	seconds after the minute	0-59
tm_min	int	minutes after the hour	0-59
tm_hour	int	hours since midnight	0-23
tm_mday	int	day of the month	1-31
tm_mon	int	months since January	0-11
tm_year	int	years since 1900	
tm_wday	int	days since Sunday	0-6
tm_yday	int	days since January 1	0-365
tm_isdst	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 4.11 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).

```
case 's':
case 'S':
    struct tm rtcTime;
    int strIndex;

    uartUsb.write( "\r\nType four digits for the current year (YYYY): ", 48 );
    for( strIndex=0; strIndex<4; strIndex++ ) {
        uartUsb.read( &str[strIndex] , 1 );
        uartUsb.write( &str[strIndex] ,1 );
    }
    str[4] = '\0';
    rtcTime.tm_year = atoi(str) - 1900;
    uartUsb.write( "\r\n", 2 );

    uartUsb.write( "Type two digits for the current month (01-12): ", 47 );
    for( strIndex=0; strIndex<2; strIndex++ ) {
        uartUsb.read( &str[strIndex] , 1 );
        uartUsb.write( &str[strIndex] ,1 );
    }
    str[2] = '\0';
    rtcTime.tm_mon  = atoi(str) - 1;
    uartUsb.write( "\r\n", 2 );

    uartUsb.write( "Type two digits for the current day (01-31): ", 45 );
    for( strIndex=0; strIndex<2; strIndex++ ) {
        uartUsb.read( &str[strIndex] , 1 );
        uartUsb.write( &str[strIndex] ,1 );
    }
    str[2] = '\0';
    rtcTime.tm_mday = atoi(str);
    uartUsb.write( "\r\n", 2 );

    uartUsb.write( "Type two digits for the current hour (00-23): ", 46 );
    for( strIndex=0; strIndex<2; strIndex++ ) {
        uartUsb.read( &str[strIndex] , 1 );
        uartUsb.write( &str[strIndex] ,1 );
    }
    str[2] = '\0';
    rtcTime.tm_hour = atoi(str);
    uartUsb.write( "\r\n", 2 );

    uartUsb.write( "Type two digits for the current minutes (00-59): ", 49 );
    for( strIndex=0; strIndex<2; strIndex++ ) {
        uartUsb.read( &str[strIndex] , 1 );
        uartUsb.write( &str[strIndex] ,1 );
    }
    str[2] = '\0';
    rtcTime.tm_min  = atoi(str);
    uartUsb.write( "\r\n", 2 );

    uartUsb.write( "Type two digits for the current seconds (00-59): ", 49 );
    for( strIndex=0; strIndex<2; strIndex++ ) {
        uartUsb.read( &str[strIndex] , 1 );
        uartUsb.write( &str[strIndex] ,1 );
    }
    str[2] = '\0';
    rtcTime.tm_sec  = atoi(str);
    uartUsb.write( "\r\n", 2 );

    rtcTime.tm_isdst = -1;
    set_time( mktime( &rtcTime ) );
    uartUsb.write( "Date and time has been set\r\n", 28 );

    break;
```

Code 4.11. *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 4.12. 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 4.12.

```
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 4.12. *Lines that were included in the function uartTask() (part 2/2).*

Code 4.13 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 4.13. *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 4.14. 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 4.14. *New implementation of the function main().*

In Code 4.15, 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 4.15. *Implementation of the function eventLogUpdate().*

The implementation of the function s*ystemElementStateUpdate()* is shown in Code 4.16. 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 4.16. *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 4.11. 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 4.16. 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 exercises

1) How can a change be implemented in the code in order to allow up to 1000 events to be stored?

Answers to the exercises

1) The value of MAX_NUMBER_OF_EVENTS should be changed to 1000.

4.3 Under the hood

4.3.1 Software debugging

In computer programming and software development, debugging is the process of finding and resolving defects or problems that prevent correct operation within computer programs, software, or systems. In 1878 Thomas Edison described the "little faults and difficulties" of mechanical engineering as "Bugs" and in the 1940s Grace Hopper at Harvard University used the term for the first in the field of computer engineering when she and her associates discovered a moth stuck in a relay and thereby impeding operation. Keil Studio Cloud has debugging capabilities that are described in this section.

Warning: The debugger features of the Keil Studio Cloud are somewhat new and require the ST-LINK in-circuit debugger and programmer that is loaded in the NUCLEO Board to be version V2J36 or later. To check the version, connect the NUCLEO board and open the “DETAILS” file in the drive assigned to the NUCLEO board (for example, D:\, named NODE_F429ZI). If an update is necessary, follow the steps indicated in [7].

A detailed and updated step by step tutorial on how to use debugging on Keil Studio Cloud is available in [8]. The debugger mode provides different ways of checking the code while it runs. It is possible to step through the code, examine the execution path, look at the values stored in variables, and more, as indicated on Table 4.12.

Table 4.12. *Main debugging resources available on Keil Studio Cloud.*

Type	Description	Comments
Step buttons	Allows to navigate the code	Available buttons are (see more details in [8]): - Step over: Advance the debugger to the next source line that follows in the program. - Step into: Advance the debugger into each function. - Step out: Advance the debugger until the current function returns.
Breakpoints	Stops code execution at a given code line	Breakpoints are useful to look at values of variables, check if a block of code is getting executed, and to suspend running code.
Inspect variables	Allows to check values stored by variables	Local and global variables, and function arguments are shown in the Variables list. When the debugger is paused the corresponding current values can be seen.
Inspect registers	Allows to check values stored by registers	Registers are special locations of the microcontroller internal memory used by the processor. In the Under the Hood section of Chapter 1 some insight of the processor's internal organization was given. This topic is not covered in detail in this book.

Proposed exercises

1) How can Example 4.4 be debugged using Keil Studio Cloud?

Answers to the exercises

1) Make Example 4.4 the active project, and open the main.cpp file. Click the bug symbol to start the debugging tool. For more details about how to do it look at the step by step tutorial available in [8].

4.4 Case study

4.4.1 Smart door locks

In chapter 1, the case study of the smart door lock was introduced [9]. 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.

Figure 4.11. Smart door lock built with Mbed contains a keypad similar to the one 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.

Proposed exercises

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.

Answers to the exercises

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 in 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 4.17, 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 4.17. 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 4.18, 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 4.18, 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 4.17. *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 4.18. *Declaration of the public data types doorState_t and matrixKeypadState_t*

The section “Declaration and initialization of public global objects” is shown in Code 4.19. 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 4.19, 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 intitalization 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 4.19. *Declaration of the public global objects.*

In Code 4.20, 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 intitalization 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 4.20. *Declaration and initialization of public global variables.*

In Code 4.21, 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 4.21. *Declaration of public functions used in the proposed solution.*

The implementation of the *main()* function is shown in Code 4.22. 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 4.22. *Declaration of public functions used in the proposed solution.*

In Code 4.23, 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. BBy means of comparing lines 19 to 52 with Code 4.11, 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.

```
void uartTask()
{
    char str[100] = "";
    char receivedChar = '\0';
    struct tm rtcTime;
    char year[5] = "";
    char month[3] = "";
    char day[3] = "";
    char hour[3] = "";
    char minute[3] = "";
    char second[3] = "";
    time_t epochSeconds;
    receivedChar = pcSerialComCharRead();
    if( receivedChar !=  '\0') {
        switch (receivedChar) {

        case 's':
        case 'S':
            pcSerialComStringWrite("\r\nType four digits for the current year (YYYY): ");
            pcSerialComStringRead( year, 4);
            pcSerialComStringWrite("\r\n");
            rtcTime.tm_year = atoi(year) - 1900;

            pcSerialComStringWrite("Type two digits for the current month (01-12): ");
            pcSerialComStringRead( month, 2);
            pcSerialComStringWrite("\r\n");
            rtcTime.tm_mon  = atoi(month) - 1;

            pcSerialComStringWrite("Type two digits for the current day (01-31): ");
            pcSerialComStringRead( day, 2);
            pcSerialComStringWrite("\r\n");
            rtcTime.tm_hour = atoi(hour);

            pcSerialComStringWrite("Type two digits for the current hour (00-23): ");
            pcSerialComStringRead( hour, 2);
            pcSerialComStringWrite("\r\n");
            rtcTime.tm_hour = atoi(hour);

            pcSerialComStringWrite("Type two digits for the current minutes (00-59): ");
            pcSerialComStringRead( minute, 2);
            pcSerialComStringWrite("\r\n");
            rtcTime.tm_min  = atoi(minute);

            pcSerialComStringWrite("Type two digits for the current seconds (00-59): ");
            pcSerialComStringRead( second, 2);
            pcSerialComStringWrite("\r\n");
            rtcTime.tm_sec  = atoi(second);

            rtcTime.tm_isdst = -1;
            set_time( mktime( &rtcTime ) );
            pcSerialComStringWrite("Date and time has been set\r\n");
            break;

        case 't':
        case 'T':
            epochSeconds = time(NULL);
            sprintf ( str, "Date and Time = %s", ctime(&epochSeconds));
            pcSerialComStringWrite( str );
            pcSerialComStringWrite("\r\n");
            break;

        default:
            availableCommands();
            break;
        }
    }
}
```

Code 4.23. *Implementation of the function uartTask().*

The implementation of the function *availableCommands()* is shown in Code 4.24. 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 4.25, 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 4.26 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 safer to always define a default case).

In Code 4.27, Code 4.28 and Code 4.29, the implementation of the functions *matrixKeypadInit()*, *matrixKeypadScan()*, and *matrixKeypadUpdate()* are shown. It can be seen that this is the same code as in Code 4.7. 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 4.24. *Implementation of the function availableCommands().*

```
1 void doorInit()
2 {
3     doorUnlockedLed = OFF;
4     doorLockedLed = ON;
5     incorrectCodeLed = OFF;
6     doorState = DOOR_CLOSED;
7 }
```

Code 4.25. *Implementation of the function doorInit().*

```
void doorUpdate()
{
    bool incorrectCode;
    char keyReleased;
    struct tm * currentTime;
    char prevkeyReleased;
    int i;

    switch( doorState ) {
    case DOOR_CLOSED:
        keyReleased = matrixKeypadUpdate();
        if ( keyReleased == 'A' ) {
            seconds = time(NULL);
            currentTime = localtime ( &seconds );

            if ( ( currentTime->tm_hour >= START_HOUR ) &&
                 ( currentTime->tm_hour <= END_HOUR ) ) {
                incorrectCode = false;
                prevkeyReleased = 'A';

                for ( i = 0; i < CODE_DIGITS; i++) {
                    while ( ( keyReleased == '\0' ) ||
                            ( keyReleased == prevkeyReleased ) ) {

                        keyReleased = matrixKeypadUpdate();
                    }
                    prevkeyReleased = keyReleased;
                    if ( keyReleased != codeSequence[i] ) {
                        incorrectCode = true;
                    }
                }

                if ( incorrectCode ) {
                    incorrectCodeLed = ON;
                    delay (1000);
                    incorrectCodeLed = OFF;
                } else {
                    doorState = DOOR_UNLOCKED;
                    doorLockedLed = OFF;
                    doorUnlockedLed = ON;
                }
            }
        }
        break;

    case DOOR_UNLOCKED:
        if ( doorHandle ) {
            doorUnlockedLed = OFF;
            doorState = DOOR_OPEN;
        }
        break;

    case DOOR_OPEN:
        if ( !doorHandle ) {
            doorLockedLed = ON;
            doorState = DOOR_CLOSED;
        }
        break;

    default:
        doorInit();
        break;
    }
}
```

Code 4.26. *Implementation of the function doorUpdate().*

```
void matrixKeypadInit()
{
    matrixKeypadState = MATRIX_KEYPAD_SCANNING;
    int pinIndex = 0;
    for( pinIndex=0; pinIndex<KEYPAD_NUMBER_OF_COLS; pinIndex++ ) {
        (keypadColPins[pinIndex]).mode(PullUp);
    }
}
```

Code 4.27. *Implementation of the function keypadInit().*

```
char matrixKeypadScan()
{
    int r = 0;
    int c = 0;
    int i = 0;

    for( r=0; r<KEYPAD_NUMBER_OF_ROWS; r++ ) {

        for( i=0; i<KEYPAD_NUMBER_OF_ROWS; i++ ) {
            keypadRowPins[i] = ON;
        }

        keypadRowPins[r] = OFF;

        for( c=0; c<KEYPAD_NUMBER_OF_COLS; c++ ) {
            if( keypadColPins[c] == OFF ) {
                return matrixKeypadIndexToCharArray[r*KEYPAD_NUMBER_OF_ROWS + c];
            }
        }
    }
    return '\0';
}
```

Code 4.28. *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         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 4.29. Implementation of the function *matrixKeypadUpdate()*.

Finally, in Code 4.30 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.

```
void pcSerialComStringWrite( const char* str )
{
    uartUsb.write( str, strlen(str) );
}

char pcSerialComCharRead()
{
    char receivedChar = '\0';
    if( uartUsb.readable() ) {
        uartUsb.read( &receivedChar, 1 );
    }
    return receivedChar;
}

void pcSerialComStringRead( char* str, int strLength )
{
    int strIndex;
    for ( strIndex = 0; strIndex < strLength; strIndex++) {
        uartUsb.read( &str[strIndex] , 1 );
        uartUsb.write( &str[strIndex] ,1 );
    }
    str[strLength]='\0';
}
```

Code 4.30. Implementation of the functions related to the PC serial communication.

References

[1]

“4x4 Keypad Module Pinout, Configuration, Features, Circuit & Datasheet”. Accessed July 9, 2021. https://components101.com/misc/4x4-keypad-module-pinout-configuration-features-datasheet

[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]

“STSW-LINK007 - ST-LINK, ST-LINK/V2, ST-LINK/V2-1, STLINK-V3 boards firmware upgrade - STMicroelectronics”. Accessed February 17, 2023. https://www.st.com/en/development-tools/stsw-link007.html

[8]

“Arm Keil Studio Cloud User Guide”. Accessed February 17, 2023. https://developer.arm.com/documentation/102497/1-5/Monitor-and-debug/Debug-a-project-with-Keil-Studio

[9]

“Smart door locks | Mbed”. Accessed July 9, 2021. https://os.mbed.com/built-with-mbed/smart-door-locks/

Please ignore this, is just to automate some references, in order to make it easier to update them later:

The last example of the previous chapter was:

Example 3.5

The example “Turn off the incorrect code LED by double-clicking the Enter button” is:

Example 4.1

The example “???” is:

Example 4.2

The example “Activate the over temperature alarm by means of the potentiometer” is

Example 4.3

The example “Measure the temperature with the LM35 temperature sensor” is

Example 4.4