# Chapter 7

## DC Motor Driving using Relays and Interrupts

# 7.1 Roadmap

## 7.1.1 What You Will Learn

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

- Summarize the fundamentals of relay modules and use them to control a DC motor.
- Develop programs to get and manage interrupts with the NUCLEO board.
- Summarize how LEDs are connected and used in electronic circuits.
- Describe how to connect a PIR sensor to the NUCLEO board using a digital input.
- Design and generate modifications of existing code to include new functionality.

## 7.1.2 Review of Previous Chapters

In previous chapters, a broad set of modules were connected to the smart home system. In order to deal with all those modules, different functions were called based on a *polling cycle*: at predefined intervals the states of the different elements of the system were checked. In this chapter, a different technique based on hardware interrupts is introduced in order to avoid the overhead of cyclically checking for a given condition. It is shown how to combine the polling cycle technique with the technique based on hardware interrupts.

## 7.1.3 Contents of This Chapter

In this chapter, a direct current (DC) motor will be connected to the smart home system by means of a relay module. The motion of the DC motor will be controlled with a set of buttons. In order to introduce the concept of a *hardware interrupt*, these buttons will not be polled at periodic intervals as in previous chapters. Instead, an *interrupt service routine* will be used to handle the button detection. As part of the implementation, it will be explained how to connect a pair of LEDs to indicate the rotation direction of the DC motor.

To explore the use of interrupts in further detail, a PIR-based motion sensor will be used. The output signal of this sensor will be tracked using interrupts.

Finally, some modifications of the existing code will be made in order to include a new alarm source in the smart home system: the detection of an intruder by means of the PIR sensor. A new alarm message will be shown on the display, and a different configuration for the strobe time of the light and the siren will be defined.

# 7.2 Motion Detection and DC Motor Control using Relays and Interrupts

## 7.2.1 Connect a DC Motor and a PIR Sensor to the Smart Home System

In this chapter, a PIR sensor, a motor, four buttons, and two LEDs are connected to the smart home system in order to implement the behavior shown in Figure 7.1. The PIR sensor is used to detect intruders. In that event, the alarm is activated. The motor is used to move a gate, which is activated by means of two buttons on the Gate control panel labeled “Open” and “Close”, as shown in Figure 7.1. The LEDs (green and red) are used to indicate if the gate is opening or closing.

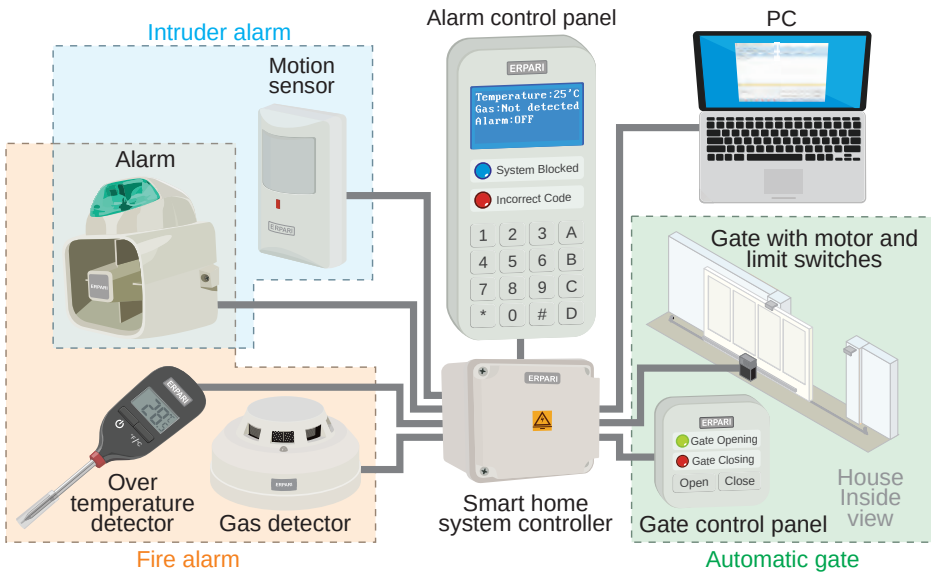



*Figure 7.1 The smart home system is now connected to an LCD display.*

The other two buttons that are connected in this chapter are used to simulate the limit switches that are activated when the gate is completely opened or closed (Figure 7.2). In this way, the motor is stopped when the gate reaches its travel limits.

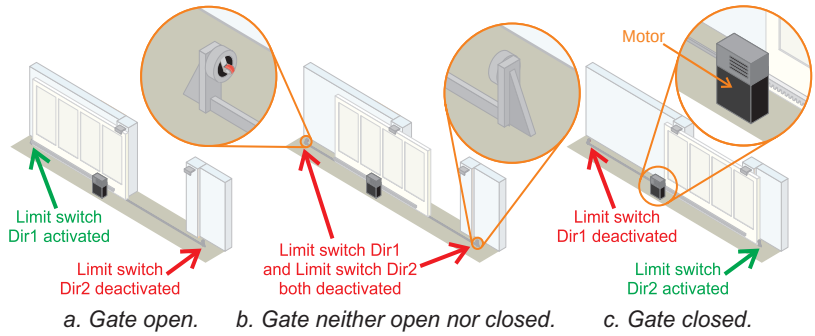



*Figure 7.2 Diagram of the limit switches that are considered in this chapter.*

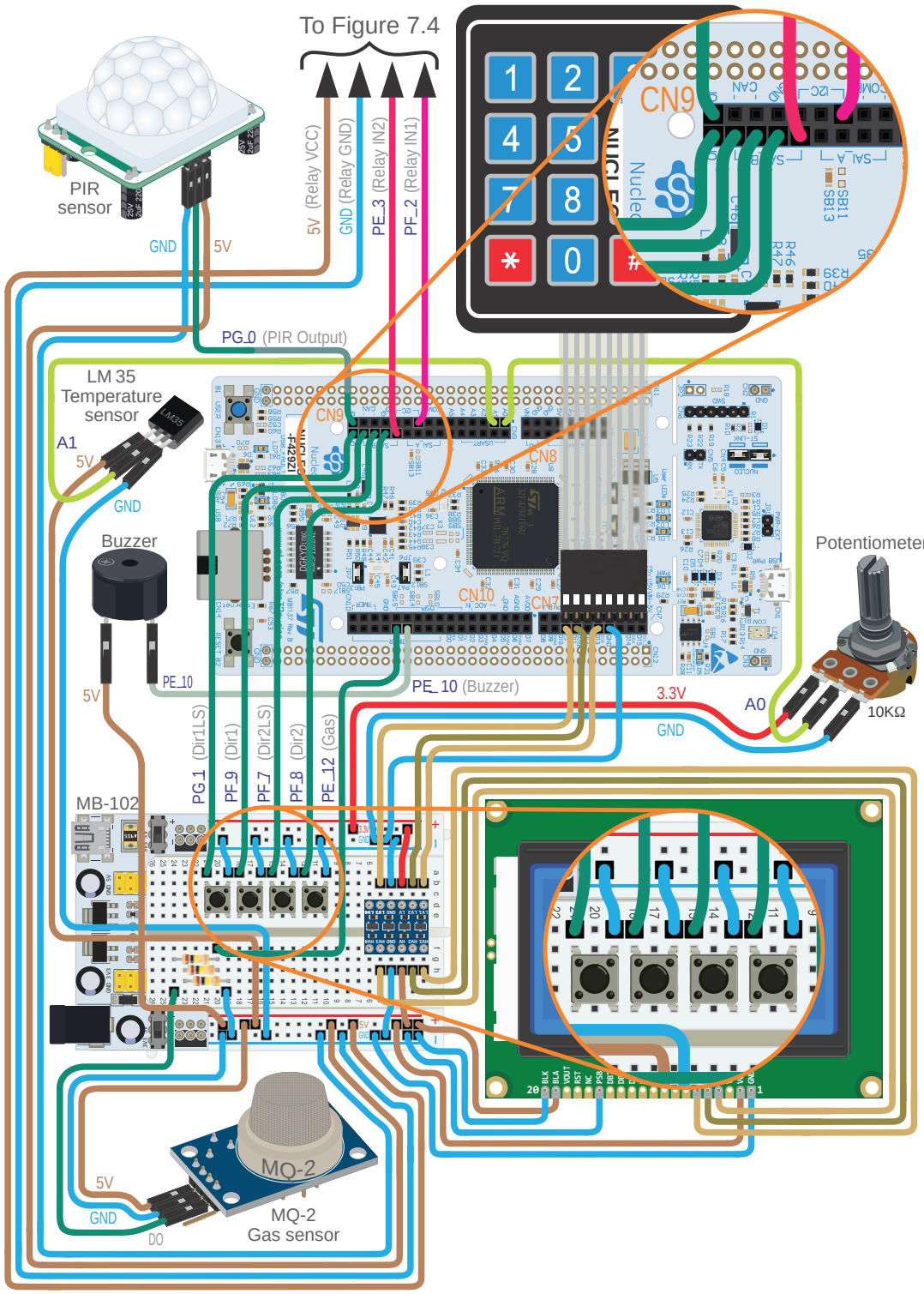



*Figure 7.3 The smart home system is now connected to a PIR sensor and a set of four buttons.*

In this chapter, a 5 V DC motor, similar to the motor shown in [1], and a HC-SR501 PIR sensor, described in [2], are connected to the smart home system, as shown in Figure 7.3 and Figure 7.4. The aim is to introduce the use of *interrupts*.

Figure 7.4 shows that a second MB102 is incorporated in the setup in order to supply the motor with an independent power supply. One of the main reasons for using relay modules in embedded systems is to turn on and off a load (such as an AC or DC motor, or a lamp) by means of a signal that is applied at the input of the relay module by a microcontroller that does not share the same power supply as the load. In this way, microcontrollers are kept safe from high voltages that might be necessary to power the load and are also *isolated* from electrical noise that could be generated when the load is activated.



**WARNING:** The GND pin of the second MB102 (indicated as "GNDmotor" in Figure 7.4) is not connected to the GND of the smart home system. In this way, the power supply of the motor is properly isolated.

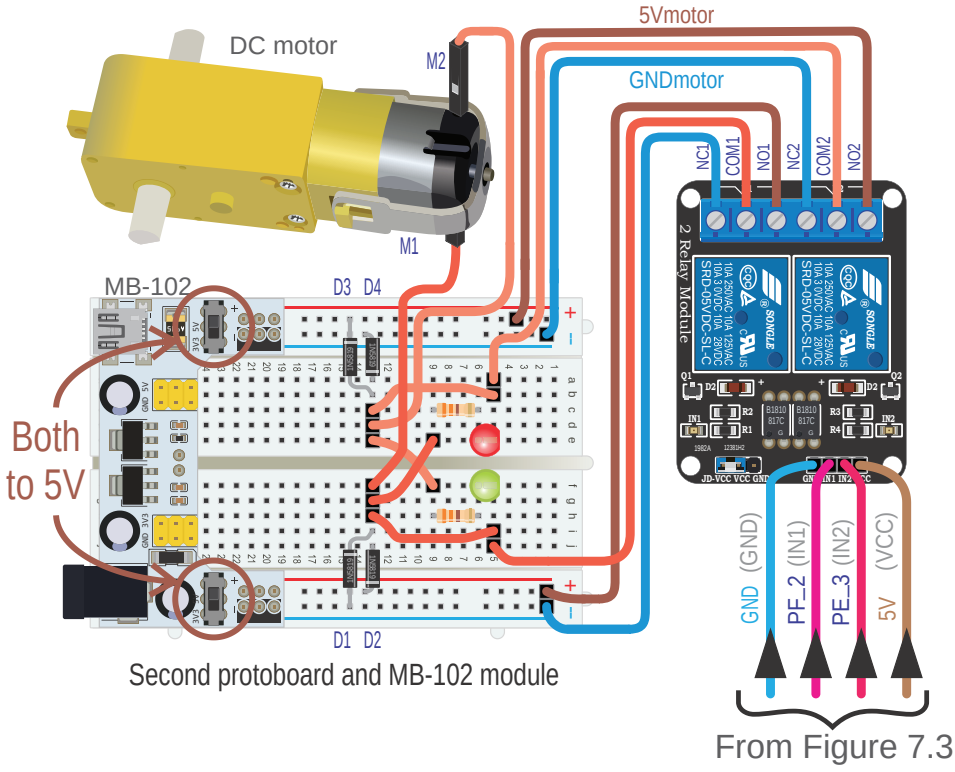



***Figure 7.4 The smart home system is now connected to a 5 V DC motor using a relay module.***

Figure 7.5 shows a conceptual diagram of the circuit that is used to activate the DC motor, LED1, and LED2. The circuit is based on two relays, RL1 and RL2. When IN1 is set to GND, the COM1 terminal of RL1 is connected to NO1 (Normally Open 1). This connects 5Vmotor to the M1 connector of the DC motor. When IN1 is left unconnected, the COM1 terminal of RL1 is connected to NC1 (Normally Closed 1). This connects GNDmotor to the M1 connector of the DC motor. The same behavior is

obtained with RL2 when applying GND to IN2 and when IN2 is left unconnected, respectively. In this way, the DC motor can be activated and its rotation direction controlled, as shown in Table 7.1.

**NOTE:** The diagram shown in Figure 7.5 is presented only to illustrate the operation of the circuit. In the actual circuit, the inputs IN1 and IN2 are not directly connected to the relay; circuitry is used in between. The details of the actual circuit are discussed in the Under the Hood section of this chapter.

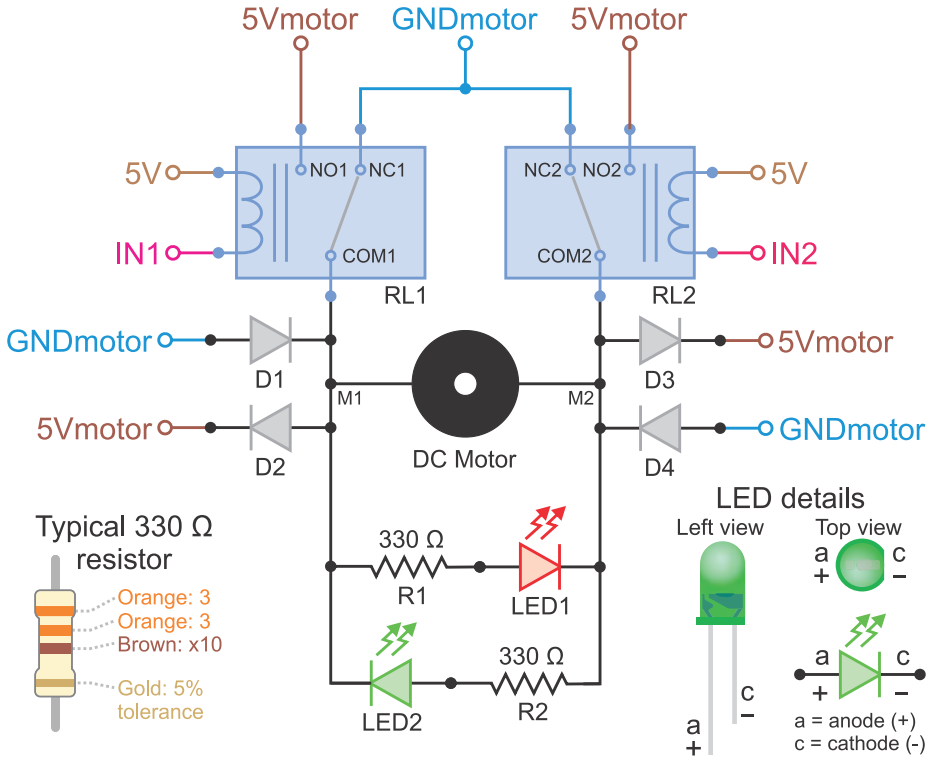



*Figure 7.5 Conceptual diagram of the circuit that is used to activate the DC motor, LED1, and LED2.*

*Table 7.1 Summary of the signals applied to the motor depending on IN1 and IN2, and the resulting behavior.*

| IN1 | M1 | IN2 | M2 | Behavior |
|---|---|---|---|---|
| GND | 5Vmotor | GND | 5Vmotor | The motor does not turn (M2 – M1 = 0 V) |
| Unconnected | GNDmotor | GND | 5Vmotor | The motor turns in one direction (M2 – M1 = +5 V) |
| GND | 5Vmotor | Unconnected | GNDmotor | The motor turns in the other direction (M2 – M1 = −5 V) |
| Unconnected | GNDmotor | Unconnected | GNDmotor | The motor does not turn (M2 – M1 = 0 V) |

The reader might note that by means of IN1 and IN2, the voltage of M1 and M2 is controlled. The aim of the relay is to *isolate* the *input* from the *output*. In this way, just a few micro-amperes are drained by IN1 and IN2, while about 100 milliamperes are provided to the DC motor by means of the connections to GNDmotor and 5Vmotor through the relay.

The diodes D1–D4 are to absorb inductive spikes from the motor inductance. In this way, positive spikes will be conducted to 5Vmotor and negative spikes to GNDmotor. The 1N5819 diode can be used for D1–D4.

This circuit can be used to control powerful DC motors that work with higher voltages and currents by replacing the 5Vmotor voltage supply with an appropriate power supply. Even alternating current (AC) motors can be controlled by means of relay-based circuits, although this topic is beyond the scope of this book.

The connections between the NUCLEO board and the relay module are summarized in Table 7.2, while the connections between the relay module and the breadboard are summarized in Table 7.3.

**WARNING:** In some relay modules, the connections VCC and GND are labeled DC+ and DC-, respectively.

*Table 7.2 Summary of the connections between the NUCLEO board and the relay module.*

| NUCLEO board | Relay module |
|---|---|
| PF_2 | IN1 |
| PE_3 | IN2 |

*Table 7.3 Summary of other connections that should be made to the relay module.*

| Relay module | Element |
|---|---|
| VCC | 5V |
| GND | GND |
| NO1 | 5Vmotor |
| COM1 | M1 |
| NC2 | GNDmotor |
| NO2 | 5Vmotor |
| COM2 | M2 |
| NC2 | GNDmotor |

In Figure 7.5, it can be seen that there are two LEDs, LED1 and LED2, connected in opposite directions (i.e., LED1 *points* from M1 to M2, while LED2 *points* from M2 to M1). These LEDs are to indicate the motor's turning direction. An LED turns on only if the voltage at its anode is higher than the voltage at its cathode. In Figure 7.5, a detailed drawing of an LED is shown, which helps to identify its anode and cathode. Given the connections shown in Figure 7.5, LED1 will turn on when the voltage in M1 is greater than the voltage in M2 ($V_{M1} > V_{M2}$), while LED2 will turn when $V_{M2} > V_{M1}$. The resistors R1 and R2 are used to limit the current across the LED. Figure 7.5 shows how to identify a typical 330 Ω resistor that has a tolerance of 5% of its value.

**WARNING:** Be sure to use 330 Ω resistors and to connect LED1 and LED2 as indicated in Figure 7.5. Otherwise, the LEDs may be damaged and/or not turn on as expected. The tolerance of the resistor and its maximum power dissipation are not relevant. Nor is it relevant whether it is a carbon or metal film resistor.

The *passive infrared sensor* (PIR sensor) works on the basis that all objects emit heat energy in the form of radiation at infrared wavelengths. This radiation is not visible to the human eye but can be detected by electronic devices. A PIR sensor detects changes in the amount of infrared radiation impinging upon it, which varies depending on the temperature and surface characteristics of the objects in front of the sensor.

The term *passive* refers to the fact that PIR devices do not radiate energy for detection purposes but work by detecting infrared radiation (radiant heat) *emitted by* or *reflected from* objects. PIR sensors are commonly used in security alarms and automatic lighting applications.

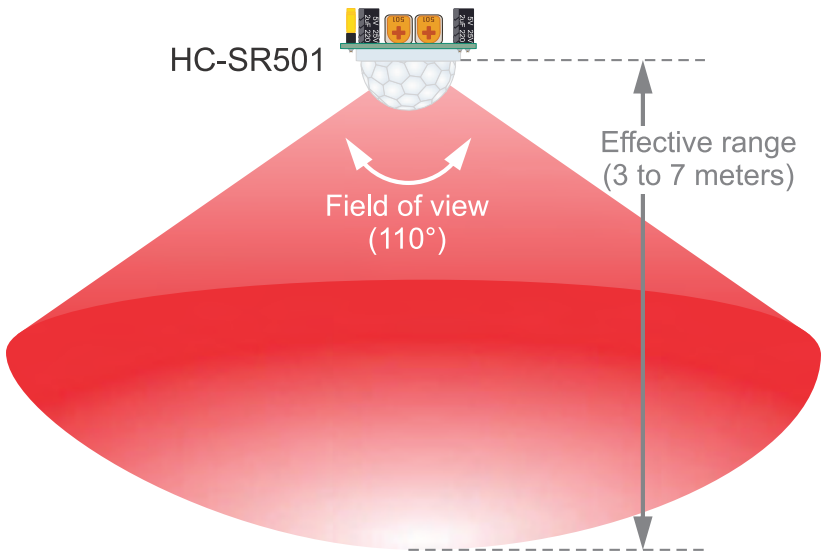



*Figure 7.6 Diagram of the field of view and the effective range of the HC-SR501 PIR sensor.*

For instance, when a person passes in front of a garden, the temperature at that point in the sensor's field of view will rise from the grass temperature to the body temperature. The sensor converts the change in the incoming infrared radiation into a change in its output voltage. The emitted radiation not only depends on the object's temperature but also on its surface characteristics, which can also be used to detect objects.

The most common PIR sensors have an effective range of approximately 10 meters (30 feet) and a field of view of approximately 180°. PIR sensors with a longer effective range and wider fields of view are available, as well as PIRs with very narrow coverage. The HC-SR501 PIR sensor that is used in this chapter has an effective range that is adjustable to between three and seven meters, and a field of view of 110°, as shown in Figure 7.6.

In Figure 7.7, the adjustments and the connection pins of the HC-SR501 PIR sensor are shown. The *sensitivity adjust* potentiometer can be used to set the effective range between three and seven meters. The *time delay adjust* allows configuration of the output signal duration (pulse duration) in the range of three seconds to five minutes. The jumper allows a setting of whether triggering signals are ignored when the output is active (*single trigger*) or are considered (*repeat trigger*). Note that the *repeat trigger* mode must be selected, as shown in Figure 7.7. Some HC-SR501 PIR sensors have this selection made from the factory using bond pads.

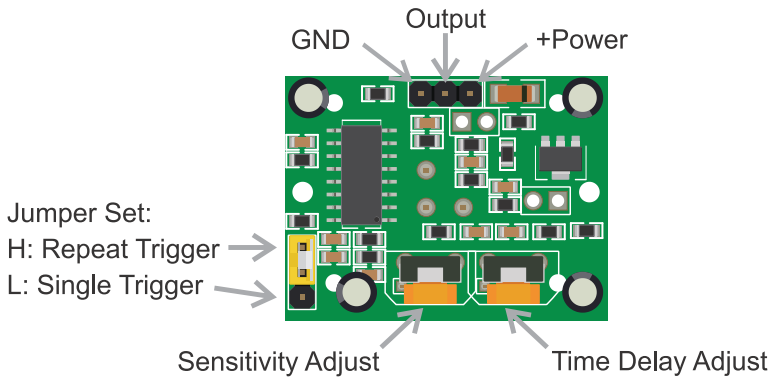



***Figure 7.7 Adjustments and connector of the HC-SR501 PIR sensor.***

The connection between the NUCLEO board and the HC-SR501 PIR sensor is shown in Table 7.4, while the connections between the HC-SR501 PIR sensor and the breadboard are summarized in Table 7.5.

***Table 7.4 Summary of the connections between the NUCLEO board and the HC-SR501 PIR sensor.***

| HC-SR501 PIR sensor | NUCLEO board |
|---|---|
| Output | PG_0 |

***Table 7.5 Summary of connections to the breadboard that should be made on the HC-SR501 PIR sensor.***

| HC-SR501 PIR sensor | Breadboard |
|---|---|
| GND | GND |
| +Power | 5 V |

**WARNING:** It will take up to a minute for the HC-SR501 PIR sensor to stabilize after power-on. Additionally, after the output signal turns inactive, there will be a three-second delay before it can be triggered again.

To test if the HC-SR501 PIR sensor and the motor are working properly, the *.bin* file of the program "Subsection 7.2.1" should be downloaded from the URL available in [3] and loaded onto the NUCLEO board. When the HC-SR501 PIR sensor detects a movement, its output signal will become active and remain active for a time $t_{Delay}$ given by the *time delay adjust* (as shown in Figure 7.7). The motor will

turn in one direction, and one of the LEDs will turn on while the output signal of the HC-SR501 PIR sensor is active. Once the motor stops, if the sensor is activated again, the motor will turn in the other direction and the other LED will turn on. This behavior continues indefinitely.

**WARNING:** Ignore all the other elements of the setup during the proposed test (Alarm LED, display, etc.).

**NOTE:** Given that the repeat trigger option is selected (as indicated in Figure 7.7) during $t_{Delay}$, the sensor can be triggered again by a movement being detected. If that happens, the output signal will be kept active and the counting of $t_{Delay}$ will start again from that point.

**TIP:** This test program can be used to adjust the HC-SR501 *Time Delay Adjust*. It is convenient to select the *Single Trigger* option, as indicated in Figure 7.7, and wave a hand over the HC-SR501 PIR sensor. The motor will turn for a time of $t_{Delay}$. Using a screwdriver, the Time Delay Adjust can be set to make $t_{Delay}$ last for an appropriate time, for example five seconds.

In Table 7.6, the buttons that are connected in Figure 7.3 are summarized. In many applications, such as 3D printers, limit switches are used to detect the end of travel of an object. In Figure 7.8, a typical limit switch is shown. In this chapter, tactile switches are used to represent limit switches.

***Table 7.6 Summary of the buttons that are connected in Figure 7.3.***

| Button name | NUCLEO board |
|---|---|
| *Dir1* (Direction 1) | PF_9 |
| *Dir1LS* (Direction 1 Limit Switch) | PG_1 |
| *Dir2* (Direction 2) | PF_8 |
| *Dir2LS* (Direction 2 Limit Switch) | PF_7 |

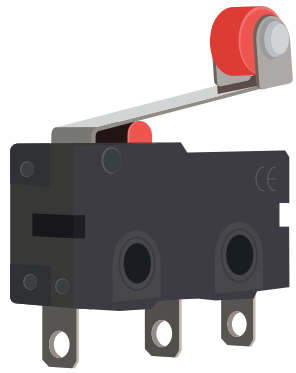


***Figure 7.8 A typical limit switch. Note the connectors on the bottom: common, normally open and normally closed.***

**WARNING:** In order to show that internal pull-up resistors can also be used to connect the buttons (instead of pull-down resistors, as in previous chapters), buttons are connected in a different way than in Chapter 1. Follow the connection diagram shown in Figure 7.3, otherwise the buttons will not work as expected.

**NOTE:** *Dir1LS* and *Dir2LS* are conceived as limit switches used to deactivate the motor when a gate or tool moved by the motor reaches the end of its travel. For this reason, once *Dir1LS* is activated, the motor will not be allowed to move again in Direction 1 until it has first moved in Direction 2.

To test the buttons, press button *Dir1*. The motor should turn in one direction, and one of the LEDs should turn on. Then press button *Dir1LS*. The motor should stop, and the LED should turn off. Next, press button *Dir2*. The motor should turn in the other direction, and the other LED should turn on. Finally, press button *Dir2LS*. The motor should stop, and the LED should turn off.

### 7.2.2 Fundamentals of Interrupt Service Routines

Embedded systems can be configured in order to promptly execute a piece of code when a given condition takes place. This behavior is called an *interrupt*, because the normal flow of the program is altered (i.e., *interrupted*). For example, an electronically controlled lathe must prioritize a halt button related to the safety of the operator over any other functionality. If the halt button is pressed, the electronic controller must alter its normal execution flow in order to execute a given *interrupt service routine* (ISR), as shown in Figure 7.9.

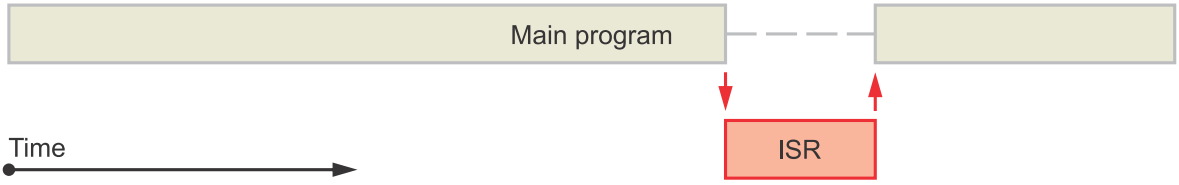



***Figure 7.9 Conceptual diagram of the normal flow of a program altered by an interrupt service routine.***

An interrupt can be caused by an electrical condition. For example, an interrupt is triggered when a signal connected to a microcontroller pin becomes 3.3 V. In this way, a microcontroller may have dozens of interrupt sources that might even be activated simultaneously. For this reason, it must be decided how to proceed when each of the events that can trigger an interrupt occurs. Table 7.7 shows a simplified representation of this concept. It can be seen that some interrupt sources might be active, while others are inactive, and that every active interrupt has an assigned priority and code to be executed when it is triggered.

*Table 7.7 Example of an interrupt service table.*

| Interrupt | Assigned pin | Active/Inactive | Priority | ISR function |
|---|---|---|---|---|
| External hardware interrupt (INT0) | D13 | Active | 1 | *ISRInt0()* |
| External hardware interrupt (INT1) | D17 | Active | 2 | *ISRInt1()* |
| UART0 received byte interrupt | – | Inactive | – | *ISRUart0()* |
| UART1 received byte interrupt | – | Inactive | – | – |
| Timer0 elapsed time interrupt | – | Active | 3 | *ISRTimer0()* |

In Table 7.7, it can be seen that interrupts can be triggered by UARTs and timers. If an interrupt from a UART is activated when a new byte (i.e., a character) is received by the UART, a given function will be executed (i.e., *ISRUart0()*). This behavior can be used to avoid *polling* the UART at regular times, as in the examples in previous chapters.

One ISR can be interrupted by another ISR, as shown in Figure 7.10. It should be noted that the priority number is used to determine which interrupt must be attended to first. In the example, ISR1 has a higher priority than ISR2. ISR1 is not interrupted by the occurrence of ISR2 (left of Figure 7.10), while ISR2 is interrupted by the occurrence of ISR1 (right of Figure 7.10).

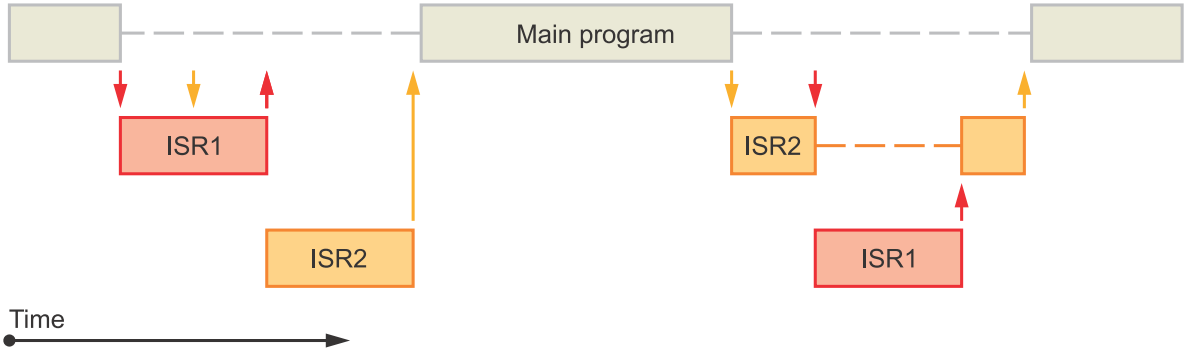



*Figure 7.10 Conceptual diagram of the normal flow of a program altered by two interrupt service routines.*

**NOTE:** Given that one ISR can be interrupted by another ISR, and given that the normal flow of the program will be altered when an ISR is called, the complexity and operation time of the ISRs must be kept as small as possible.

Besides hardware interrupts, there are other types of interrupts, such as software interrupt instructions or software exceptions. For example, if a division by zero is executed, then a software interrupt can take place. If this occurs, the programmer may provide a piece of code to be executed to attempt to overcome the issue. This specific piece of code, also called an ISR, may be responsible for notifying the user that a division by zero is not allowed or may just set a given Boolean variable in order to later, when possible, notify the user that a division by zero is not allowed.

In the examples below, interrupts will be used to detect when buttons *Dir1*, *Dir2*, *Dir1LS*, and *Dir2LS* are pressed, as well as to detect when the PIR motion detector is activated.

**NOTE:** The situations described in this chapter (for example, intruder detection or motor activation) might also be implemented without using interrupts. Interrupts are chosen here with the aim of introducing the topic.

**Example 7.1: Control a DC Motor using Interrupts**

### Objective

Introduce the use of a direct current motor.

### Summary of the Expected Behavior

By means of buttons *Dir1* and *Dir2*, the rotation direction of the motor is controlled. Two LEDs are used to indicate the direction in which the motor is turning.

### Test the Proposed Solution on the Board

Import the project "Example 7.1" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Press "m" on the PC keyboard. A message indicating "The motor is stopped" should be displayed on the PC. Press button *Dir1*. The motor should turn in one direction, and one of the LEDs should turn on. Press "m" again on the PC keyboard. A message indicating "The motor is turning in direction 1" should be displayed on the PC. Press button *Dir2*. The motor should turn in the other direction, and the other LED should turn on. Press "m" again on the PC keyboard. A message indicating "The motor is turning in direction 2" should be displayed on the PC.

### Discussion of the Proposed Solution

The proposed solution is based on a new software module named *motor* and some new lines in the *user_interface* module. The motor is controlled by means of a new set of ISRs, which are triggered by the buttons *Dir1* and *Dir2*.

### Implementation of the Proposed Solution

The initialization of the *motor* module is done at the beginning of the program by calling the function *motorControlInit()* from *smartHomeSystemInit()*, as can be seen on line 6 of Code 7.1. The function *motorControlUpdate()* is included in *smartHomeSystemUpdate()* (line 15). In order to implement these calls, the library *motor* is included in *smart_home_system.cpp*, as can be seen in Table 7.8.

```
1  void smartHomeSystemInit()
2  {
3      userInterfaceInit();
4      fireAlarmInit();
5      pcSerialComInit();
6      motorControlInit();
7  }
8
9  void smartHomeSystemUpdate()
10 {
11     userInterfaceUpdate();
12     fireAlarmUpdate();
13     pcSerialComUpdate();
14     eventLogUpdate();
15     motorControlUpdate();
16     delay(SYSTEM_TIME_INCREMENT_MS);
17 }
```

***Code 7.1 New implementation of the functions smartHomeSystemInit() and smartHomeSystemUpdate().***

***Table 7.8 Sections in which lines were added to smart_home_system.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "motor.h"` |

The implementation of *motor.cpp* is shown in Code 7.2 and Code 7.3. The libraries that are included are shown from line 3 to line 6 of Code 7.2. On line 10, the motor update time is defined. On lines 14 and 15, the global objects that will control the motor are created and assigned to available pins. It is necessary to declare these objects as *DigitalInOut* to allow the pin to be configured as *unconnected*. This is achieved by using the *.mode(OpenDrain)* configuration. On lines 19 and 20, two variables of the data type *motorDirection_t* (defined in *motor.h*) are declared.

**NOTE:** For the sake of brevity, only the file sections that have some content are shown in the Code. The full versions of the files are available in [3].

```
1  //=====[Libraries]=============================================================
2
3  #include “mbed.h”
4  #include “arm_book_lib.h”
5
6  #include “motor.h”
7
8  //=====[Declaration of private defines]========================================
9
10 #define MOTOR_UPDATE_TIME 9
11
12 //=====[Declaration and initialization of public global objects]===============
13
14 DigitalInOut motorM1Pin(PF_2);
15 DigitalInOut motorM2Pin(PE_3);
16
17 //=====[Declaration and initialization of public global variables]=============
18
```

```
19  motorDirection_t motorDirection;
20  motorDirection_t motorState;
21
22  //=====[Implementations of public functions]=================================
23
24  void motorControlInit()
25  {
26      motorM1Pin.mode(OpenDrain);
27      motorM2Pin.mode(OpenDrain);
28
29      motorM1Pin.input();
30      motorM2Pin.input();
31
32      motorDirection = STOPPED;
33      motorState = STOPPED;
34  }
35
36  motorDirection_t motorDirectionRead()
37  {
38      return motorDirection;
39  }
40
41  void motorDirectionWrite( motorDirection_t direction )
42  {
43      motorDirection = direction;
44  }
```

***Code 7.2 Implementation of motor.cpp file (Part 1/2).***

On lines 24 to 34, the implementation of the function *motorControlInit()* is shown. On lines 26 and 27, the pins connected to the motor are configured as *open drain* and on lines 29 and 30 as input. In this way, both pins are in a *high impedance* state (which can be considered as unconnected), so the relays that control the motor are not energized. On lines 32 and 33, *motorDirection* and *motorState* are initialized as STOPPED. From lines 36 to 44, the implementations of the functions *motorDirectionRead()*, which returns the value of the variable *motorDirection*, and *motorDirectionWrite*, which assigns the received parameter (*Direction*) to *motorDirection*, are shown.

In Code 7.3, the implementation of the function *motorControlUpdate()* is shown. This function is responsible for the operation of the motor depending on the value of the variable *motorState*. The finite-state machine that controls the motor is executed every 100 ms, taking into account the value of MOTOR_UPDATE_TIME and the fact that *motorControlUpdate()* is called every 10 ms. If the value of *motorState* is DIRECTION_1 (the motor is turning in DIRECTION_1) and the value of *motorDirection* is DIRECTION_2 or STOPPED (lines 13 and 14), then the motor is stopped by putting both motor pins in high impedance (lines 15 and 16). The same behavior is implemented for the other direction (lines 21 to 28).

In the STOPPED state (lines 30 to 45), the direction of the motor is defined depending on the value of *motorDirection*. The pin that corresponds to the received direction is configured as output (lines 34 and 41), and LOW is assigned to it in order to activate the motor (lines 35 and 42), while the other pin is configured in high impedance (lines 33 and 40). In this way, only the relay that corresponds to the pin of the selected direction will be energized. In all cases, the variable *motorState* is updated (lines 17, 26, 36, and 43).

```
 1  void motorControlUpdate()
 2  {
 3      static int motorUpdateCounter = 0;
 4
 5      motorUpdateCounter++;
 6
 7      if ( motorUpdateCounter > MOTOR_UPDATE_TIME ) {
 8
 9          motorUpdateCounter = 0;
10
11          switch ( motorState ) {
12              case DIRECTION_1:
13                  if ( motorDirection == DIRECTION_2 ||
14                       motorDirection == STOPPED ) {
15                      motorM1Pin.input();
16                      motorM2Pin.input();
17                      motorState = STOPPED;
18                  }
19                  break;
20
21              case DIRECTION_2:
22                  if ( motorDirection == DIRECTION_1 ||
23                       motorDirection == STOPPED ) {
24                      motorM1Pin.input();
25                      motorM2Pin.input();
26                      motorState = STOPPED;
27                  }
28                  break;
29
30              case STOPPED:
31              default:
32                  if ( motorDirection == DIRECTION_1 ) {
33                      motorM2Pin.input();
34                      motorM1Pin.output();
35                      motorM1Pin = LOW;
36                      motorState = DIRECTION_1;
37                  }
38
39                  if ( motorDirection == DIRECTION_2 ) {
40                      motorM1Pin.input();
41                      motorM2Pin.output();
42                      motorM2Pin = LOW;
43                      motorState = DIRECTION_2;
44                  }
45                  break;
46          }
47      }
48  }
```

***Code 7.3 Implementation of motor.cpp file (Part 2/2).***

In Code 7.4, the implementation of *motor.h* is shown. It can be seen that the data type *motorDirection_t* is defined on lines 8 to 12, and the prototypes of the public functions implemented in Code 7.3 are declared on lines 16 to 21.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _MOTOR_H_
4  #define _MOTOR_H_
5
6  //=====[Declaration of public data types]======================================
7
8  typedef enum {
9      DIRECTION_1,
10     DIRECTION_2,
11     STOPPED
12 } motorDirection_t;
13
14 //=====[Declarations (prototypes) of public functions]=========================
15
16 void motorControlInit();
17 void motorDirectionWrite( motorDirection_t direction );
18
19 motorDirection_t motorDirectionRead();
20
21 void motorControlUpdate();
22
23 //=====[#include guards - end]=================================================
24
25 #endif // _MOTOR_H_
```

***Code 7.4 Implementation of motor.h file.***

In Table 7.9, the sections in which lines were added to *user_interface.cpp* are shown. It can be seen that *motor.h* has been included, and the public global objects of type InterruptIn *motorDirection1Button* and *motorDirection2Button* were assigned to the pins PF_9 and PF_8, respectively. The private functions *motorDirection1ButtonCallback()* and *motorDirection2ButtonCallback()* are declared.

***Table 7.9 Sections in which lines were added to user_interface.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "motor.h"` |
| Declaration and initialization of public global objects | `InterruptIn motorDirection1Button(PF_9);`<br>`InterruptIn motorDirection2Button(PF_8);` |
| Declarations (prototypes) of private functions | `static void motorDirection1ButtonCallback();`<br>`static void motorDirection2ButtonCallback();` |

In Code 7.5, the new implementation of the function *userInterfaceInit()* of the module *user_interface* is shown. On lines 3 and 4, the two buttons that will control the direction of the motors are configured with an internal pull-up resistor. On lines 6 and 7, the interrupt is configured for these two buttons.

Whenever a transition from high to low state (*falling edge*) is detected in those pins, a *callback function* is called. These functions are referred to as the *handlers* for the interrupts related to *motorDirection1Button.fall* and *motorDirection2Button.fall*. For pin PF_9, the callback function is *motorDirection1ButtonCallback()*, and for pin PF_8 the callback function is *motorDirection2ButtonCallback()*. Note that the callback functions are preceded by the reference operator (&). Lines 9 to 12 remain unchanged from the previous version of this function.

```
1  void userInterfaceInit()
2  {
3      motorDirection1Button.mode(PullUp);
4      motorDirection2Button.mode(PullUp);
5
6      motorDirection1Button.fall(&motorDirection1ButtonCallback);
7      motorDirection2Button.fall(&motorDirection2ButtonCallback);
8
9      incorrectCodeLed = OFF;
10     systemBlockedLed = OFF;
11     matrixKeypadInit( SYSTEM_TIME_INCREMENT_MS );
12     userInterfaceDisplayInit();
13 }
```

*Code 7.5 New implementation of the function userInterfaceInit().*

In Code 7.6, the implementation of the callbacks *motorDirection1ButtonCallback()* and *motorDirection2ButtonCallback()* is shown. Each of these functions calls *motorDirectionWrite()* (shown in Code 7.3) with the parameter DIRECTION_1 (line 3) or DIRECTION_2 (line 8).

```
1  static void motorDirection1ButtonCallback()
2  {
3      motorDirectionWrite( DIRECTION_1 );
4  }
5
6  static void motorDirection2ButtonCallback()
7  {
8      motorDirectionWrite( DIRECTION_2 );
9  }
```

*Code 7.6 Implementation of the functions motorDirection1ButtonCallback() and motorDirection2ButtonCallback().*

To implement the new command "m", the lines shown in Table 7.10 were added to *pcSerialComCommandUpdate()* and *availableCommands()* in *pc_serial_com.cpp*. In Table 7.11, the sections in which lines were added to *pc_serial_com.cpp* are shown. It can be seen that a new private function, *commandShowCurrentMotorState()*, is declared.

*Table 7.10 Functions in which lines were added in pc_serial_com.cpp.*

| Functions | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'm': case 'M': commandShowCurrentMotorState(); break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'm' or 'M' to show the motor status\r\n" );` |

*Table 7.11 Sections in which lines were added in pc_serial_com.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "motor.h"` |
| Declarations (prototypes) of private functions | `static void commandShowCurrentMotorState();` |

When "m" is pressed on the PC keyboard, the function *commandShowCurrentMotorState()* shown in Code 7.7 is called. On line 3, the function *motorDirectionRead()* (shown in Code 7.2) is called. One of three different messages is sent to the PC console (lines 5 to 9) depending on the returned value.

```
1  static void commandShowCurrentMotorState()
2  {
3      switch ( motorDirectionRead() ) {
4          case STOPPED:
5              pcSerialComStringWrite( "The motor is stopped\r\n" ); break;
6          case DIRECTION_1:
7              pcSerialComStringWrite( "The motor is turning in direction 1\r\n" ); break;
8          case DIRECTION_2:
9              pcSerialComStringWrite( "The motor is turning in direction 2\r\n" ); break;
10     }
11 }
```

***Code 7.7 Implementation of the function commandShowCurrentMotorState().***

### Proposed Exercise

1. How can the program be modified in order to include a new button that stops the motor, regardless of which direction it is turning?

### Answer to the Exercise

1. A new button should be included, and code should be implemented following the same procedure as the direction buttons, with the difference that the callback function should use the function *motorDirectionWrite* with STOPPED as the parameter.

## Example 7.2: Use a DC Motor to Open and Close a Gate

### Objective

Expand the functionality of the external interrupts and modify the code to include a gate.

### Summary of the Expected Behavior

By means of the buttons *Dir1* and *Dir2*, the motor rotation direction is controlled. The buttons *Dir1LS* and *Dir2LS* are used to indicate that a gate has reached a *Limit Switch*. In that situation, the motor should stop.

**NOTE:** In the implementation proposed in this example, the actual gate is not included; this gate might be the gate of a house or any other gate that the user might choose.

### Test the Proposed Solution on the Board

Import the project "Example 7.2" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Press button *Dir1*. The motor should turn in a given direction, and one of the

LEDs should turn on. Press "g" on the PC keyboard. A message indicating "The gate is opening" should be displayed on the PC. Press the button *Dir1LS*. The motor should stop, and LED1 should turn off. Press "g" again on the PC keyboard. A message indicating "The gate is open" should be displayed on the PC.

Press button *Dir2*. The motor should turn in the other direction, and the other LED should turn on. Press "g" again on the PC keyboard. A message indicating "The gate is closing" should be displayed on the PC. Press button *Dir2LS*. The motor should stop, and LED2 should turn off. Press "g" again on the PC keyboard. A message indicating "The gate is closed" should be displayed on the PC.

## Discussion of the Proposed Solution

The proposed solution is based on a new module, named *gate*. The motor and the gate are controlled by means of a new set of ISRs, which are triggered by the buttons *Dir1LS* and *Dir2LS*.

## Implementation of the Proposed Solution

The initialization of the *gate* module is done at the beginning of the program by means of a call to the function *gateInit()* from *smartHomeSystemInit()*, as can be seen on line 7 of Code 7.8. In order to implement this call, the library *gate* is included in *smart_home_system.cpp*, as can be seen in Table 7.12.

```
1  void smartHomeSystemInit()
2  {
3      userInterfaceInit();
4      fireAlarmInit();
5      pcSerialComInit();
6      motorControlInit();
7      gateInit();
8  }
```

***Code 7.8 New implementation of the function smartHomeSystemInit().***

***Table 7.12 Sections in which lines were added to smart_home_system.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "gate.h"` |

Since, in this example, the motor is associated with a gate, some modifications in the code are needed in the module *user_interface*. The user will open or close a gate instead of turning a motor in direction 1 or 2. To account for this change, the variables and functions related to the motor are renamed as shown in Table 7.13 and Table 7.14 and in the new implementation of *userInterfaceInit()* shown in Code 7.9 (lines 3 to 7).

***Table 7.13 Public global objects that were renamed in user_interface.cpp.***

| Object name in Example 7.1 | Object name in Example 7.2 |
|---|---|
| InterruptIn motorDirection1Button(PF_9); | InterruptIn gateOpenButton(PF_9); |
| InterruptIn motorDirection2Button(PF_8); | InterruptIn gateCloseButton(PF_8); |

*Table 7.14 Private functions that were renamed in user_interface.cpp.*

| Function name in Example 7.1 | Function name in Example 7.2 |
|---|---|
| static void motorDirection1ButtonCallback(); | static void gateOpenButtonCallback(); |
| static void motorDirection2ButtonCallback(); | static void gateCloseButtonCallback(); |

```
1  void userInterfaceInit()
2  {
3      gateOpenButton.mode(PullUp);
4      gateCloseButton.mode(PullUp);
5
6      gateOpenButton.fall(&gateOpenButtonCallback);
7      gateCloseButton.fall(&gateCloseButtonCallback);
8
9      incorrectCodeLed = OFF;
10     systemBlockedLed = OFF;
11     matrixKeypadInit( SYSTEM_TIME_INCREMENT_MS );
12     userInterfaceDisplayInit();
13 }
```

*Code 7.9 New implementation of the function userInterfaceInit().*

Code 7.6 from Example 7.1 is modified as shown in Code 7.10. In this example, the rotation direction of the motor represents the opening or closing of the gate, so the functions *gateOpen()* and *gateClose()* from the Gate module are used. In order to use these functions, *gate.h* is included as shown in Table 7.15.

```
1  static void gateOpenButtonCallback()
2  {
3      gateOpen();
4  }
5
6  static void gateCloseButtonCallback()
7  {
8      gateClose();
9  }
```

*Code 7.10 Changes in the name and implementation of functions of user_interface.cpp file.*

*Table 7.15 Sections in which lines were added to user_interface.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "gate.h"` |

The implementation of *gate.cpp* is shown in Code 7.11 and Code 7.12. The libraries that are included are shown from lines 3 to 7 of Code 7.11. The external interrupts are assigned to pins PG_1 and PF_7 and declared on lines 11 and 12, respectively. On lines 16 and 17, two private global variables are created that will store the state of each limit switch when they are pressed. The interrupt handlers of each of the external interrupts are declared on lines 23 and 24.

The function *gateInit()* is shown on lines 28 to 39 of Code 7.11. The pins that simulate the limit switches of the gate are configured with internal pull-up resistors on lines 30 and 31, and the callbacks are defined on lines 33 and 34. Finally, on lines 36 to 38, the variables that store the status of the limit switches and the gate are initialized, setting the gate to closed.

**NOTE:** If the gate is not closed during the initialization, the system will synchronize with the limit switches as soon as the gate open limit switch or the gate close limit switch is activated.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "gate.h"
7  #include "motor.h"
8
9  //=====[Declaration and initialization of public global objects]===============
10
11 InterruptIn gateOpenLimitSwitch(PG_1);
12 InterruptIn gateCloseLimitSwitch(PF_7);
13
14 //=====[Declaration and initialization of private global variables]============
15
16 static bool gateOpenLimitSwitchState;
17 static bool gateCloseLimitSwitchState;
18
19 static gateStatus_t gateStatus;
20
21 //=====[Declarations (prototypes) of private functions]=======================
22
23 static void gateOpenLimitSwitchCallback();
24 static void gateCloseLimitSwitchCallback();
25
26 //=====[Implementations of public functions]==================================
27
28 void gateInit()
29 {
30     gateOpenLimitSwitch.mode(PullUp);
31     gateCloseLimitSwitch.mode(PullUp);
32
33     gateOpenLimitSwitch.fall(&gateOpenLimitSwitchCallback);
34     gateCloseLimitSwitch.fall(&gateCloseLimitSwitchCallback);
35
36     gateOpenLimitSwitchState = OFF;
37     gateCloseLimitSwitchState = ON;
38     gateStatus = GATE_CLOSED;
39 }
```

***Code 7.11 Implementation of gate.cpp file (Part 1/2).***

In Code 7.12, the implementations of the functions *gateOpen()* and *gateClose()* are shown from lines 1 to 17. If the state of the limit switch is OFF (lines 3 and 12), the motor is set to the requested direction (lines 4 and 13), the status of the gate is updated (lines 5 and 14), and the opposite limit switch is set to OFF (lines 6 and 15). Lines 19 to 22 show the implementation of the function *gateStatusRead()*, which returns the value of the variable *gateStatus*.

The implementation of the private functions that handle the interrupts is shown from lines 26 to 42. These handlers have similarities with the implementations of the functions *gateOpen()* and *gateClose()*. If the motor is turning in the direction that corresponds to the callback (lines 28 and 37), then the motor is stopped (lines 29 and 38), the status of the gate is updated (lines 30 and 39), and the limit switch is set to ON (lines 31 and 40).

```
 1  void gateOpen()
 2  {
 3      if ( !gateOpenLimitSwitchState ) {
 4          motorDirectionWrite( DIRECTION_1 );
 5          gateStatus = GATE_OPENING;
 6          gateCloseLimitSwitchState = OFF;
 7      }
 8  }
 9
10  void gateClose()
11  {
12      if ( !gateCloseLimitSwitchState ) {
13          motorDirectionWrite( DIRECTION_2 );
14          gateStatus = GATE_CLOSING;
15          gateOpenLimitSwitchState = OFF;
16      }
17  }
18
19  gateStatus_t gateStatusRead()
20  {
21      return gateStatus;
22  }
23
24  //=====[Implementations of private functions]==========================
25
26  static void gateOpenLimitSwitchCallback()
27  {
28      if ( motorDirectionRead() == DIRECTION_1 ) {
29          motorDirectionWrite(STOPPED);
30          gateStatus = GATE_OPEN;
31          gateOpenLimitSwitchState = ON;
32      }
33  }
34
35  static void gateCloseLimitSwitchCallback()
36  {
37      if ( motorDirectionRead() == DIRECTION_2 ) {
38          motorDirectionWrite(STOPPED);
39          gateStatus = GATE_CLOSED;
40          gateCloseLimitSwitchState = ON;
41      }
42  }
```

***Code 7.12 Implementation of gate.cpp file (Part 2/2).***

In Code 7.13, the implementation of *gate.h* is shown. It can be seen that the data type *gateStatus_t* is defined on lines 8 to 13, and the prototypes of the public functions defined in Code 7.11 and Code 7.12 are declared on lines 17 to 22.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _GATE_H_
4  #define _GATE_H_
5
6  //=====[Declaration of public data types]======================================
7
8  typedef enum {
9      GATE_CLOSED,
10     GATE_OPEN,
11     GATE_OPENING,
12     GATE_CLOSING,
13 } gateStatus_t;
14
15 //=====[Declarations (prototypes) of public functions]=========================
16
17 void gateInit();
18
19 void gateOpen();
20 void gateClose();
21
22 gateStatus_t gateStatusRead();
23
24 //=====[#include guards - end]=================================================
25
26 #endif // _GATE_H_
```

***Code 7.13 Implementation of gate.h file.***

To implement the new command "g", the lines shown in Table 7.16 are added to *pcSerialComCommand Update()* and *availableCommands()* in *pc_serial_com.cpp*. In Table 7.17, the sections in which lines are added to *pc_serial_com.cpp* are shown. It can be seen that a new private function, *commandShowCurrentMotorState()*, is declared.

***Table 7.16 Functions in which lines were added in pc_serial_com.cpp.***

| Function | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'g': case 'G': commandShowCurrentGateState(); break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'g' or 'G' to show the gate status\r\n" );` |

***Table 7.17 Sections in which lines were added to pc_serial_com.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "gate.h"` |
| Declarations (prototypes) of private functions | `static void commandShowCurrentGateState();` |

When "g" is pressed on the PC keyboard, the function *commandShowCurrentGateState()* shown in Code 7.14 is called. On line 3, the function *gateStatusRead()* (shown in Code 7.12) is called and, depending on the returned value, one of four different messages is sent to the PC console (lines 4 to 7).

```
1  static void commandShowCurrentGateState()
2  {
3      switch ( gateStatusRead() ) {
4          case GATE_CLOSED: pcSerialComStringWrite( "The gate is closed\r\n" ); break;
5          case GATE_OPEN: pcSerialComStringWrite( "The gate is open\r\n" ); break;
6          case GATE_OPENING: pcSerialComStringWrite( "The gate is opening\r\n" ); break;
7          case GATE_CLOSING: pcSerialComStringWrite( "The gate is closing\r\n" ); break;
8      }
9  }
```

***Code 7.14 Implementation of the function commandShowCurrentGateState().***

### Proposed Exercise

1. What should be changed in the code to detect the buttons if they are now connected to 3.3 V instead of GND?

### Answer to the Exercise

1. In Code 7.11, lines 30 and 31 should be modified to use the *pullDown* parameter, and lines 33 and 34 should use the *rise* interrupt type.

## Example 7.3: Use of a PIR Sensor to Detect Intruders

### Objective

Introduce the reading of a PIR sensor using interrupts.

### Summary of the Expected Behavior

Intruders are detected by the PIR sensor, and the corresponding event is registered in the event log.

### Test the Proposed Solution on the Board

Import the project "Example 7.3" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Wave a hand over the PIR sensor. A message indicating "MOTION_ON" should be displayed on the serial terminal. Then, after a time defined by $t_{Delay}$, a message indicating "MOTION_OFF" should be displayed on the serial terminal. Press "h" on the PC keyboard or "B" on the matrix keypad. The system will stop tracking the PIR sensor. Press "i" on the PC keyboard or "A" on the matrix keypad. The system will restart its tracking of the PIR sensor.

**NOTE:** If key "h" or "B" is pressed just when the PIR sensor has detected a movement, the message "The motion sensor has been deactivated" followed by "MOTION_ON" can be seen on the serial terminal. After this last activation, the PIR sensor will not be activated again until key "i" or "A" is pressed. If key "h" or "B" is pressed many times, "The motion sensor has been deactivated" will be printed many times. If "i" or "A" is pressed many times, "The motion sensor has been activated" will be printed many times.

## Discussion of the Proposed Solution

The proposed solution is based on a new module named *motion_sensor*. This module makes use of an interrupt to detect the pulse that the PIR sensor generates when it detects movement. An interrupt is triggered by a rising edge of the PIR sensor output signal. To detect the end of the pulse, an interrupt that is triggered when the signal transitions from high to low state (*falling edge*) is enabled. Figure 7.11 illustrates the initialization of the *motion_sensor* module (i.e., *motionSensorInit()*), the pulse generated when motion is detected, the content of the interrupt callback triggered (i.e., *motionDetected()*), and the content of the callback triggered when the pulse ceases (i.e., *motionCeased()*).

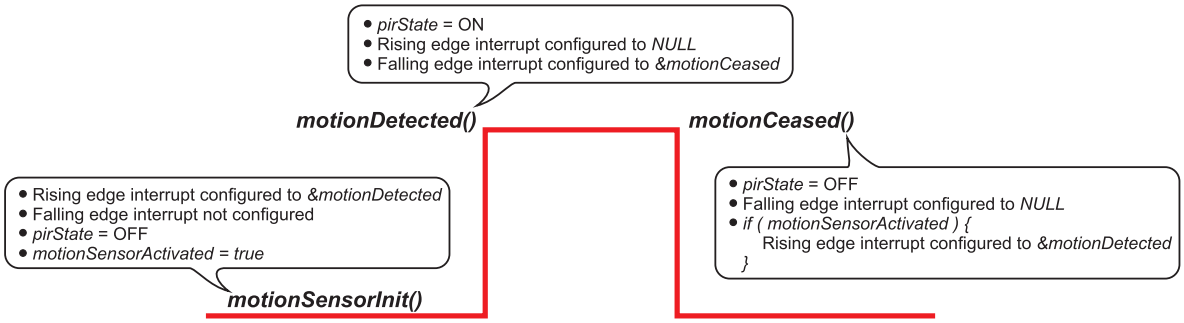



***Figure 7.11 Pulse generated by PIR sensor when motion is detected and the corresponding initialization and callbacks.***

In the initialization, the callback of the rising edge interrupt of *pirOutputSignal* (i.e., PG_0) is configured to *motionDetected()*, the callback of its falling edge interrupt is not configured, the Boolean variable *pirState* is assigned the OFF state, and *motionSensorActivated* is assigned true. When motion is detected by the PIR sensor, a rising edge pulse is generated on *pirOutputSignal*, which triggers the rising edge interrupt. This interrupt calls its callback function, *motionDetected()*, which assigns ON to *pirState*, disables the rising edge interrupt (i.e., NULL is assigned), activates the *pirOutputSignal* falling edge interrupt, and configures its callback to *motionCeased()*. Lastly, when motion ceases, a falling edge pulse is generated on *pirOutputSignal*, which triggers the falling edge interrupt. This interrupt calls its callback function, *motionCeased()*, which assigns OFF to *pirState*, disables the falling edge interrupt (i.e., NULL is assigned), and, if *motionSensorActivated* is true, then activates the *pirOutputSignal* rising edge interrupt and configures its callback to *motionDetected()*.

In the implementation introduced in this example, it is possible to deactivate the motion sensor detection at any time by pressing keys "h" or "B", even when the PIR sensor is detecting motion. In that situation, as was seen in the "Test the Proposed Solution on the Board" section, the message "The motion sensor has been deactivated" will be shown on the serial terminal, and the PIR sensor will not be activated again until keys "i" or "A" are pressed. Note that if the motion sensor detection is deactivated when the PIR sensor is detecting motion, then the falling edge interrupt will be disabled when *motionCeased()* is called as a consequence of the falling edge on *pirOutputSignal*. This is discussed below as the corresponding program code is shown.

## Implementation of the Proposed Solution

Code 7.15 shows the new implementation of the function *smartHomeSystemInit()*. It can be seen that the function *motionSensorInit()* is called on line 8 to initialize the *motion_sensor* module.

```
1  void smartHomeSystemInit()
2  {
3      userInterfaceInit();
4      fireAlarmInit();
5      pcSerialComInit();
6      motorControlInit();
7      gateInit();
8      motionSensorInit();
9  }
```

***Code 7.15 Details of the new implementation of the function smartHomeSystemInit().***

In Table 7.18, the sections in which lines were added to *smart_home_system.cpp* are shown. It can be seen that *motion_sensor.h* has been included.

***Table 7.18 Sections in which lines were added to smart_home_system.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "motion_sensor.h"` |

To implement the new commands "i" and "h", the lines shown in Table 7.19 were added to *pcSerialComCommandUpdate()* and *availableCommands()* in *pc_serial_com.cpp*. In Table 7.19, the sections in which lines were added to *pc_serial_com.cpp* are shown. It can be seen that two new private functions are declared: *commandMotionSensorActivate()* and *commandMotionSensorDeactivate()*.

***Table 7.19 Functions in which lines were added in pc_serial_com.cpp***

| Function | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'i': case 'I': commandMotionSensorActivate();`<br>`  break;`<br>`case 'h': case 'H': commandMotionSensorDeactivate();`<br>`  break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'i' or 'I' to`<br>`  activate the motion sensor\r\n" );`<br>`pcSerialComStringWrite( "Press 'h' or 'H' to`<br>`  deactivate the motion sensor\r\n" );` |

***Table 7.20 Sections in which lines were added in pc_serial_com.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "motion_sensor.h"` |
| Declarations (prototypes) of private functions | `static void commandMotionSensorActivate();`<br>`static void commandMotionSensorDeactivate();` |

In Code 7.16, the implementations of *commandMotionSensorActivate()* and *commandMotionSensor Deactivate()* are shown. These functions call *motionSensorActivate()* and *motionSensorDeactivate()*, respectively.

```
1  static void commandMotionSensorActivate()
2  {
3      motionSensorActivate();
4  }
5
6  static void commandMotionSensorDeactivate()
7  {
8      motionSensorDeactivate();
9  }
```

***Code 7.16 Implementation of commandMotionSensorActivate() and commandMotionSensorDeactivate().***

The implementation of *motion_sensor.cpp* is shown in Code 7.17 and Code 7.18. The libraries that are included are shown from lines 3 to 7. On line 11, a public global object of type InterruptIn named *pirOutputSignal* is declared and assigned to the pin PG_0. This pin will be used to detect the pulse generated by the PIR sensor when it identifies movement, as was explained using Figure 7.11. This pulse will be processed by the private functions *motionDetected()* and *motionCeased()*, declared on lines 20 and 21, respectively. It will modify the state of the global private variable named *pirState*, which is declared on line 15. Finally, another global private variable named *motionSensorActivated*, which is declared on line 16, will define whether the tracking of the motion sensor is active.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "motion_sensor.h"
7  #include "pc_serial_com.h"
8
9  //=====[Declaration and initialization of public global objects]===============
10
11 InterruptIn pirOutputSignal(PG_0);
12
13 //=====[Declaration and initialization of private global variables]============
14
15 static bool pirState;
16 static bool motionSensorActivated;
17
18 //=====[Declarations (prototypes) of private functions]=======================
19
20 static void motionDetected();
21 static void motionCeased();
```

***Code 7.17 Details of the implementation of motion_sensor.cpp (Part 1/2).***

The implementation of public and private functions of the *motion_sensor* module is shown in Code 7.18. From lines 3 to 8, the function *motionSensorInit()* is implemented. On line 5, the callback function of *pirOutputSignal* interrupt is configured with the function *motionDetected()* when there is a rising edge on PG_0 (i.e., when there is a rising edge on PG_0 the function *motionDetected()* is called).

On line 6, *pirState* is initialized to OFF because it is assumed that at the beginning the PIR sensor is inactive. On line 7, *motionSensorActivated* is initialized to true in order to activate the motion sensor. Therefore, the tracking of this sensor will be active, since the smart home system is initialized. From lines 10 to 13, the public function *motionSensorRead()* is implemented. This function returns the value of *pirState*.

On line 15, *motionSensorActivate()* is implemented. First, it assigns true to *motionSensorActivated*. Then, if *pirState* is OFF (line 18), it configures an interrupt associated with a rising edge on *pirOutputSignal* with *motionDetected()* as its callback. Note that if *pirState* is ON, this rising edge interrupt will be configured by the callback associated with the falling edge interrupt, as discussed above (Figure 7.11). Finally, this function sends the string "The motion sensor has been activated" to the serial terminal (line 21).

On line 24, *motionSensorDeactivate()* is implemented. First, it assigns false to *motionSensorActivated*. Then, if *pirState* is OFF (line 27), it disables the interrupt associated with a rising edge on *pirOutputSignal* (line 28). Note that if *pirState* is ON, this rising edge interrupt will be configured by the callback associated with the rising edge interrupt, as discussed above (Figure 7.11). Lastly, this function sends the string "The motion sensor has been deactivated" to the serial terminal (line 30).

```
1  //=====[Implementations of public functions]=================================
2
3  void motionSensorInit()
4  {
5      pirOutputSignal.rise(&motionDetected);
6      pirState = OFF;
7      motionSensorActivated = true;
8  }
9
10 bool motionSensorRead()
11 {
12     return pirState;
13 }
14
15 void motionSensorActivate()
16 {
17     motionSensorActivated = true;
18     if ( !pirState ) {
19         pirOutputSignal.rise(&motionDetected);
20     }
21     pcSerialComStringWrite( "The motion sensor has been activated\r\n" );
22 }
23
24 void motionSensorDeactivate()
25 {
26     motionSensorActivated = false;
27     if ( !pirState ) {
28         pirOutputSignal.rise(NULL);
29     }
30     pcSerialComStringWrite("The motion sensor has been deactivated\r\n");
31 }
32
```

```
33  //=====[Implementations of private functions]==================================
34
35  static void motionDetected()
36  {
37      pirState = ON;
38      pirOutputSignal.rise(NULL);
39      pirOutputSignal.fall(&motionCeased);
40  }
41
42  static void motionCeased()
43  {
44      pirState = OFF;
45      pirOutputSignal.fall(NULL);
46      if ( motionSensorActivated ) {
47          pirOutputSignal.rise(&motionDetected);
48      }
49  }
```

***Code 7.18 Details of the implementation of motion_sensor.cpp (Part 2/2).***

As was mentioned earlier, when a rising edge is detected on *pirOutputSignal* (pin PG_0), the function *motionDetected()* is called (recall the interrupt that is configured on lines 5 and 19). This function sets *pirState* to ON (line 37) to keep track of the state of the PIR sensor, deactivates the rising edge interrupt (line 38), and configures a falling edge interrupt that triggers the function *motionCeased()* (line 39).

The function *motionCeased()*, from lines 42 to 49, first sets *pirState* to OFF on line 44. Then, on line 45, the falling edge interrupt is deactivated. On line 46, if *motionSensorActivated* is true, then *pirOutputSignal.rise(&motionDetected)* on line 47 is used to configure an interrupt to be triggered by a rising edge on PG_0 and to assign *motionDetected()* as its handler. In this way, it is established what to do when *pirOutputSignal* becomes active again. Note that if *motionSensorActivated* is false, then the rising interrupt is not enabled. Thus, the falling and the rising edge interrupts will be disabled and, therefore, the motion sensor is deactivated. To activate the motion sensor, "i" can be pressed on the PC keyboard. This causes *motionSensorActivate()* to be called, as was explained above.

In Code 7.19, the implementation of *motion_sensor.h* is shown. The prototypes of the public functions are declared from lines 8 to 11. The implementation of these functions was shown in Code 7.18.

```
1   //=====[#include guards - begin]===============================================
2
3   #ifndef _MOTION_SENSOR_H_
4   #define _MOTION_SENSOR_H_
5
6   //=====[Declarations (prototypes) of public functions]=========================
7
8   void motionSensorInit();
9   bool motionSensorRead();
10  void motionSensorActivate();
11  void motionSensorDeactivate();
12
13  //=====[#include guards - end]=================================================
14
15  #endif // _MOTION_SENSOR_H_
```

***Code 7.19 Details of the implementation of motion_sensor.h.***

Code 7.20 shows the new implementation of the function *eventLogUpdate()*. It can be seen that lines 23 to 25 have been added in order to determine if the state of the PIR sensor has changed since the last update. If a change has taken place, then the corresponding message is displayed on the serial terminal (line 24), and the state of *motionLastState* is updated (line 26).

```
 1  void eventLogUpdate()
 2  {
 3      bool currentState = sirenStateRead();
 4      eventLogElementStateUpdate( sirenLastState, currentState, "ALARM" );
 5      sirenLastState = currentState;
 6
 7      currentState = gasDetectorStateRead();
 8      eventLogElementStateUpdate( gasLastState, currentState, "GAS_DET" );
 9      gasLastState = currentState;
10
11      currentState = overTemperatureDetectorStateRead();
12      eventLogElementStateUpdate( tempLastState, currentState, "OVER_TEMP" );
13      tempLastState = currentState;
14
15      currentState = incorrectCodeStateRead();
16      eventLogElementStateUpdate( ICLastState, currentState, "LED_IC" );
17      ICLastState = currentState;
18
19      currentState = systemBlockedStateRead();
20      eventLogElementStateUpdate( SBLastState ,currentState, "LED_SB" );
21      SBLastState = currentState;
22
23      currentState = motionSensorRead();
24      eventLogElementStateUpdate( motionLastState ,currentState, "MOTION" );
25      motionLastState = currentState;
26  }
```

***Code 7.20 Details of the new implementation of the function eventLogUpdate().***

In Table 7.21, the sections in which lines were added to *event_log.cpp* are shown. It can be seen that *motion_sensor.h* has been included, and the private Boolean variable *motionLastState* has been declared and initialized to OFF.

***Table 7.21 Sections in which lines were added to event_log.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "motion_sensor.h"` |
| Declaration and initialization of private global variables | `static bool motionLastState   = OFF;` |

The matrix keypad can be used in order to activate or deactivate the tracking of the PIR sensor. In Table 7.22, the line added to *user_interface.cpp* to include the library *motion_sensor.h* is shown.

***Table 7.22 Sections in which lines were added to user_interface.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "motion_sensor.h"` |

In Code 7.21, the new implementation of *userInterfaceMatrixKeypadUpdate()* is shown. The new code is from lines 27 to 34. If the system is not blocked (line 27), then if the "A" key is pressed (line 28), *motionSensorActivate()* is called (line 29), and if the "B" key is pressed (line 30), *motionSensorDeactivate()* is called (line 32).

```
static void userInterfaceMatrixKeypadUpdate()
{
    static int numberOfHashKeyReleased = 0;
    char keyReleased = matrixKeypadUpdate();

    if( keyReleased != '\0' ) {

        if( sirenStateRead() && !systemBlockedStateRead() ) {
            if( !incorrectCodeStateRead() ) {
                codeSequenceFromUserInterface[numberOfCodeChars] = keyReleased;
                numberOfCodeChars++;
                if ( numberOfCodeChars >= CODE_NUMBER_OF_KEYS ) {
                    codeComplete = true;
                    numberOfCodeChars = 0;
                }
            } else {
                if( keyReleased == '#' ) {
                    numberOfHashKeyReleased++;
                    if( numberOfHashKeyReleased >= 2 ) {
                        numberOfHashKeyReleased = 0;
                        numberOfCodeChars = 0;
                        codeComplete = false;
                        incorrectCodeState = OFF;
                    }
                }
            }
        } else if ( !systemBlockedStateRead() ) {
            if( keyReleased == 'A' ) {
                motionSensorActivate();
            }
            if( keyReleased == 'B' ) {
                motionSensorDeactivate();
            }
        }
    }
}
```

***Code 7.21 New implementation of userInterfaceMatrixKeypadUpdate().***

## Proposed Exercises

1. How can the code be changed in order to use more than one PIR sensor?

2. Why are the new module in this example and its public functions called *motion_sensor* instead of *pir*?

3. Why are the functions *commandMotionSensorActivate()* and *commandMotionSensorDeactivate()* used in the module *pc_serial_com* instead of calling the public functions *motionSensorActivate()* and *motionSensorDeactivate()* directly?

## Answers to the Exercises

1. In *motion_sensor.cpp*, new InterruptIn objects must be declared and the corresponding functions to handle each interrupt must be written.

2. Because in this way, any code calling the module can treat its functions as independent of the implementation of the sensor. In this case a PIR sensor was used, but different technologies could be used to provide the same functionality for the smart home system; in this scenario, the public functions of the module would need to be rewritten, but their names would remain unchanged, as would any calling functions.

3. The functions *commandMotionSensorActivate()* and *commandMotionSensorDeactivate()* are used in the module *pc_serial_com* to make the implementation similar to the implementation used in the other commands.

**Example 7.4: Use of the PIR Sensor as an Intruder Detection Alarm**

### Objective

Trigger the alarm when an intruder is detected.

### Summary of the Expected Behavior

The siren and the alarm are also triggered by the PIR sensor.

### Test the Proposed Solution on the Board

Import the project "Example 7.4" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Wave a hand over the PIR sensor. The siren and the strobe light will turn on and off every 1000 milliseconds, and the display will show "Intruder Detected". Deactivate the alarm using the matrix keypad or the PC keyboard in the same way as in previous chapters. Press the B1 User button (from now on it will be called "Fire alarm test button"). The siren and the strobe light will turn on and off every 500 milliseconds, and the display will show "Fire Alarm Activated!". Wave a hand over the PIR sensor. The siren and the strobe light will turn on and off every 100 milliseconds. The display will indicate "Fire Alarm Activated!" because during a fire the smoke is also registered as movement by the PIR sensor.

**NOTE:** As discussed in the previous chapter, the on and off time of the siren and the strobe light are not always 100 ms. In the next chapter, a technique will be introduced to tackle this.

### Discussion of the Proposed Solution

The proposed solution is based on the modification of several parts of the code and on new software modules called *alarm* and *intruder_alarm*. The modifications are needed because in previous versions of the code, the alarm was only related to the fire detection subsystem. The *alarm* module will be responsible for checking if any of the alarm sources are active.

### Implementation of the Proposed Solution

In Table 7.23, the sections in which lines were added to *smart_home_system.cpp* are shown. It can be seen that *alarm.h* and *intruder_alarm.h* have been included.

*Table 7.23 Sections in which lines were added to smart_home_system.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "alarm.h"`<br>`#include "intruder_alarm.h"` |

Code 7.22 shows the new implementation of the functions *smartHomeSystemInit()* and *smartHomeSystemUpdate()*. It can be seen that the new functions *alarmInit()* and *intruderAlarmInit()* are called on lines 8 and 10, respectively, and *motionSensorInit()* has been removed, since this function is called by *intruderAlarmInit()*. The functions *intruderAlarmUpdate()* and *alarmUpdate()* are included in *smartHomeSystemUpdate()* (lines 18 and 19).

```
1  //=====[Implementations of public functions]=================================
2
3  void smartHomeSystemInit()
4  {
5      userInterfaceInit();
6      alarmInit();
7      fireAlarmInit();
8      intruderAlarmInit();
9      pcSerialComInit();
10     motorControlInit();
11     gateInit();
12 }
13
14 void smartHomeSystemUpdate()
15 {
16     userInterfaceUpdate();
17     fireAlarmUpdate();
18     intruderAlarmUpdate();
19     alarmUpdate();
20     eventLogUpdate();
21     pcSerialComUpdate();
22     delay(SYSTEM_TIME_INCREMENT_MS);
23 }
```

*Code 7.22 New implementation of the function smartHomeSystemInit and smartHomeSystemUpdate.*

The new module *alarm* is presented in Code 7.23, Code 7.24, Code 7.25, and Code 7.26. This module contains functionality that was previously carried out by the *fire_alarm* module. In Code 7.23, in lines 3 to 12, the libraries used in this module are included. The variable *alarmState* on line 22 and the private function *alarmDeactivate()* on line 26 are declared.

The strobe time of the siren and the strobe light, which was previously defined in the *fire_alarm* module, is defined in this module after the modifications to the code. Additionally, the different strobe times have new meanings. If only the intruder alarm is activated, the strobe time has a value of 1000 milliseconds. If only the fire alarm is activated, then the strobe time has a value of 500 milliseconds. Finally, if both the intruder alarm and the fire alarm are activated, the strobe time has a value of 100 milliseconds. These differences can be seen in the declaration of private #defines on lines 16 to 18 and the implementations of the private function *alarmStrobeTime()* (lines 12 to 27 in Code 7.25).

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "alarm.h"
7  #include "siren.h"
8  #include "strobe_light.h"
9  #include "code.h"
10 #include "matrix_keypad.h"
11 #include "fire_alarm.h"
12 #include "intruder_alarm.h"
13
14 //=====[Declaration of private defines]========================================
15
16 #define STROBE_TIME_INTRUDER_ALARM          1000
17 #define STROBE_TIME_FIRE_ALARM               500
18 #define STROBE_TIME_FIRE_AND_INTRUDER_ALARM  100
19
20 //=====[Declaration and initialization of private global variables]============
21
22 static bool alarmState;
23
24 //=====[Declarations (prototypes) of private functions]========================
25
26 static void alarmDeactivate();
```

***Code 7.23 Details of the implementation of alarm.cpp (Part 1/3).***

The implementation of public and private functions is shown in Code 7.24. The function *alarmInit()* (lines 3 to 7) is used to initialize the variable *alarmState* and the siren and strobe light using its public functions. The deactivation of the alarm, which was previously included in *fireAlarmDeactivationUpdate()*, has moved to the function *alarmUpdate()* (lines 10 to 30).

If a correct code is entered, then the function *alarmDeactivate()* is called (line 15). The function *alarmUpdate()* also updates the strobe time of the siren and the strobe light by means of the functions *sirenUpdate()* (line 19) and *strobeLightUpdate()* (line 20). Depending on the state of the alarm sources – gas, over temperature and intruder detection (lines 20 to 23) – *alarmState* (line 26), *sirenState* (line 27), and *strobeLightState* (line 28) are updated. Finally, the public function *alarmStateRead()* (lines 32 to 35) returns the value of *alarmState*.

The private function *alarmDeactivate()* (lines 3 to 10 of Code 7.25) implements the functionality that was previously located in *fireAlarmDeactivate()*, taking into account the new alarm source.

In Code 7.26, the implementation of *alarm.h* is shown. It can be seen that the prototypes of the public functions are declared from lines 8 to 10.

```
1  //=====[Implementations of public functions]=================================
2
3  void alarmInit()
4  {
5      alarmState = OFF;
6      sirenInit();
7      strobeLightInit();
8  }
9
10 void alarmUpdate()
11 {
12     if ( alarmState ) {
13
14         if ( codeMatchFrom(CODE_KEYPAD) ||
15              codeMatchFrom(CODE_PC_SERIAL) ) {
16             alarmDeactivate();
17         }
18
19         sirenUpdate( alarmStrobeTime() );
20         strobeLightUpdate( alarmStrobeTime() );
21
22     } else if ( gasDetectedRead() ||
23                 overTemperatureDetectedRead() ||
24                 intruderDetectedRead() )  {
25
26         alarmState = ON;
27         sirenStateWrite(ON);
28         strobeLightStateWrite(ON);
29     }
30 }
31
32 bool alarmStateRead()
33 {
34     return alarmState;
35 }
```

*Code 7.24 Details of the implementation of alarm.cpp (Part 2/3).*

```
1  //=====[Implementations of private functions]================================
2
3  static void alarmDeactivate()
4  {
5      alarmState = OFF;
6      sirenStateWrite(OFF);
7      strobeLightStateWrite(OFF);
8      intruderAlarmDeactivate();
9      fireAlarmDeactivate();
10 }
11
12 static int alarmStrobeTime()
13 {
14     if ( ( gasDetectedRead() || overTemperatureDetectedRead() ) &&
15            intruderDetectedRead() ) {
16         return STROBE_TIME_FIRE_AND_INTRUDER_ALARM;
17
18     } else if ( gasDetectedRead() || overTemperatureDetectedRead() ) {
19         return STROBE_TIME_FIRE_ALARM;
20
21     } else if ( intruderDetectedRead() ) {
22         return STROBE_TIME_INTRUDER_ALARM;
23
24     } else {
25         return 0;
26     }
27 }
```

*Code 7.25 Details of the implementation of alarm.cpp (Part 3/3).*

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _ALARM_H_
4  #define _ALARM_H_
5
6  //=====[Declarations (prototypes) of public functions]=========================
7
8  void alarmInit();
9  void alarmUpdate();
10 bool alarmStateRead();
11
12 //=====[#include guards - end]=================================================
13
14 #endif // _ALARM_H_
```

***Code 7.26 Details of the implementation of alarm.h.***

The new module *intruder_alarm* is shown in Code 7.27 and Code 7.28. The reader will note that this module is similar to the new implementation of the module *fire_alarm*, which is presented in Code 7.29, Code 7.30, and Code 7.31. The main differences between these two modules are that *fire_alarm* has two sensors (gas and temperature), *intruder_alarm* has only one sensor (PIR sensor), and *fire_alarm* has a test button, which after the modifications in the code is called *fireAlarmTestButton*.

In Code 7.27, the libraries used in the *intruder_alarm* module are included on lines 3 to 7. Two private global variables are declared and initialized on lines 11 and 12: *intruderDetected* and *intruderDetectorState*. In Code 7.27 and Code 7.30, the implementation of the public functions of the modules *fire_alarm* and *intruder_alarm* is shown. These two modules are described together to emphasize their similarities.

The functions that end with "Init" (lines 3 to 6 of Code 7.27 and lines 3 to 8 of Code 7.30) call the functions that initialize the sensors associated with the alarm. The functions that end with "Read" (lines 30 to 38 of Code 7.27 and lines 34 to 52 of Code 7.30) return the values of private variables. The functions that end with "Update" (lines 21 to 28 of Code 7.27 and lines 10 to 32 of Code 7.30) read the sensors and update the variables that end with "Detected" (used to activate the alarm) and "DetectorState". Finally, the functions that end with "Deactivate" (lines 40 to 43 of Code 7.27 and lines 54 to 58 of Code 7.30) assign OFF to the variables that end with "Detected" in order to turn off the alarm.

In Code 7.29, the libraries used in the new implementation of the *fire_alarm* module are included on lines 3 to 11. It is important to note that the libraries *code.h* and *matrix_keypad.h* are no longer needed after the modifications in this module. In the declaration of private #defines, the constants related to the strobe time of the siren are removed. Also, due to the module modifications, the private functions are removed in this new implementation.

The file headers of *intruder_alarm* and *fire_alarm* are shown in Code 7.28 and Code 7.31. It can be seen that the prototypes of the public functions are declared (lines 8 to 13 in Code 7.28 and lines 8 to 16 in Code 7.31).

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "intruder_alarm.h"
7  #include "motion_sensor.h"
8
9  //=====[Declaration and initialization of private global variables]============
10
11 static bool intruderDetected = OFF;
12 static bool intruderDetectorState = OFF;
13
14 //=====[Implementations of public functions]===================================
15
16 void intruderAlarmInit()
17 {
18     motionSensorInit();
19 }
20
21 void intruderAlarmUpdate()
22 {
23     intruderDetectorState = motionSensorRead();
24
25     if ( intruderDetectorState ) {
26         intruderDetected = ON;
27     }
28 }
29
30 bool intruderDetectorStateRead()
31 {
32     return intruderDetectorState;
33 }
34
35 bool intruderDetectedRead()
36 {
37     return intruderDetected;
38 }
39
40 void intruderAlarmDeactivate()
41 {
42     intruderDetected = OFF;
43 }
```

***Code 7.27 Details of the implementation of intruder_alarm.cpp.***

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _INTRUDER_ALARM_H_
4  #define _INTRUDER_ALARM_H_
5
6  //=====[Libraries]=============================================================
7
8  void intruderAlarmInit();
9  void intruderAlarmUpdate();
10 void intruderAlarmDeactivate();
11
12 bool intruderDetectorStateRead();
13 bool intruderDetectedRead();
14
15 //=====[#include guards - end]=================================================
16
17 #endif // _INTRUDER_ALARM_H_
```

***Code 7.28 Details of the implementation of intruder_alarm.h.***

```
//=====[Libraries]=============================================================

#include "mbed.h"
#include "arm_book_lib.h"

#include "fire_alarm.h"

#include "user_interface.h"
#include "date_and_time.h"
#include "temperature_sensor.h"
#include "gas_sensor.h"

//=====[Declaration of private defines]========================================

#define TEMPERATURE_C_LIMIT_ALARM              50.0

//=====[Declaration and initialization of public global objects]===============

DigitalIn fireAlarmTestButton(BUTTON1);

//=====[Declaration and initialization of private global variables]============

static bool gasDetected                  = OFF;
static bool overTemperatureDetected      = OFF;
static bool gasDetectorState             = OFF;
static bool overTemperatureDetectorState = OFF;
```

***Code 7.29 Details of the new implementation of fire_alarm.cpp (Part 1/2).***

```
//=====[Implementations of public functions]===================================

void fireAlarmInit()
{
    temperatureSensorInit();
    gasSensorInit();
    fireAlarmTestButton.mode(PullDown);
}

void fireAlarmUpdate()
{
    temperatureSensorUpdate();
    gasSensorUpdate();

    overTemperatureDetectorState = temperatureSensorReadCelsius() >
                                   TEMPERATURE_C_LIMIT_ALARM;

    if ( overTemperatureDetectorState ) {
        overTemperatureDetected = ON;
    }

    gasDetectorState = !gasSensorRead();

    if ( gasDetectorState ) {
        gasDetected = ON;
    }

    if( fireAlarmTestButton ) {
        overTemperatureDetected = ON;
        gasDetected = ON;
    }
}

bool gasDetectorStateRead()
{
    return gasDetectorState;
```

```
37  }
38
39  bool overTemperatureDetectorStateRead()
40  {
41      return overTemperatureDetectorState;
42  }
43
44  bool gasDetectedRead()
45  {
46      return gasDetected;
47  }
48
49  bool overTemperatureDetectedRead()
50  {
51      return overTemperatureDetected;
52  }
53
54  void fireAlarmDeactivate()
55  {
56      overTemperatureDetected = OFF;
57      gasDetected             = OFF;
58  }
```

*Code 7.30 Details of the new implementation of fire_alarm.cpp (Part 2/2).*

```
1   //=====[#include guards - begin]===============================================
2
3   #ifndef _FIRE_ALARM_H_
4   #define _FIRE_ALARM_H_
5
6   //=====[Declarations (prototypes) of public functions]=========================
7
8   void fireAlarmInit();
9   void fireAlarmUpdate();
10  void fireAlarmDeactivate();
11
12  bool gasDetectorStateRead();
13  bool gasDetectedRead();
14
15  bool overTemperatureDetectorStateRead();
16  bool overTemperatureDetectedRead();
17
18  //=====[#include guards - end]=================================================
19
20  #endif // _FIRE_ALARM_H_
```

*Code 7.31 Details of the new implementation of fire_alarm.h.*

Because there are two alarm sources, the display should show two different messages. In Code 7.32, the new implementation of the function *userInterfaceDisplayAlarmStateUpdate()* of the *user_interface* module is modified to account for this change. If the alarm is related to the gas detector or the temperature sensor (line 3), then the message is the same as in previous examples (lines 4 to 25). If the alarm is related to the intruder detector (line 26), then a new message is displayed: "Intruder Detected" (lines 27 to 40). If both alarm sources are active, the display will show the fire alarm message.

```
1    static void userInterfaceDisplayAlarmStateUpdate()
2    {
3        if ( ( gasDetectedRead() ) || ( overTemperatureDetectedRead() ) ) {
4            switch( displayFireAlarmGraphicSequence ) {
5            case 0:
6                displayBitmapWrite( GLCD_fire_alarm[0] );
7                displayFireAlarmGraphicSequence++;
8                break;
9            case 1:
10               displayBitmapWrite( GLCD_fire_alarm[1] );
11               displayFireAlarmGraphicSequence++;
12               break;
13           case 2:
14               displayBitmapWrite( GLCD_fire_alarm[2] );
15               displayFireAlarmGraphicSequence++;
16               break;
17           case 3:
18               displayBitmapWrite( GLCD_fire_alarm[3] );
19               displayFireAlarmGraphicSequence = 0;
20               break;
21           default:
22               displayBitmapWrite( GLCD_ClearScreen );
23               displayFireAlarmGraphicSequence = 0;
24               break;
25           }
26       } else if ( intruderDetectedRead() ) {
27           switch( displayIntruderAlarmGraphicSequence ) {
28           case 0:
29               displayBitmapWrite( GLCD_intruder_alarm );
30               displayIntruderAlarmGraphicSequence++;
31               break;
32           case 1:
33           default:
34               displayBitmapWrite( GLCD_ClearScreen );
35               displayIntruderAlarmGraphicSequence = 0;
36               break;
37           }
38       }
39   }
```

***Code 7.32 New implementation of the function userInterfaceDisplayAlarmStateUpdate().***

In Table 7.24, the sections in which lines were added to *user_interface.cpp* are shown. It can be seen that *alarm.h* and *intruder_alarm.h* have been included. A file *GLCD_intruder_alarm.h*, which contains the message that the display will show when an intruder is detected, is also included. The file *GLCD_clear_screen.h* contains the values for a blank screen that were previously included in *GLCD_intruder_alarm.h*.

The private global variable *displayIntruderAlarmGraphicSequence* is used to make the message appear on the display, and the variable name of *displayAlarmGraphicSequence* has been replaced by *displayFireAlarmGraphicSequence*, as shown in Table 7.25.

***Table 7.24 Sections in which lines were added to user_interface.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "alarm.h"`<br>`#include "intruder_alarm.h"`<br>`#include "GLCD_intruder_alarm.h"`<br>`#include "GLCD_clear_screen.h"` |
| Declaration and initialization of private global variables | `static int displayIntruderAlarmGraphicSequence = 0;` |

*Table 7.25 Variables that were renamed in user_interface.cpp.*

| Variable name in Example 7.3 | Variable name in Example 7.4 |
|---|---|
| displayAlarmGraphicSequence | displayFireAlarmGraphicSequence |

### Proposed Exercise

1. How can the code be changed to activate the intruder alarm *only* when the PIR sensor is active for more than four seconds?

### Answer to the Exercise

1. The function *intruderAlarmUpdate()* should be modified. In Code 7.33, the proposed implementation is shown. Because *intruderAlarmUpdate()* is called every 10 milliseconds, when *intruderDetectorCount* reaches a value of 400, the PIR sensor has been active for roughly 4 seconds.

```
 1  void intruderAlarmUpdate()
 2  {
 3      static int intruderDetectorCount = 0;
 4
 5      intruderDetectorState = motionSensorRead();
 6
 7      if ( intruderDetectorState ) {
 8          intruderDetectorCount++;
 9      } else {
10          intruderDetectorCount = 0;
11      }
12
13      if ( intruderDetectorCount > 400 ) {
14          intruderDetected = ON;
15      }
16  }
```

*Code 7.33 New implementation of intruderAlarmUpdate() that solves the proposed exercise.*

## 7.3 Under the Hood

### 7.3.1 Basic Principles of a Relay Module

In this chapter, a relay module was used to control a DC motor. Figure 7.12 shows a diagram of a typical circuit that is used in a relay module. As was mentioned in subsection 7.2.1, the aim of this circuit is to isolate the input (i.e., IN1 and 5 V) from the output (i.e., COM1, NC1, and NO1). It is also designed to use an output of the microcontroller to drive IN1. This implies that IN1 can take only three possible states: GND, 3.3 V, or unconnected, and is expected to drain or sink a current as small as possible from the microcontroller.

For these reasons, the optocoupler shown in Figure 7.12 is used, as well as the optional JDVCC power supply connection. When GND is applied to IN1, there will be a current established from VCC that will go through R1, the LED inside the optocoupler, and LED1. In this way, LED1 will turn on, and the LED inside the optocoupler will activate the transistor. This transistor will allow current to flow from

JVDCC to R2 through the base (B) of the T1 transistor. In this way, the T1 transistor is activated, which allows a current to flow between its collector terminal (C) and its emitter terminal (E). This current activates the coil of the relay, which causes its internal switch to connect COM1 and NO1.

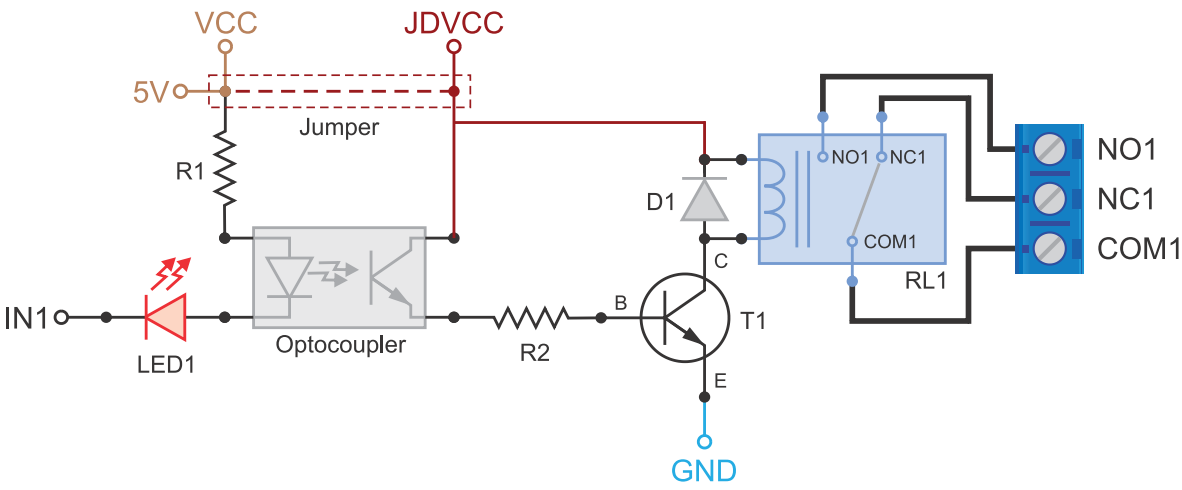



*Figure 7.12 Diagram of a typical circuit that is used in a relay module.*

Once IN1 is left unconnected, the optocoupler is unenergized, which causes the T1 transistor to turn off and the current through the coil of the RL1 relay to be cut off. This makes the internal switch of the relay move back via a spring in order to connect COM1 and NC1. It causes a high reverse voltage over the terminals of the coil. In order to prevent this voltage from damaging the circuit, the D1 diode is put in place, which prevents sparks from occurring.

**TIP:** In typical low-power applications, a jumper can be connected between VCC and JDVCC in order to avoid the need for an extra power supply. Note that in this case, the purpose of the optocoupler (i.e., to isolate the 5 V supply and the IN1 input from the stage composed by R2, T1, and D1) is voided. In any case, the relay RL1 isolates the output of the relay module (NO1, NC1, and COM1) from the rest of the circuit.

## Proposed Exercise

1. How can a relay module be used to control an AC motor?

## Answer to the Exercise

1. The proposed circuit connection is shown in Figure 7.13.

**WARNING:** The circuit shown in Figure 7.13 can be used with 110 or 220 V AC, but special care must be taken when working with voltages above 50 V.

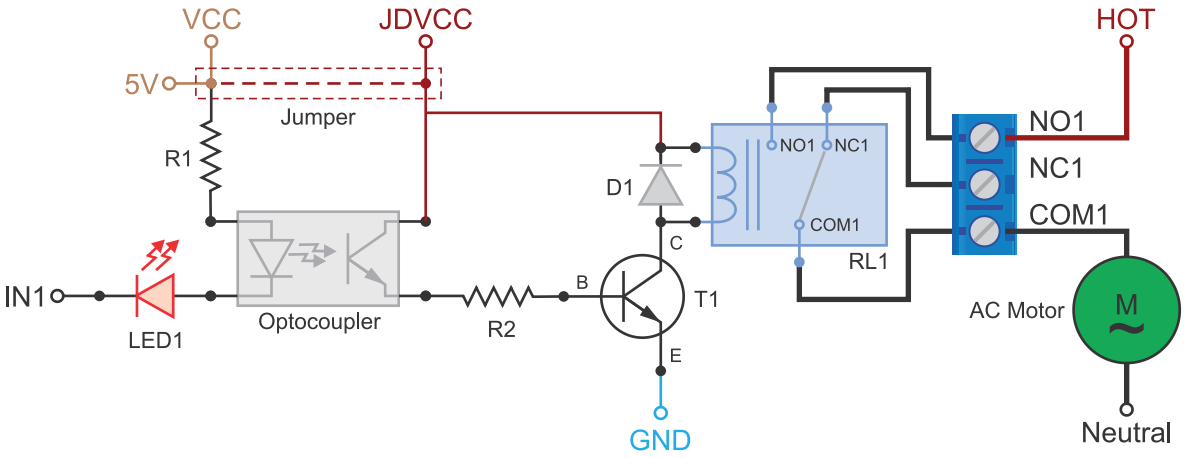



*Figure 7.13 Diagram of a typical circuit that is used to turn on and off an AC motor using a relay module.*

## 7.4 Case Study

### 7.4.1 Smart Street Lighting

In this chapter, a PIR sensor was connected to the NUCLEO board, and a DC motor was controlled using a relay module. This allowed a gate to be closed when intruders were detected. A smart street lighting system, built with Mbed and containing some similar features, can be found in [4]. In Figure 7.14, a diagram of the whole system is shown.

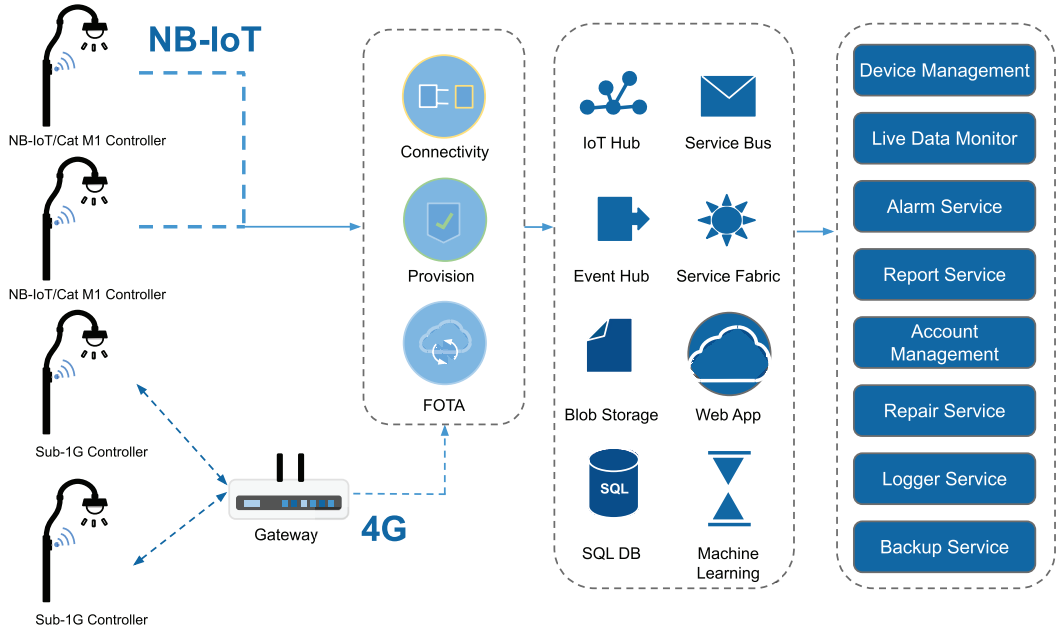



*Figure 7.14 A diagram of the Smart Street Lighting system.*

The smart street lighting system provides advanced dimming and on/off schedules that can be configured to optimize energy consumption during off-peak hours. In addition, using a built-in light sensor, the light is able to automatically switch off when daylight is detected. The system also provides fault detection and operator alerting via text or email, which allows for timely maintenance.

It might be noticed that the principle of operation of the light level sensor that is placed on each lamp of the smart street lighting system is very similar to the principle of operation of the PIR sensor that was used in this chapter. The lamps can be controlled using relay modules, such as the one used in this chapter, or solid-state relays, depending on the specific features of the lamp and the control system. Moreover, the smart street lighting system uses a set of tools to monitor and control the state of the lamps over the internet, having behavior and resources that are very similar to the tools used in this chapter.

**NOTE:** In the next chapter, a light level sensor will be included in the smart home system.

## Proposed Exercise

1. How can an AC lamp be turned on and off using a relay module?

## Answer to the Exercise

1. The proposed circuit connection is shown in Figure 7.15. It can be seen that it is very similar to the circuit used in Figure 7.13. Different AC-powered devices can be controlled using a relay module.

**WARNING:** The circuit shown in Figure 7.15 can be used with 110 or 220 V AC, but special care must be taken when working with voltages above 50 V.

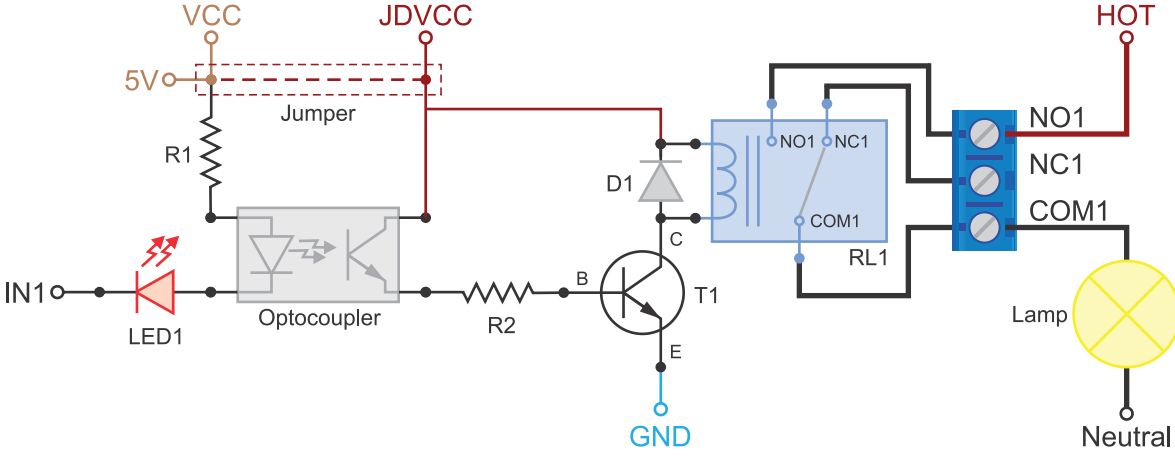



*Figure 7.15 Diagram of a typical circuit that is used to turn on and off an AC lamp using a relay module.*

## References

[1] "Toy/Hobby DC Motor Pinout Wiring, Specifications, Uses Guide and Datasheet". Accessed July 9, 2021.
https://components101.com/motors/toy-dc-motor

[2] "HC-SR501 PIR Sensor Working, Pinout & Datasheet". Accessed July 9, 2021.
https://components101.com/sensors/hc-sr501-pir-sensor

[3] "GitHub - armBookCodeExamples/Directory". Accessed July 9, 2021.
https://github.com/armBookCodeExamples/Directory/

[4] "Smart Street Lighting | Mbed". Accessed July 9, 2021.
https://os.mbed.com/built-with-mbed/smart-street-lighting/

# Chapter 8

Advanced Time Management, Pulse-Width Modulation, Negative Feedback Control, and Audio Message Playback

# 8.1 Roadmap

## 8.1.1 What You Will Learn

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

- Describe how to connect RGB LEDs to the NUCLEO board using digital output pins.
- Describe how to connect a light sensor to the NUCLEO board using an analog input pin.
- Develop programs to control the brightness of the RGB LED using pulse-width modulation.
- Summarize the fundamentals of timers that are integrated into a typical microcontroller.
- Implement time management on embedded systems using microcontroller timers.
- Generate an audio message using pulse-width modulation.
- Develop a simple negative feedback control system.

## 8.1.2 Review of Previous Chapters

In Chapter 3, the *delay()* function was used to vary the blinking rate of LED LD1 to indicate which element had triggered the fire alarm. In that chapter, the behavior was implemented first by means of a continuous delay (100 ms, 500 ms, or 1000 ms depending on the source of the alarm) and then by a delay built up from a set of 10 ms delays, in order to improve the responsiveness of the program. In this chapter, a new way of managing time intervals will be introduced, which will improve the responsiveness even more.

## 8.1.3 Contents of This Chapter

In this chapter, the use of integrated timers that are found in a typical microcontroller is explained. By means of these timers, time management will be implemented in order to control the behavior of the system. It will be shown that time control based on integrated timers provides a precise and responsive behavior in embedded system implementations.

*Pulse-width modulation* (PWM) is also introduced, by means of which the brightness level of an RGB LED can be controlled. An RGB (red, green, and blue) LED allows for the implementation of a wide variety of colors by appropriately combining the brightness level of each of the red, green, and blue elements of the RGB LED. It will be explained how to obtain an audio signal (an analog output voltage level that can be heard using headphones) by means of a PWM signal and an appropriate low pass filter.

Finally, the fundamentals of control theory are introduced through an example wherein light is sensed using a LDR (light-dependent resistor or photoresistor), and the brightness of an RGB LED is adjusted using PWM in order to achieve the brightness level, which is set using a potentiometer.

# 8.2 Analog Signal Generation with the NUCLEO Board

## 8.2.1 Connect an RGB LED, a Light Sensor, and an Audio Plug to the Smart Home System

In this chapter, the smart home system is provided with a decorative RGB light, a light sensor, and the capability to playback an audio message that says, "Welcome to the Smart Home System." A conceptual diagram of this setup is shown in Figure 8.1. The aim of this setup is to introduce the use of timers and the fundamentals of control theory.

The audio message and the signal to control the RGB light are generated using the PWM technique, as will be discussed in the examples below. The LDR sensor is used to measure the RGB light in order to be able to adjust its intensity to a value that is set using the potentiometer, which is now incorporated in the Gate control panel (Figure 8.1).

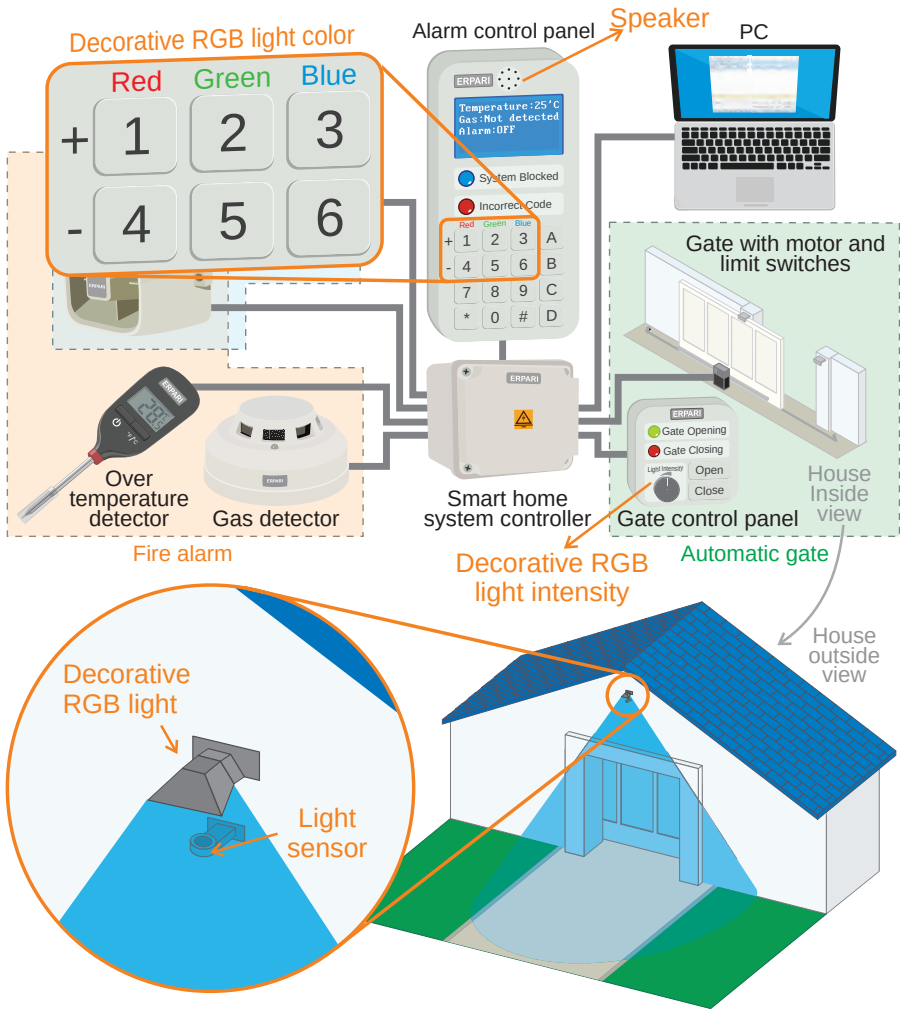



*Figure 8.1 The smart home system is now connected to an LCD display.*

Figure 8.2 shows the connections of the RGB LED [1], LDR light sensor [2], RC (resistor-capacitor) low pass filter [3], and the headphones that are connected to the smart home system in this chapter.

**NOTE:** In the implementation detailed in this chapter, the speaker is replaced by headphones, while the potentiometer that was connected in previous chapters is used to control the RGB intensity.

**NOTE:** The buzzer is now connected to the pin PC_9. In the following pages it will be explained why this is the case.

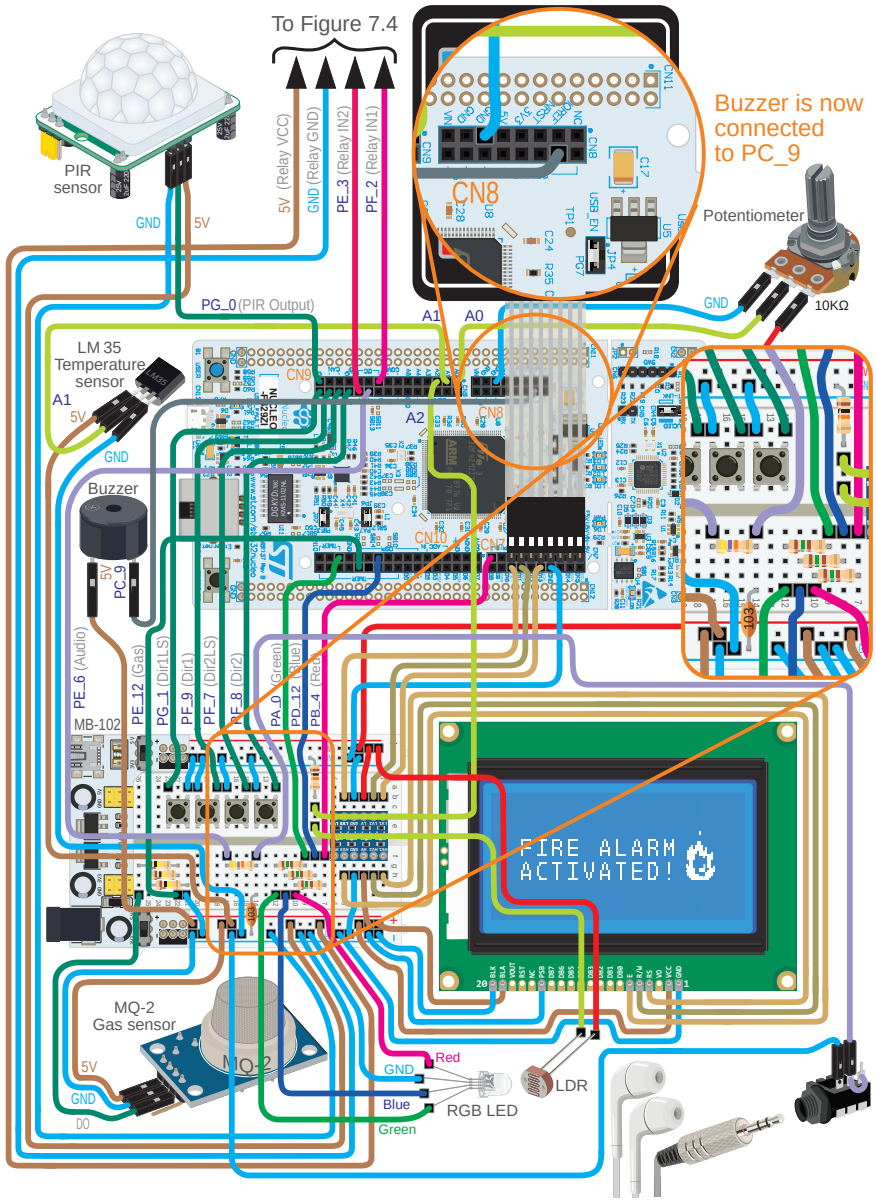



*Figure 8.2 The smart home system now has an RGB LED, a light sensor, and a circuit for audio playback.*

Figure 8.3 shows a diagram of the RGB LED connections and how to identify each of the pins of the RGB LED. The 150 Ω resistors are used to limit the current through each of the LEDs. The red, blue, and green LEDs are all built inside the package of the RGB LED.

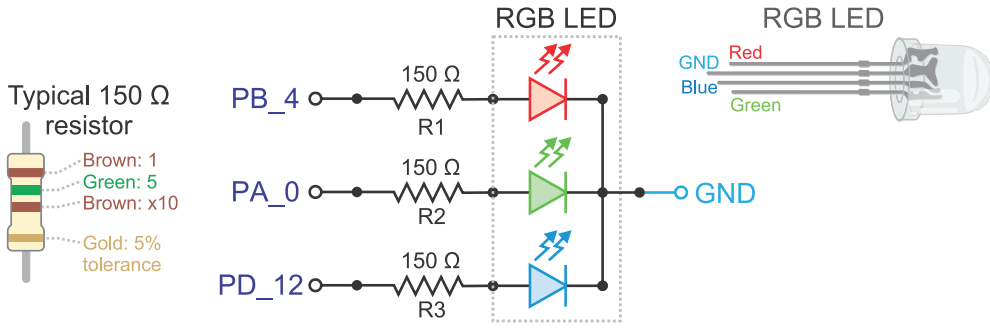



*Figure 8.3 Diagram of the connection of the RGB LED.*

Turning the different LEDs on and off obtains the set of colors shown in Figure 8.4. If the light intensity of each LED is modulated, a palette of millions of colors can be obtained. This behavior can be achieved by using the *pulse-width modulation* (PWM) technique, which is explained in subsection 8.2.2.

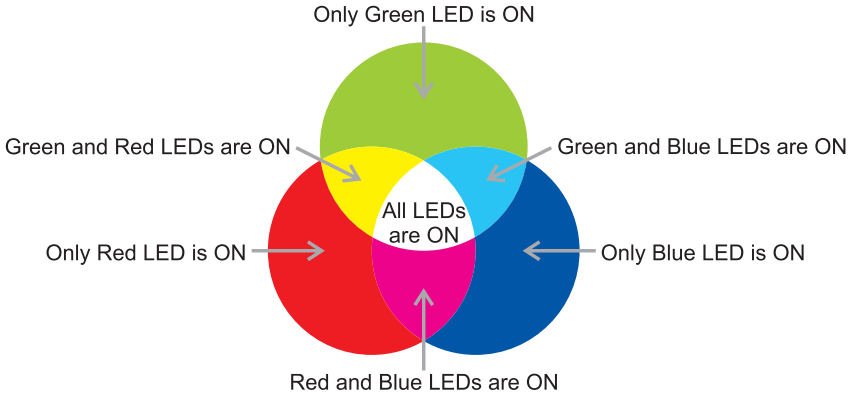



*Figure 8.4 Diagram of the light colors that result dependent on the LEDs that are turned on.*

To test if the RGB LED is working, the *.bin* file of the program "Subsection 8.2.1.a" should be downloaded from the URL available in [4] and loaded onto the NUCLEO board. The program should vary the color of the RGB LED through the palette shown in Figure 8.4 and all the intermediate colors as the button B1 USER is pressed. At the same time, the intensity of each of the LEDs of the RGB LED is printed on the serial terminal on a scale from 0 to 1.

**NOTE:** In Figure 8.5 it can be seen that PB_4, PA_0, and PD_12 are associated with PWM3/1, PWM2/1, and PWM4/1, respectively, which means PWM timer 3/channel 1, PWM timer 2/channel 1, and PWM timer 4/channel 1. By connecting each LED to a different PWM timer, it is possible to control their intensities independently, as will be shown in the examples below.

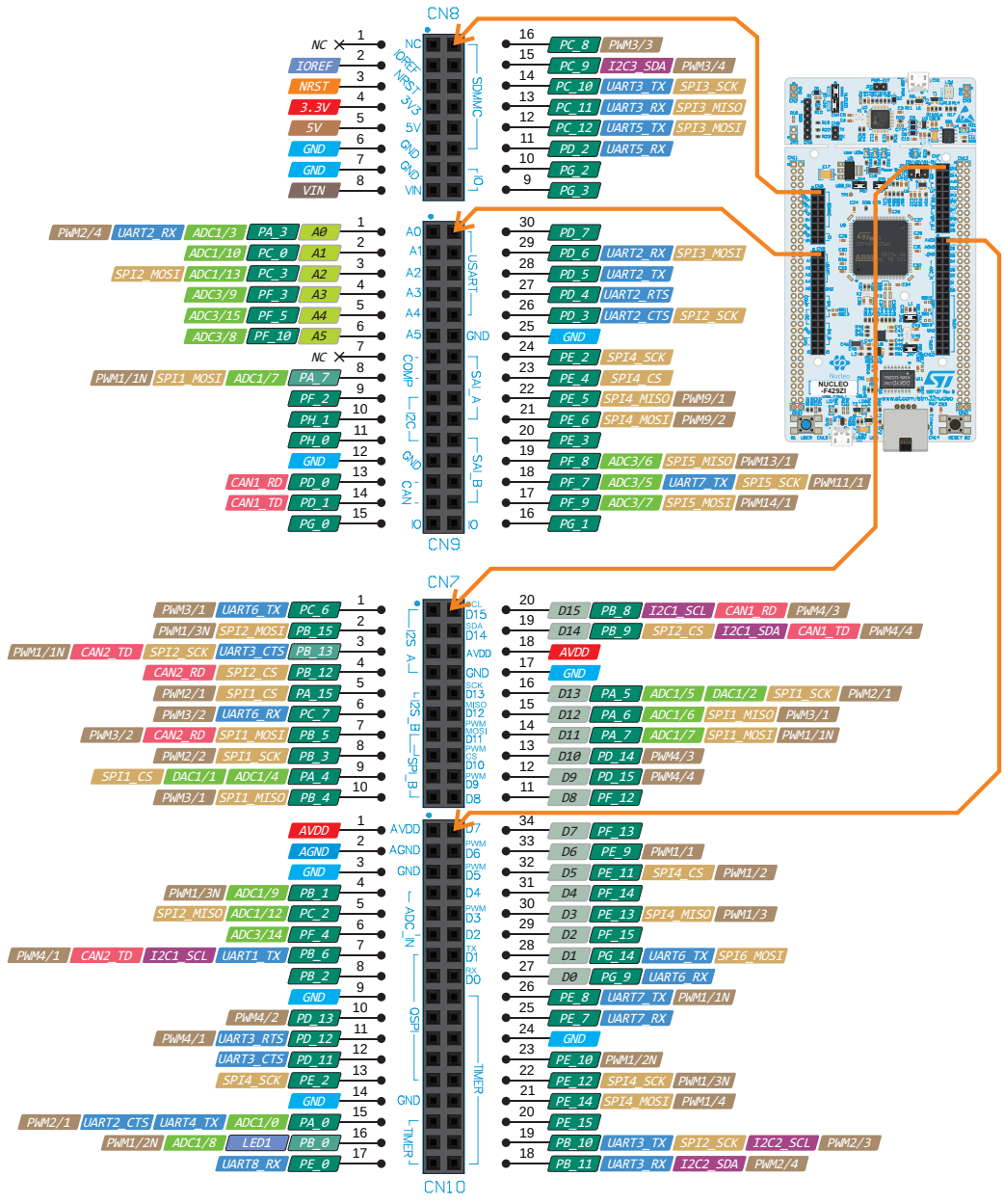



*Figure 8.5 ST Zio connectors of the NUCLEO-F429ZI board.*

Figure 8.6 shows the connections of a circuit used to sense light intensity by means of an LDR. This component varies its resistance depending on the amount of light sensed. In this way, the voltage at the analog input A2 varies as the light intensity over the LDR varies.

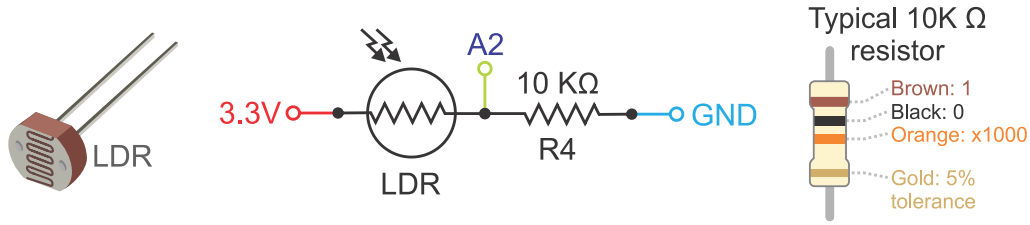



*Figure 8.6 Diagram of the connection of the LDR.*

**TIP:** The LDR pins are identical, so the LDR can be connected either way around.

Figure 8.7 shows the RC circuit that is used to obtain the analog audio signal from a digital signal at pin PE_6 by using PWM (associated to PWM9/2, as can be seen in Figure 8.5). The resistor R5 and the capacitor C1 make up a *low pass filter*. Subsection 8.2.2 explains how this setup works.

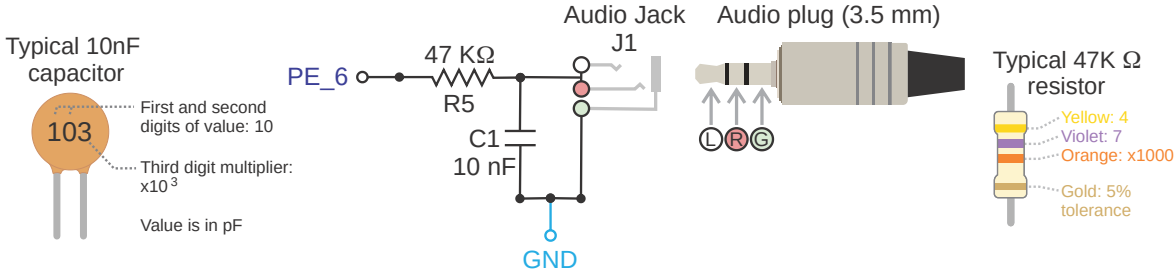



*Figure 8.7 Diagram of the connection of the low pass filter and the audio jack.*

**TIP:** If the components shown in Figure 8.7 are not available, then the values of R5 and C1 can be modified without noticeably degrading the audio signal. For example, a 39 KΩ or 56 KΩ resistor can be used for R5, while a 4.7 nF or 22 nF capacitor can be used for C1. The power dissipation of the resistors and the capacitor voltage rate are not relevant because of the low current and low voltage of this application.

To test if the LDR and the low pass filter are working properly, the *.bin* file of the program "Subsection 8.2.1.b" should be downloaded from the URL available in [4] and loaded onto the NUCLEO board. Plug headphones into the audio jack, and an audio message "Welcome to the smart home system" should be heard just after the NUCLEO board powers up. To listen to the message again, reset the NUCLEO board by means of the Reset B2 button.

**NOTE:** The audio level is not very loud. To get a louder audio level, an audio amplifier should be used.

**TIP:** Ignore all the other elements of the setup during the proposed test (Alarm LED, display, etc.).

Also, as the knob of the potentiometer is turned, the *set point* of the brightness level is modified. The set point is indicated by a message on the serial terminal where the reading of the potentiometer is shown on a scale from 0 to 1 (it is indicated as "SP", as shown in Figure 8.8). The program uses the LDR to sense the light intensity. The reading of the LDR is shown on a scale from 0 to 1 (Figure 8.8). When the light that impinges on the LDR is blocked, the reading shown on the serial terminal should diminish.

**NOTE:** Example 8.5 will explain in detail the concept of *set point* and how the brightness of the RGB LED can be controlled by means of the readings of the potentiometer and the LDR, as in the conceptual diagram shown on Figure 8.1.

```
SP: 0.7369 | LDR: 0.7046
SP: 0.7374 | LDR: 0.7151
SP: 0.7361 | LDR: 0.7232
SP: 0.7352 | LDR: 0.7342
SP: 0.7369 | LDR: 0.7384
```

***Figure 8.8 Information shown on the serial terminal when program "Subsection 8.2.1.b" is running.***

It was mentioned in Example 6.5 that the function *displayBitmapWrite()* requires sending hundreds of bytes of data to the graphical display, which may interfere with the time management of the strobe light and the siren implemented by means of LD1 and the buzzer, respectively. Due to this, when gas and over temperature are detected, the time the siren and the strobe light are on and off is not always 100 ms as expected. This problem is addressed in this chapter by means of using PWM to control the siren and the strobe light.

In order to use the PWM technique to control the siren and the strobe light, it should be noted in Figure 8.5 that PB_0 is connected to LED1 of the NUCLEO board (the one labeled as LD1 that is used to mock the strobe light). It should also be noted that PB_0 is associated with PWM1/2N. This means that the strobe light (LD1) can be controlled using channel 2 of PWM timer 1 (the N stands for inverted, which means that the true state is implemented with 0 V).

In Figure 8.5, it can be seen that the pin PE_10 that was used in previous chapters to activate the buzzer (which simulates the siren) is also related to PWM1/2N. In this way, if PWM timer 1 is used to control LD1 and the buzzer, then both will be driven by the same signal. In this particular case this is a problem, because LD1 is turned off by placing a low state on this LED, while the buzzer is turned off by placing a high state on the corresponding pin. In other words, it will not be possible to turn off both LD1 and the buzzer at the same time using the PWM1/2N signal to control both elements. Therefore, a different PWM timer should be used to control the siren, because the connection of LD1 cannot be modified.

In Table 8.1, it can be seen that all of the PWM timers available in Figure 8.5 are already occupied. In order to solve this problem, *alternative peripheral instances* must be used to assign a PWM timer to the buzzer.

***Table 8.1 Summary of the PWM timers that are already in use or occupied by other functionalities.***

| PWM timer | Used by | Pin | PWM used |
|---|---|---|---|
| 1 | LD1 | PB_0 | PWM1/2N |
| 2 | RGB LED (G) | PA_0 | PWM2/1 |
| 3 | RGB LED (R) | PB_4 | PWM3/1 |
| 4 | RGB LED (B) | PD_12 | PWM4/1 |
| 5 | Not available in Figure 8.5 | – | – |
| 6 | Not available in Figure 8.5 | – | – |
| 7 | Not available in Figure 8.5 | – | – |
| 8 | Not available in Figure 8.5 | – | – |
| 9 | Audio playback | PE_6 | PWM9/2 |
| 10 | Not available in Figure 8.5 | – | – |
| 11 | Button Dir2LS | PF_7 | – |
| 12 | Not available in Figure 8.5 | – | – |
| 13 | Button Dir2 | PF_8 | – |
| 14 | Button Dir1 | PF_9 | – |

Alternative peripheral instances are explained in [5]. The idea is that all pins are defined in the *PinNames.h* file of each board. For example, the pins corresponding to the NUCLEO-F429ZI board can be found in [6]. Code 8.1 shows the first part of this file. It can be seen that alternative possibilities that use other hardware peripheral instances are mentioned. In particular, it is highlighted that these alternative possibilities can be used as any other "normal" pin and that these pins are not displayed on the board pinout image shown in Figure 8.5.

```
1  //============================================================================
2  // Notes
3  //
4  // - The pins mentioned Px_y_ALTz are alternative possibilities which use other
5  //   HW peripheral instances. You can use them the same way as any other "normal"
6  //   pin (i.e. PwmOut pwm(PA_7_ALT0);). These pins are not displayed on the board
7  //   pinout image on mbed.org.
8  //
9  // - The pins which are connected to other components present on the board have
10 //   the comment "Connected to xxx". The pin function may not work properly in this
11 //   case. These pins may not be displayed on the board pinout image on mbed.org.
12 //   Please read the board reference manual and schematic for more information.
13 //
14 // - Warning: pins connected to the default STDIO_UART_TX and STDIO_UART_RX pins are
15 //   commented
16 //   See https://os.mbed.com/teams/ST/wiki/STDIO for more information.
17 //
18 //============================================================================
```

***Code 8.1 Notes on the PinNames.h file of the NUCLEO-F429ZI board.***

Code 8.2 and Code 8.3 show the section of *PinNames.h* regarding PWM pins. For example, on line 4 of Code 8.2, it can be seen that PA_0 is related to PWM2 and channel 1 (channel 1 is indicated by the 1 that is the penultimate value of line 4), which is used to control the green LED of the RGB LED (see Figure 8.3). This "normal" functionality of PA_0 is shown in Figure 8.5. In line 1 of Code 8.3, it can be seen that PB_0 is related to PWM1 and channel 2, with inverted behavior (ultimate value 1 of line 1). Inverted behavior means that a logic true is set by a 0 V value, as was explained above. This functionality is also shown on Figure 8.5.

```
1  //*** PWM ***
2
3  MSTD_CONSTEXPR_OBJ_11 PinMap PinMap_PWM[] = {
4      {PA_0,       PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 1, 0)},
5      {PA_1,       PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 2, 0)},
6      {PA_2,       PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 3, 0)},
7      {PA_2_ALT0,  PWM_9,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM9, 1, 0)},
8      {PA_3,       PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 4, 0)},
9      {PA_3_ALT0,  PWM_9,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM9, 2, 0)},
10     {PA_5,       PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 1, 0)},
11     {PA_5_ALT0,  PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 1, 1)},
12     {PA_6,       PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 1, 0)},
13     {PA_6_ALT0,  PWM_13, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF9_TIM13, 1, 0)},
14     {PA_7,       PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 1, 1)},
15     {PA_7_ALT0,  PWM_3,   STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 2, 0)},
16     {PA_7_ALT1,  PWM_8,   STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 1, 1)},
17     {PA_7_ALT2,  PWM_14, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF9_TIM14, 1, 0)},
18     {PA_8,       PWM_1,   STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 1, 0)},
19     {PA_9,       PWM_1,   STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 2, 0)},
20     {PA_10,      PWM_1,   STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 3, 0)},
21     {PA_11,      PWM_1,   STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 4, 0)},
22     {PA_15,      PWM_2,   STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 1, 0)},
```

***Code 8.2 Notes on the PinNames.h file of the NUCLEO-F429ZI board.***

It can be seen that PWM functionality is available as alternative functionality in more pins indicated as Px_y_ALTz, which are not shown in Figure 8.5. For example, PWM timer 8 is available in PA_5_ALT0 (line 11 of Code 8.2). However, PA_5 is already occupied by the SPI1_SCK functionality used by the graphical

LCD display and, therefore, PA_5 cannot be used without interfering with the display. PWM timer 8 is also available in PA_7_ALT1 (line 20 of Code 8.2), but PA_7 is used by SPI1_MOSI, also used by the graphical LCD display. PWM timer 8 is also available in PB_0_ALT1, PB_1_ALT1, PB_14_ALT0, PB_15_ALT0, PC_6_ALT0, PC_7_ALT0, PC_8_ALT0, and PC_9_ALT0 (Code 8.3). Some of these pins are not being used, as, for example, PC_9. For this reason, the buzzer is connected to PC_9 (see Figure 8.2). In the program code in the examples, a PWM object will be created and assigned to PC_9_ALT0. In this way, this pin will be associated with PWM8/4 (PWM timer 8/channel 4), as can be seen in line 32 of Code 8.3.

```
 1      {PB_0,       PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 4, 1)},
 2      {PB_0_ALT0,  PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 3, 0)},
 3      {PB_0_ALT1,  PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 2, 1)},
 4      {PB_1,       PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 3, 1)},
 5      {PB_1_ALT0,  PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 4, 0)},
 6      {PB_1_ALT1,  PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 3, 1)},
 7      {PB_3,       PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 2, 0)},
 8      {PB_4,       PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 1, 0)},
 9      {PB_5,       PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 2, 0)},
10      {PB_6,       PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 1, 0)},
11      {PB_7,       PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 2, 0)},
12      {PB_8,       PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 3, 0)},
13      {PB_8_ALT0,  PWM_10, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM10, 1, 0)},
14      {PB_9,       PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 4, 0)},
15      {PB_9_ALT0,  PWM_11, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM11, 1, 0)},
16      {PB_10,      PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 3, 0)},
17      {PB_11,      PWM_2,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM2, 4, 0)},
18      {PB_13,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 1, 1)},
19      {PB_14,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 2, 1)},
20      {PB_14_ALT0, PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 2, 1)},
21      {PB_14_ALT1, PWM_12, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF9_TIM12, 1, 0)},
22      {PB_15,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 3, 1)},
23      {PB_15_ALT0, PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 3, 1)},
24      {PB_15_ALT1, PWM_12, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF9_TIM12, 2, 0)},
25      {PC_6,       PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 1, 0)},
26      {PC_6_ALT0,  PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 1, 0)},
27      {PC_7,       PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 2, 0)},
28      {PC_7_ALT0,  PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 2, 0)},
29      {PC_8,       PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 3, 0)},
30      {PC_8_ALT0,  PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 3, 0)},
31      {PC_9,       PWM_3,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM3, 4, 0)},
32      {PC_9_ALT0,  PWM_8,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM8, 4, 0)},
33      {PD_12,      PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 1, 0)},
34      {PD_13,      PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 2, 0)},
35      {PD_14,      PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 3, 0)},
36      {PD_15,      PWM_4,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF2_TIM4, 4, 0)},
37      {PE_5,       PWM_9,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM9, 1, 0)},
38      {PE_6,       PWM_9,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM9, 2, 0)},
39      {PE_8,       PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 1, 1)},
40      {PE_9,       PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 1, 0)},
41      {PE_10,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 2, 1)},
42      {PE_11,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 2, 0)},
42      {PE_12,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 3, 1)},
43      {PE_13,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 3, 0)},
44      {PE_14,      PWM_1,  STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF1_TIM1, 4, 0)},
45      {PF_6,       PWM_10, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM10, 1, 0)},
46      {PF_7,       PWM_11, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF3_TIM11, 1, 0)},
47      {PF_8,       PWM_13, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF9_TIM13, 1, 0)},
48      {PF_9,       PWM_14, STM_PIN_DATA_EXT(STM_MODE_AF_PP, GPIO_NOPULL, GPIO_AF9_TIM14, 1, 0)},
49      {NC, NC, 0}
50  };
```

***Code 8.3 Notes on the PinNames.h file of the NUCLEO-F429ZI board.***

**TIP:** To augment the strobe light functionality, an optional external high-brightness LED can be connected by means of a transistor, as shown in Figure 8.9. This high-brightness LED cannot be connected directly to a NUCLEO board pin, as the NUCLEO board pin cannot provide the current that the LED needs to turn on at its maximum brightness. For this reason, transistor T1 is used, which acts as a controlled switch allowing the higher current to flow through the high-brightness LED when the NUCLEO board pin is set to 3.3 V, and blocking the current through the high-brightness LED when the NUCLEO board pin is set to 0 V.

Note that this circuit is activated with high state and deactivated with low state, the same logic as the NUCLEO board LD1 LED, which is internally connected to the PB_0 pin. Therefore, if the NUCLEO board pin of this circuit is connected to the PB_0 pin of the NUCLEO board, then the high-brightness LED will turn on and off concurrently with the NUCLEO board LD1 LED without needing to modify the code and with a much higher brightness.

The resistor RB is used to limit the current sourced from the NUCLEO board pin. The resistor RC is used to limit the current flow through the high-brightness LED. The brightness of the high-brightness LED can be augmented by reducing the value of RC. This should be done with caution, however, because if the value used is too low, then the high-brightness LED or the transistor T1 (or both) could be damaged.

The maximum current that the BC548C transistor can handle is 100 mA. If a high-brightness LED that demands a higher current is used, then a transistor with a higher maximum current should be used.

If a brighter light is needed for a specific application, the circuit introduced in Figure 7.12 can be used, with the proviso that the warning detailed in Chapter 7 is heeded. It is not recommended to switch a relay more than once a second or its lifetime can be severely reduced.

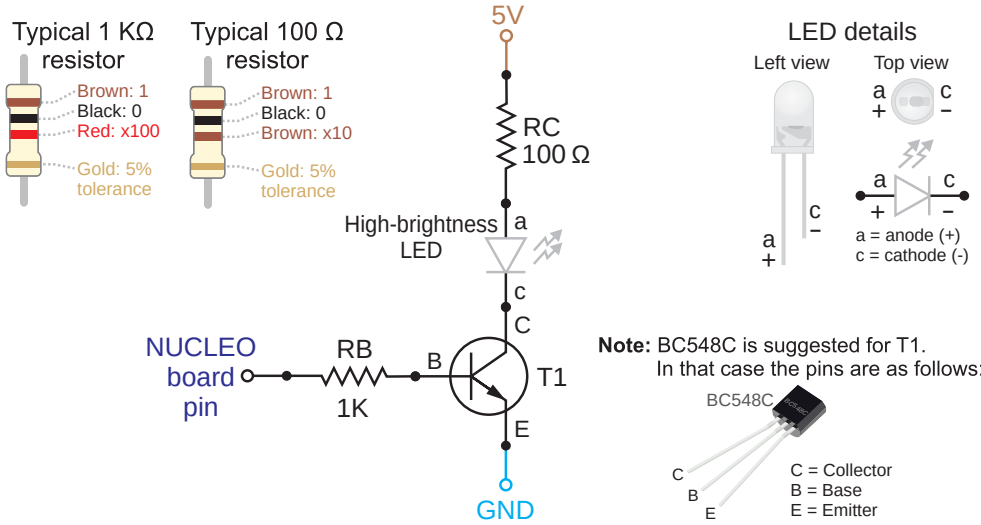



*Figure 8.9 Connection of a high-brightness LED to a NUCLEO board pin (optional).*

### 8.2.2 Fundamentals of Timers, Pulse-Width Modulation, and Audio Message Playback

This subsection explains how to use the timers of a microcontroller to generate periodic events. Also, based on these periodic events, it is explained how to use the *pulse-width modulation (PWM)* technique to control the brightness of an LED and to generate audio signals.

Timers are the basis of time management in microcontrollers. The *delay()* function used in Chapter 3 is based on the timers of the microcontroller. In that case, no other code was able to be executed when *delay()* was taking place. This led to the code having low responsiveness, and it was concluded that in order to overcome this issue, it was better to implement a long delay by means of many consecutive short delays. This allowed some other tasks to be attended to in the gaps between these short delays. However, this solution results in inaccurate durations of the delays and many calls to the delay functions.

Implementing the delays with a timer linked to an *interrupt service routine* means that the processor can do other things during the counting. This leads to more accurate and repeatable delays.

In Figure 8.10, a basic diagram of a timer is shown. On the left, it can be seen that the clock signal can be internal or external, which is configured by writing into special locations of the microcontroller internal memory known as *registers* that control the multiplexor selection. Then, there is a "*Down Counter*" module that decrements its value each time there is a pulse of the selected clock. Once this counter reaches zero, an interrupt can be triggered, depending on the control configuration. The timer can also be configured to automatically load the "Initial value" and restart the count from there each time it reaches zero.

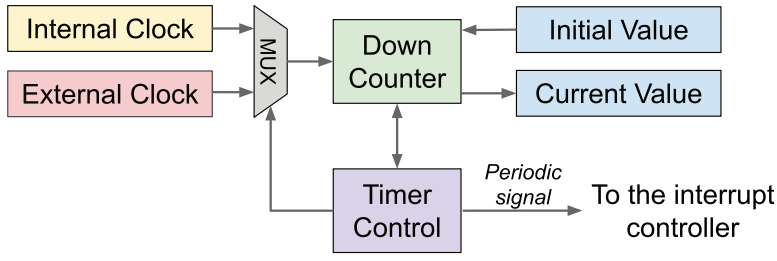



***Figure 8.10 Simplified diagram of a timer.***

In this way, the periodic signal can be configured by changing the "Initial value" register. The current value of the timer count can be read any time by means of reading the "Current value" register.

Microcontrollers' timers can be used to generate periodic signals, known as *tickers*, as shown in Figure 8.11. The period of the tickers can be adjusted by writing into special registers. There are also registers that are used to enable or disable the tickers, as well as to enable or disable *interrupts* related to the tickers' signals. When writing programs in the C/C++ language, the compiler takes care of all the details regarding the registers.

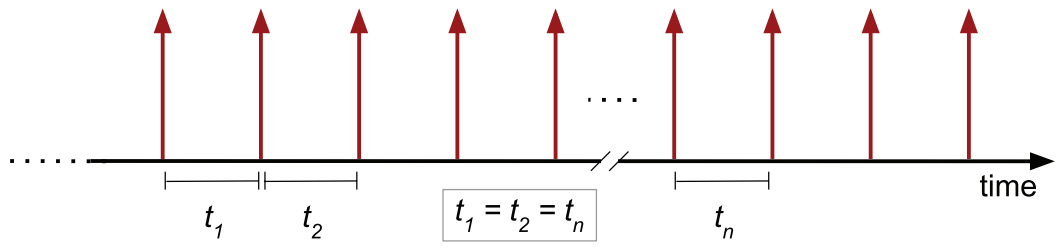



*Figure 8.11 Periodic signal generated by the built-in timer of a microcontroller.*

Most timers inside microcontrollers are also equipped with special hardware that allows easy implementation of PWM. To see what PWM is and how it can be used to dim the brightness of an LED, download the *.bin* file of the program "Subsection 8.2.2" from [4] and load it onto the NUCLEO board. The program will make LEDs LD1, LD2, and LD3 turn on and off according to the time intervals shown in Table 8.2. When pressing button *Dir1*, connected to pin PF_9, it will be clear to see when the LEDs are on and off. In the case of pressing button *Dir2*, connected to pin PF_8, it will still be appreciated when they are on and off. However, in the case of pressing button *Dir2LS*, connected to pin PF_7, it will not be appreciated when the LEDs are on or off. Instead, the LEDs will appear to shine just a little bit (exactly 20% of full brightness, given that they are on 2 ms and off 8 ms; 2 ms / (2 ms + 8ms) = 0.2).

*Table 8.2 On time and off time of the LEDs used in the program "Subsection 8.2.2".*

| Button | On time | Off time |
|---|---|---|
| Button Dir1 connected to pin PF_9 | 200 milliseconds | 800 milliseconds |
| Button Dir2 connected to pin PF_8 | 20 milliseconds | 80 milliseconds |
| Button Dir2LS connected to pin PF_7 | 2 milliseconds | 8 milliseconds |

Figure 8.12 shows how the brightness of an LED varies as the *duty cycle* of the signal varies.

In frame (d) of Figure 8.13, the waveform of the message "Welcome to the Smart Home System" is shown. This message lasts for three seconds, and the amplitude of the analog signal is sampled regularly (every 125 µs, or 8000 samples per second). Each sample is quantized to the nearest value within a range of digital steps with 8-bit resolution. This is known as *pulse-code modulation* (PCM) and is the standard form of digital audio in computers, compact discs, digital telephony, and other digital audio applications.

In this way, an array of 24,000 values (samples) named *welcomeMessageData* is obtained. In the (a) frame of Figure 8.13, four consecutive values of this array are shown: [n], [n+1], [n+2], and [n+3]. In this example these values are 191, 127, 64, and 120, which corresponds to 75%, 50%, 25%, and 47% of the maximum value obtainable using the 8 bits resolution (255).

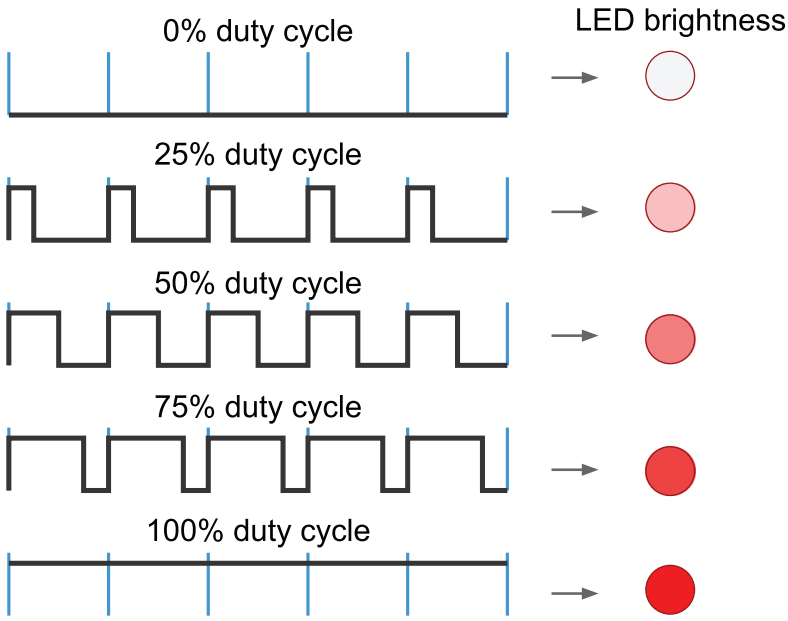



***Figure 8.12 Example of the variation of LED brightness by the pulse width modulation technique.***

Frame (c) of Figure 8.13 shows how the duty cycle of a PWM digital signal is *modulated* using the data of *welcomeMessageData*, normalized to 255. The period of the signal is 25 µs, so every 5 periods (125 µs), its duty cycle is adjusted according to a new value read from the *welcomeMessageData* array.

Frame (b) of Figure 8.13 shows the analog output signal that is obtained when the signal shown in (b) is filtered by the low pass filter introduced in Figure 8.7. It can be seen that in this way a 500 µs piece out of the three seconds' length audio signal is obtained.

By means of repeating this process for the whole set of values of the array *welcomeMessageData*, the message "Welcome to the smart home system" is obtained.

**NOTE:** The value of $V_{Max}$ (in frame (c) of Figure 8.13) varies depending on several factors.

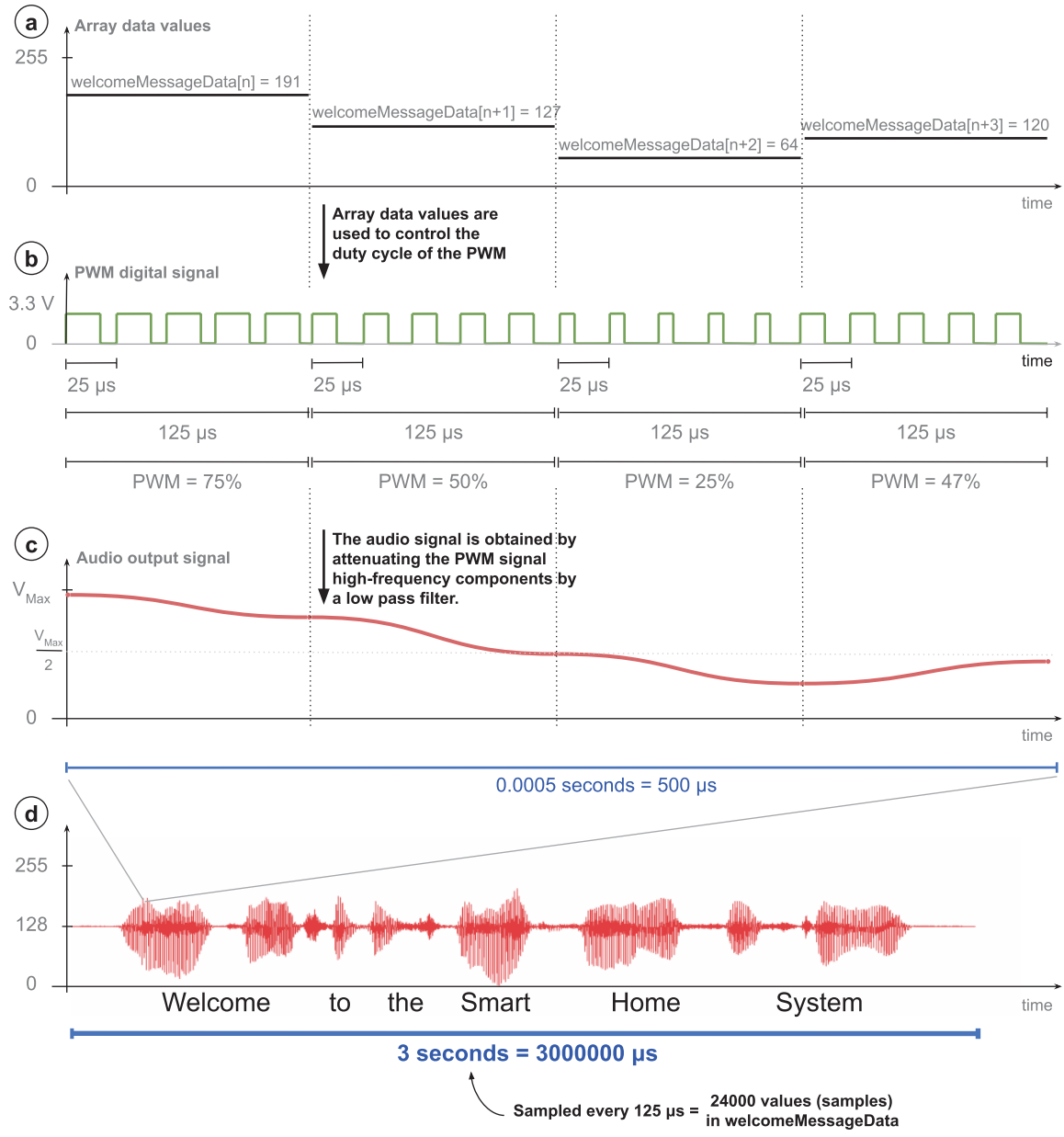



*Figure 8.13 Detail on how the "Welcome to the smart home system" message is generated using PWM.*

## Example 8.1: Implementation of PWM to Control the Brightness of an RGB LED

### Objective

Introduce an implementation of pulse-width modulation (PWM).

### Summary of the Expected Behavior

The brightness of the RGB LED should change as the knob of the potentiometer is rotated.

### Test the Proposed Solution on the Board

Import the project "Example 8.1" using the URL available in [4], build the project, and drag the *.bin* file onto the NUCLEO board. Rotate the knob of the potentiometer gradually and observe how the RGB LED turns on in a white color, and its brightness changes as the knob of the potentiometer is rotated.

### Discussion of the Proposed Solution

The proposed solution is based on three new modules: *light_system*, responsible for updating the duty cycle of the PWM; *bright_control*, responsible for generating the PWM signal with a given duty cycle; and *light_level_control*, responsible for the reading of the potentiometer. In this way, if the element that controls the light level is changed (for instance, using a set of buttons instead of the potentiometer), only the *light_level_control* module needs to be changed, and there is no need to update the *light_system* module.

### Implementation of the Proposed Solution

The initialization of the *light_system* module is done at the beginning of the program by means of a call to the function *lightSystemInit()* from *smartHomeSystemInit()*, as can be seen on line 10 of Code 8.4. The function *lightSystemUpdate()* is included in *smartHomeSystemUpdate()* (line 21) to periodically update the duty cycle of the PWM signal. In order to implement these calls, the library *light_system.h* is included in *smart_home_system.cpp*, as can be seen in Table 8.3.

```
 1  void smartHomeSystemInit()
 2  {
 3      userInterfaceInit();
 4      alarmInit();
 5      fireAlarmInit();
 6      intruderAlarmInit();
 7      pcSerialComInit();
 8      motorControlInit();
 9      gateInit();
10      lightSystemInit();
11  }
12
13  void smartHomeSystemUpdate()
14  {
15      userInterfaceUpdate();
16      fireAlarmUpdate();
17      intruderAlarmUpdate();
18      alarmUpdate();
19      eventLogUpdate();
20      pcSerialComUpdate();
21      lightSystemUpdate();
22      delay(SYSTEM_TIME_INCREMENT_MS);
23  }
```

***Code 8.4 New implementation of the functions smartHomeSystemInit and smartHomeSystemUpdate.***

*Table 8.3 Sections in which lines were added to smart_home_system.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "light_system.h"` |

The assignment of the *sirenPin* was modified for the reasons that were discussed on section 8.2.1. This change is summarized on Table 8.4.

*Table 8.4 Lines that were modified in siren.cpp.*

| Section | Previous line | New line |
|---|---|---|
| Declaration and initialization of public global objects | `DigitalOut sirenPin(PE_10);` | `DigitalOut sirenPin(PC_9);` |

The new module *light_system* is shown in Code 8.5 and Code 8.6. The libraries that are included are shown from lines 3 to 7 of Code 8.5. On line 11, a private global variable name *dutyCycle* is declared and initialized. From lines 15 to 18, the implementation of the function *lightSystemInit()* is shown. This calls the function that initializes the module *bright_control* (line 17). Finally, the function *lightSystemUpdate()*, which reads the value of the potentiometer (line 22) and updates the duty cycle for each of the three LEDs (lines 24 to 26), is shown.

```
1  //=====[Libraries]=============================================================
2
3  #include "arm_book_lib.h"
4
5  #include "light_system.h"
6  #include "bright_control.h"
7  #include "light_level_control.h"
8
9  //=====[Declaration and initialization of private global variables]============
10
11 static float dutyCycle = 0.5;
12
13 //=====[Implementations of public functions]===================================
14
15 void lightSystemInit()
16 {
17     brightControlInit();
18 }
19
20 void lightSystemUpdate()
21 {
22     dutyCycle = lightLevelControlRead();
23
24     setDutyCycle( RGB_LED_RED, dutyCycle );
25     setDutyCycle( RGB_LED_GREEN, dutyCycle );
26     setDutyCycle( RGB_LED_BLUE, dutyCycle );
27 }
```

*Code 8.5 Details of the implementation of light_system.cpp.*

In Code 8.6, the implementation of *light_system.h* is shown. It can be seen that the data type *lightSystem_t* and the prototypes of the public functions are declared on lines 8 to 12 and 16 to 17, respectively.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _LIGHT_SYSTEM_H_
4  #define _LIGHT_SYSTEM_H_
5
6  //=====[Declaration of public data types]======================================
7
8  typedef enum {
9      RGB_LED_RED,
10     RGB_LED_GREEN,
11     RGB_LED_BLUE,
12 } lightSystem_t;
13
14 //=====[Declarations (prototypes) of public functions]=========================
15
16 void lightSystemInit();
17 void lightSystemUpdate();
18
19 //=====[#include guards - end]=================================================
20
21 #endif // _LIGHT_SYSTEM_H_
```

***Code 8.6 Details of the implementation of light_system.h.***

The implementation of the new module *light_level_control* is presented in Code 8.7. The libraries that are included are shown from lines 3 to 6. On line 10, the public global object *potentiometer*, related to the potentiometer connected to the analog input 2 (A0), is declared. The reader may notice that the public functions implementation (from lines 14 to 21) is quite simple. In this case, no average is applied to the analog signal samples as the voltage is more stable than when reading sensors; this was explained in Chapter 3. In this case, a small variation in the reading due to noise has no significant impact on the system behavior. The functions *lightLevelControlInit()* (line 14) and *lightLevelControlUpdate()* (line 16) are kept to maintain the same structure used as, for example, in the *temperature_sensor* module.

```
1  //=====[Libraries]=============================================================
2
3  #include "arm_book_lib.h"
4
5  #include "smart_home_system.h"
6  #include "light_level_control.h"
7
8  //=====[Declaration and initialization of public global objects]===============
9
10 AnalogIn potentiometer(A0);
11
12 //=====[Implementations of public functions]===================================
13
14 void lightLevelControlInit() { }
15
16 void lightLevelControlUpdate() { }
17
18 float lightLevelControlRead()
19 {
20     return potentiometer.read();
21 }
22
23 //=====[Implementations of private functions]==================================
```

***Code 8.7 Details of the implementation of light_level_control.cpp.***

Code 8.8 shows the implementation of *light_level_control.h*. The prototypes of the public functions are declared from lines 8 to 10.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _LIGHT_LEVEL_CONTROL_H_
4  #define _LIGHT_LEVEL_CONTROL_H_
5
6  //=====[Declarations (prototypes) of public functions]=========================
7
8  void lightLevelControlInit();
9  void lightLevelControlUpdate();
10 float lightLevelControlRead();
11
12 //=====[#include guards - end]=================================================
13
14 #endif // _LIGHT_LEVEL_CONTROL_H_
```

***Code 8.8 Details of the implementation of light_level_control.h.***

The new module *bright_control* is presented in Code 8.9, Code 8.10, and Code 8.11. From lines 3 to 8 of Code 8.9, the libraries used in this module are included. A private definition of LEDS_QUANTITY is declared on line 12. On line 18, the public global array object *RGBLed* of DigitalOut, relating to each of the colors of the RGB LED (PB_4 for red, PA_0 for green, and PD_12 for blue), is declared. These LEDs represent the lights of the smart home system that will be controlled using PWM, with a duty cycle that will be defined by the light level control (represented by a potentiometer).

```
1  //=====[Libraries]=============================================================
2
3  #include "arm_book_lib.h"
4
5  #include "bright_control.h"
6
7  #include "light_level_control.h"
8  #include "pc_serial_com.h"
9
10 //=====[Declaration of private defines]========================================
11
12 #define LEDS_QUANTITY 3
13
14 //=====[Declaration and initialization of public global objects]===============
15
16 DigitalOut RGBLed[] = {(PB_4), (PA_0), (PD_12)};
17
18 Ticker tickerBrightControl;
19
20 //=====[Declaration and initialization of private global variables]============
21
22 static int onTime[LEDS_QUANTITY];
23 static int offTime[LEDS_QUANTITY];
24
25 static int tickRateMsBrightControl = 1;
26 static int tickCounter[LEDS_QUANTITY];
27
28 static float periodSFloat[LEDS_QUANTITY];
29
30 //=====[Declarations (prototypes) of private functions]========================
31
32 static void setPeriod( lightSystem_t light, float period );
33 static void tickerCallbackBrightControl( );
```

***Code 8.9 Details of the implementation of bright_control.cpp (Part 1/2).***

In this example, the PWM signal will be generated using a timer interrupt associated with a ticker. The private global object *tickerBrightControl* of type ticker is declared on line 18. The global variables that account for the on and off time of the RGB LEDs are declared on lines 22 and 23, respectively. On line 25, *tickRateMsBrightControl* is declared and initialized, which will be used to set the tick rate to 1 ms to control the brightness.

Line 26 declares an array of int to account for the ticks of the ticker. On line 28, an array of type float is declared to store the period of each LED expressed in seconds. Each position of this array stores the period of each of the RGB LEDs' PWM signals.

The prototype of the function *setPeriod()* is declared on line 32. This function will be used to set the period of each of the LEDs. The callback function *tickerCallbackBrightControl()* is declared on line 33. This callback function will be called once every millisecond, as discussed below.

Code 8.10 shows the implementation of *brightControlInit()*. In line 5, *tickerBrightControl* is configured. The first parameter indicates that *tickerCallbackBrightControl()* must be called at the tick rate expressed in seconds by the second parameter, *(float) tickRateMsBrightControl) / 1000.0*. The variable is divided by 1000 and the result is *cast* to a float in order to get 0.001 (which corresponds to one millisecond).

```
1  //=====[Implementations of public functions]===================================
2
3  void brightControlInit()
4  {
5      tickerBrightControl.attach( tickerCallbackBrightControl,
6                                  ( (float) tickRateMsBrightControl) / 1000.0 );
7
8      setPeriod( RGB_LED_RED, 0.01f );
9      setPeriod( RGB_LED_GREEN, 0.01f );
10     setPeriod( RGB_LED_BLUE, 0.01f );
11
12     setDutyCycle( RGB_LED_RED, 0.5f );
13     setDutyCycle( RGB_LED_GREEN, 0.5f );
14     setDutyCycle( RGB_LED_BLUE, 0.5f );
15 }
16
17 void setDutyCycle( lightSystem_t light, float dutyCycle )
18 {
19     onTime[light] = int ( ( periodSFloat[light] * dutyCycle ) * 1000 );
20     offTime[light] = int ( periodSFloat[light] * 1000) - onTime[light];
21 }
22
23 //=====[Implementations of private functions]==================================
24
25 void setPeriod( lightSystem_t light, float period )
26 {
27     periodSFloat[light] = period;
28 }
29
30 static void tickerCallbackBrightControl( )
31 {
32     int i;
33
34     for (i = 0 ; i < LEDS_QUANTITY ; i++) {
35         tickCounter[i]++;
36         if ( RGBLed[i].read() == ON ) {
```

```
37              if( tickCounter[i] > onTime[i] ) {
38                  tickCounter[i] = 1;
39                  if ( offTime[i] ) RGBLed[i] = OFF;
40
41              }
42          } else {
43              if( tickCounter[i] > offTime[i] ) {
44                  tickCounter[i] = 1;
45                  if ( onTime[i] ) RGBLed[i] = ON;
46              }
47          }
48      }
49  }
```

***Code 8.10 Details of the implementation of bright_control.cpp (Part 2/2).***

For all three LEDs, a period of 10 milliseconds (0.01 f) is set from lines 8 to 10 using the private function *setPeriod()*. An initial duty cycle of 50% (0.5 f) is set from lines 12 to 14, using the public function *setDutyCycle()*.

The function *setDutyCycle()* is shown on lines 17 to 21 of Code 8.10. This function receives a parameter named *light* of type *lightSystem_t*, defined in *light_system.h* (Code 8.6), and a float named *dutyCycle*. Because the handler of *tickerBrightControl* is called once a millisecond, the duty cycle needs to be converted from a percentage to a time expressed in milliseconds. This implies a truncation that affects the PWM signal accuracy and resolution. To set the on time, the period defined for each of the LEDs is multiplied in line 19 by the duty cycle received as a parameter. Because the period is defined in seconds, it is multiplied by 1000 to get the value in milliseconds. Finally, the result is *cast* (forcing one data type to be converted into another) to an int variable. On line 20, the off time is computed.

**NOTE:** The cast operation removes the decimals of the result, so if the result before the cast is 2.9, the value after the cast will be 2. As the reader may notice, this operation degrades the accuracy and resolution of the PWM signal. In the next example, a way to tackle this issue will be presented.

In Code 8.10, the implementation of *setPeriod()* is shown on lines 25 to 28. This function receives a parameter named *light* of type *lightSystem_t* and a float named *period*. The value of *period* is stored in the position *light* of the array *periodSFloat* declared on line 28 of Code 8.9.

The implementation of the callback function *tickerCallbackBrightControl()* is shown on lines 30 to 49 of Code 8.10. A *for* loop is used to update the state of the pins connected to each of the colors of the RGB LEDs. The array *tickCounter* counts the milliseconds elapsed since the last transition (low to high or high to low) of each color and is incremented in each call (line 35). If the RGB LED color pin being compared is ON (line 36), then the *tickCounter* is compared with the stored value of *onTime* (line 37). Otherwise, when it is OFF (line 42), the *tickCounter* is compared with the stored value of *offTime* (line 43). When these values are reached, the *tickCounter* is reset (lines 38 and 44), and the corresponding RGB LED color pin is toggled if the duration of the next time (*onTime* or *offTime)* is different from zero (lines 39 and 45).

In Code 8.11, the implementation of *bright_control.h* is shown. In this particular case, a library is included on line 8, because the user-defined type *lightSystem_t* that is used in this module is defined in *light_system.h*. It can be seen that the prototypes of the public functions are declared on lines 16 and 17.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _BRIGHT_CONTROL_H_
4  #define _BRIGHT_CONTROL_H_
5
6  //=====[Libraries]=============================================================
7
8  #include "light_system.h"
9
10 //=====[Declarations (prototypes) of public functions]=========================
11
12 void brightControlInit();
13 void setDutyCycle( lightSystem_t light, float dutyCycle );
14
15 //=====[#include guards - end]=================================================
16
17 #endif // _BRIGHT_CONTROL_H_
```

***Code 8.11 Details of the implementation of bright_control.h.***

The initialization of the *light_level_control* module is added to *userInterfaceInit()* (line 14 of Code 8.12), and *lightLevelControlUpdate()* is called in *userInterfaceUpdate()* (line 23 of Code 8.12). As described, these functions have no functionality and are only included to maintain the same structure as in previous implementations. In order to make these calls, the library *bright_control.h* is included in *user_interface.cpp*, as can be seen in Table 8.5.

```
1  void userInterfaceInit()
2  {
3      gateOpenButton.mode(PullUp);
4      gateCloseButton.mode(PullUp);
5
6      gateOpenButton.fall(&gateOpenButtonCallback);
7      gateCloseButton.fall(&gateCloseButtonCallback);
8
9      incorrectCodeLed = OFF;
10     systemBlockedLed = OFF;
11     matrixKeypadInit( SYSTEM_TIME_INCREMENT_MS );
12     userInterfaceDisplayInit();
13
14     lightLevelControlInit();
15 }
16
17 void userInterfaceUpdate()
18 {
19     userInterfaceMatrixKeypadUpdate();
20     incorrectCodeIndicatorUpdate();
21     systemBlockedIndicatorUpdate();
22     userInterfaceDisplayUpdate();
23     lightLevelControlUpdate();
24 }
```

***Code 8.12 New implementation of the functions userInterfaceInit and userInterfaceUpdate.***

*Table 8.5 Sections in which lines were added to user_interface.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "bright_control.h"` |

### Proposed Exercises

1. How can the PWM resolution be improved?

2. Why is the function *setPeriod()* private to the module *bright_control*, while the function *setDutyCycle()* is public?

### Answers to the Exercises

1. A larger period could be defined for each of the PWM signals (lines 8 to 10 of Code 8.10). The period can be increased until the LED starts blinking instead of changing its brightness; then the maximum period has been reached.

2. The function *setPeriod()* is private because it is only used by the module *bright_control*, and the function *setDutyCycle()* is public because it is used by other modules.

## Example 8.2: Implementation of PWM using the PwmOut Class

### Objective

Use the PwmOut class to control the period and duty cycle of a PWM signal.

### Summary of the Expected Behavior

The behavior should be the same as the previous example, although the reader may notice that the accuracy and the resolution of the PWM signal is improved.

### Test the Proposed Solution on the Board

Import the project "Example 8.2" using the URL available in [4], build the project, and drag the *.bin* file onto the NUCLEO board. Rotate the knob of the potentiometer gradually, and observe how the RGB LED turns on in a white color, and its brightness changes as the knob of the potentiometer is rotated.

### Discussion of the Proposed Solution

The proposed solution modifies the module *bright_control* to use the *PwmOut* object.

### Implementation of the Proposed Solution

In Code 8.13, the new implementation of *bright_control.cpp* is shown. An object of the *class* PwmOut is used in line 11 to declare each of the pins connected to the colors of the RGB LED *PwmOut*. The reader should notice that in Code 8.9, an array of DigitalOut objects was used to declare the pins connected to the RGB LED.

As a consequence, the reader may notice that the code is now far simpler than in the previous example, by comparing Code 8.13 with Code 8.9 and Code 8.10. In Table 8.6, the sections in which lines were removed from *bright_control.cpp* are shown. Functions in which lines were removed are shown on Table 8.7. All the other lines remain the same. The implementations of *setPeriod()* and *setDutyCycle()* are reduced in Code 8.13 to a single line. In *setDutyCycle()*, the duty cycle is set on line 32, and in *setPeriod()*, the period is configured on line 39. The on and off times of the PWM signal are not truncated in this implementation, so the accuracy and resolution are improved.

***Table 8.6 Sections in which lines were removed from bright_control.cpp.***

| Section | Lines that were removed |
|---|---|
| Declaration of private defines | `#define LEDS_QUANTITY 3` |
| Declaration and initialization of public global objects | `Ticker tickerBrightControl;` |
| Declaration and initialization of private global variables | `static int onTime[LEDS_QUANTITY];`<br>`static int offTime[LEDS_QUANTITY];`<br>`int tickRateMSBrightControl = 1;`<br>`static int tickCounter[LEDS_QUANTITY];`<br>`static float periodSFloat[LEDS_QUANTITY];` |
| Declarations (prototypes) of private functions | `static void tickerCallbackBrightControl( );` |

***Table 8.7 Functions in which lines were removed from bright_control.cpp.***

| Section | Lines that were removed |
|---|---|
| void brightControlInit() | `atickerBrightControl.attach(`<br>`  tickerCallbackBrightControl,`<br>`( (float) tickRateMSBrightControl) / 1000.0 );` |
| static void tickerCallbackBrightControl() | This function was removed. |

The reader may also notice that because of modularization and the use of functions, a different implementation of PWM was introduced, and the only changes were in the bright control module. All the other new modules presented in the previous examples are *abstracted* from the way the PWM is implemented.

```
1  //=====[Libraries]=============================================================
2
3  #include "arm_book_lib.h"
4
5  #include "bright_control.h"
6
7  #include "light_level_control.h"
8
9  //=====[Declaration and initialization of public global objects]===============
10
11 PwmOut RGBLed[] = {(PB_4), (PA_0), (PD_12)};
12
13 //=====[Declaration and initialization of private global variables]============
14
15 static void setPeriod( lightSystem_t light, float period );
16
17 //=====[Implementations of public functions]===================================
18
19 void brightControlInit()
20 {
21     setPeriod( RGB_LED_RED, 0.01f );
22     setPeriod( RGB_LED_GREEN, 0.01f );
23     setPeriod( RGB_LED_BLUE, 0.01f );
24
25     setDutyCycle( RGB_LED_RED, 0.5 );
26     setDutyCycle( RGB_LED_GREEN, 0.5 );
27     setDutyCycle( RGB_LED_BLUE, 0.5 );
28 }
29
30 void setDutyCycle( lightSystem_t light, float dutyCycle )
31 {
32     RGBLed[light].write(dutyCycle);
33 }
34
35 //=====[Implementations of private functions]==================================
36
37 static void setPeriod( lightSystem_t light, float period )
38 {
39     RGBLed[light].period(period);
40 }
```

***Code 8.13 Details of the implementation of bright_control.cpp.***

## Proposed Exercise

1. How can the code of Examples 8.1 and 8.2 be compared, considering that their functionality is the same?

## Answer to the Exercise

1. In Example 8.1, the PWM technique was implemented using a *ticker* object and a set of functions that were implemented, like *setDutyCycle()*, *setPeriod()*, and *tickerCallbackBrightControl()*. In Example 8.2, the object *PwmOut* was used, which simplifies the usage of the PWM technique. Having implemented all the details in Example 8.1 helps in understanding what is going on in the background when the object *PwmOut* is used.

### Example 8.3: Control the Siren and Strobe Light using PWM

#### Objective

Use the PwmOut object to control the siren and the strobe light.

#### Summary of the Expected Behavior

The behavior should be the same as the previous example, although the problem related to the timing of the strobe light and siren not always being 100 ms is addressed.

#### Test the Proposed Solution on the Board

Import the project "Example 8.3" using the URL available in [4], build the project, and drag the *.bin* file onto the NUCLEO board. Activate the motion sensor and the fire alarm, and observe that the time for which the strobe light and siren are on and off is always 100 ms.

#### Discussion of the Proposed Solution

The proposed solution modifies the module *siren* and *strobe_light t*o use the PwmOut object. In this way, there is not a DigitalOut object that is set on and off, as was the case in the previous implementation, but a PwmOut object that is configured to alternate its state over time or to remain in the off state all the time, depending on the value of the variable *sirenState*.

#### Implementation of the Proposed Solution

The proposed new implementation of *siren.cpp* and *strobe_light.cpp* is shown in Code 8.14 and Code 8.15, respectively. These two implementations are identical except for the use of *siren* and *strobeLight* in each case, and the value of the parameter in lines 24 and 46.

On line 12, the object is changed from DigitalOut to PwmOut. In order to avoid changes in other files, the prototypes of all the public functions were left unmodified. When the function *sirenUpdate()* or *strobeLightUpdate()* is called, the current strobe time (declared on line 17: *currentStrobeTime*) is compared with the received parameter (line 40). If they are different, then the received strobe time is multiplied by two (because the strobe time accounts only for the time the alarm must be in ON state, while *sirenPin.period* is the sum of the time it is on and off), converted from milliseconds to seconds (i.e., multiplied by 1000), and cast to float to set the PWM period (line 41). Then, the PWM signal duty cycle is set to 50% (line 42), and the current strobe time is updated (line 43).

In order to turn off the siren, a 100% duty cycle is set (line 46 of Code 8.14) because the buzzer is turned off with a high state signal. In the case of the strobe light, a 0% duty cycle is set (line 46 of Code 8.15) because LD1 is turned off with a low state signal. The current strobe time is set to 0 in both programs (line 47). The initialization of each module is implemented by setting a period of 1 second (line 23) with a 100% duty cycle (line 24) (in the case of the *strobe light*, a 0% duty cycle is set).

```
//=====[Libraries]=============================================================

#include "mbed.h"
#include "arm_book_lib.h"

#include "siren.h"

#include "smart_home_system.h"

//=====[Declaration and initialization of public global objects]===============

PwmOut sirenPin(PC_9_ALT0);

//=====[Declaration and initialization of private global variables]============

static bool sirenState = OFF;
static int currentStrobeTime = 0;

//=====[Implementations of public functions]===================================

void sirenInit()
{
    sirenPin.period(1.0f);
    sirenPin.write(1.0f);
}

bool sirenStateRead()
{
    return sirenState;
}

void sirenStateWrite( bool state )
{
    sirenState = state;
}

void sirenUpdate( int strobeTime )
{
    if( sirenState ) {
        if (currentStrobeTime != strobeTime) {
            sirenPin.period( (float) strobeTime * 2 / 1000 );
            sirenPin.write(0.5f);
            currentStrobeTime = strobeTime;
        }
    } else {
        sirenPin.write(1.0f);
        currentStrobeTime = 0;
    }
}
```

***Code 8.14 Details of the new implementation of siren.cpp.***

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "strobe_light.h"
7
8  #include "smart_home_system.h"
9
10 //=====[Declaration and initialization of public global objects]===============
11
12 PwmOut strobeLight(LED1);
13
14 //=====[Declaration and initialization of private global variables]============
15
16 static bool strobeLightState = OFF;
17 static int currentStrobeTime = 0;
18
19 //=====[Implementations of public functions]===================================
20
21 void strobeLightInit()
22 {
23     strobeLight.period(1.0f);
24     strobeLight.write(0.0f);
25 }
26
27 bool strobeLightStateRead()
28 {
29     return strobeLightState;
30 }
31
32 void strobeLightStateWrite( bool state )
33 {
34     strobeLightState = state;
35 }
36
37 void strobeLightUpdate( int strobeTime )
38 {
39     if( strobeLightState ) {
40         if (currentStrobeTime != strobeTime) {
41             strobeLight.period( (float) strobeTime * 2 / 1000 );
42             strobeLight.write(0.5f);
43             currentStrobeTime = strobeTime;
44         }
45     } else {
46         strobeLight.write(0.0f);
47         currentStrobeTime = 0;
48     }
49 }
```

***Code 8.15 Details of the new implementation of strobe_light.cpp.***

### Example 8.4: Adjustment of the Color of the Decorative RGB LED

#### Objective

Upgrade the code to allow independent control of each LED.

#### Summary of the Expected Behavior

The color of the decorative RGB LED is configured using the matrix keypad.

#### Test the Proposed Solution on the Board

Import the project "Example 8.4" using the URL available in [4], build the project, and drag the *.bin* file onto the NUCLEO board. Rotate the knob of the potentiometer and set the maximum brightness of the RGB LED. Press button 4 five times on the matrix keypad and observe how gradually the red color of the RGB LED turns off. Press button 5 five times on the matrix keypad and observe how gradually the green color of the RGB LED turns off. Press button 6 five times on the matrix keypad and observe how gradually the blue color of the RGB LED turns off. Now press buttons 1, 2, and 3 several times in order to see how the red, green, and blue colors of the RGB LED turn on. Rotate the knob of the potentiometer to see that the light color of the RGB LED remains unchanged, while the brightness increases and decreases.

**NOTE:** Buttons 1 through 6 of the matrix keypad are only available to use for this functionality when the alarm is OFF.

#### Discussion of the Proposed Solution

The proposed solution modifies only the modules *user_interface* and *light_system*. An independent factor for each color is introduced in order to multiply the duty cycle that is configured using the potentiometer. In this way, each color can be varied independently.

#### Implementation of the Proposed Solution

In Code 8.16, the new implementation of the function *userInterfaceMatrixKeypadUpdate()* is shown. A new set of options is included using the *if* statement from lines 34 to 51. In each of these options, the function *lightSystemBrightnessChangeRGBFactor()* is called with all the possible combinations of colors and true or false.

```
static void userInterfaceMatrixKeypadUpdate()
{
    static int numberOfHashKeyReleased = 0;
    char keyReleased = matrixKeypadUpdate();

    if( keyReleased != '\0' ) {

        if( alarmStateRead() && !systemBlockedStateRead() ) {
            if( !incorrectCodeStateRead() ) {
                codeSequenceFromUserInterface[numberOfCodeChars] = keyReleased;
                numberOfCodeChars++;
                if ( numberOfCodeChars >= CODE_NUMBER_OF_KEYS ) {
                    codeComplete = true;
                    numberOfCodeChars = 0;
                }
            } else {
                if( keyReleased == '#' ) {
                    numberOfHashKeyReleased++;
                    if( numberOfHashKeyReleased >= 2 ) {
                        numberOfHashKeyReleased = 0;
                        numberOfCodeChars = 0;
                        codeComplete = false;
                        incorrectCodeState = OFF;
                    }
                }
            }
        } else if ( !systemBlockedStateRead() ) {
            if( keyReleased == 'A' ) {
                motionSensorActivate();
            }
            if( keyReleased == 'B' ) {
                motionSensorDeactivate();
            }
            if( keyReleased == '1' ) {
                lightSystemBrightnessChangeRGBFactor( RGB_LED_RED, true );
            }
            if( keyReleased == '2' ) {
                lightSystemBrightnessChangeRGBFactor( RGB_LED_GREEN, true );
            }
            if( keyReleased == '3' ) {
                lightSystemBrightnessChangeRGBFactor( RGB_LED_BLUE, true );
            }
            if( keyReleased == '4' ) {
                lightSystemBrightnessChangeRGBFactor( RGB_LED_RED, false );
            }
            if( keyReleased == '5' ) {
                lightSystemBrightnessChangeRGBFactor( RGB_LED_GREEN, false );
            }
            if( keyReleased == '6' ) {
                lightSystemBrightnessChangeRGBFactor( RGB_LED_BLUE, false );
            }
        }
    }
}
```

***Code 8.16 Details of the new implementation of userInterfaceMatrixKeypadUpdate().***

In order to make these calls, the library *light_system.h* is included in *user_interface.cpp*, as can be seen in Table 8.8.

***Table 8.8 Sections in which lines were added to user_interface.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "light_system.h"` |

The new implementation of the function *lightSystemUpdate()* is shown in Code 8.17. On lines 5 to 7, the function *setDutyCycle()* now includes an independent brightness factor for each color. These variables are declared and initialized to 0.5 f in *light_system.cpp*, as can be seen in Table 8.9.

The new function *lightSystemBrightnessChangeRGBFactor()* is shown in Code 8.17. Depending on the value of the parameters *light* and *state*, the brightness factor of each color is increased or decreased by 0.1. *If* statements are used to keep the brightness factor values between 0 and 1. The prototype of this function is included in *light_system.h*, as shown in Table 8.10.

```
 1  void lightSystemUpdate()
 2  {
 3      dutyCycle = lightLevelControlRead();
 4
 5      setDutyCycle( RGB_LED_RED, brightnessRGBLedRedFactor*dutyCycle );
 6      setDutyCycle( RGB_LED_GREEN, brightnessRGBLedGreenFactor*dutyCycle );
 7      setDutyCycle( RGB_LED_BLUE, brightnessRGBLedBlueFactor*dutyCycle );
 8  }
 9
10  void lightSystemBrightnessChangeRGBFactor( lightSystem_t light, bool state )
11  {
12      switch( light ) {
13          case RGB_LED_RED:
14              if ( state ) brightnessRGBLedRedFactor+=0.1;
15                  else brightnessRGBLedRedFactor-=0.1;
16              if ( brightnessRGBLedRedFactor > 1) brightnessRGBLedRedFactor=1.0;
17              if ( brightnessRGBLedRedFactor < 0) brightnessRGBLedRedFactor=0.0;
18          break;
19          case RGB_LED_GREEN:
20              if ( state ) brightnessRGBLedGreenFactor+=0.1;
21                  else brightnessRGBLedGreenFactor-=0.1;
22              if ( brightnessRGBLedGreenFactor > 1) brightnessRGBLedGreenFactor=1.0;
23              if ( brightnessRGBLedGreenFactor < 0) brightnessRGBLedGreenFactor=0.0;
24          break;
25          case RGB_LED_BLUE:
26              if ( state ) brightnessRGBLedBlueFactor+=0.1;
27                  else brightnessRGBLedBlueFactor-=0.1;
28              if ( brightnessRGBLedBlueFactor > 1) brightnessRGBLedBlueFactor=1.0;
29              if ( brightnessRGBLedBlueFactor < 0) brightnessRGBLedBlueFactor=0.0;
30          break;
31          default:
32          break;
33      }
34  }
```

***Code 8.17 Details of the new implementation of lightSystemUpdate() and the implementation of lightSystemBrightnessChangeRGBFactor().***

*Table 8.9 Sections in which lines were added to light_system.cpp.*

| Section | Lines that were added |
|---|---|
| Declaration and initialization of private global variables | `static float brightnessRGBLedRedFactor = 0.5f;`<br>`static float brightnessRGBLedGreenFactor = 0.5f;`<br>`static float brightnessRGBLedBlueFactor = 0.5f;` |

*Table 8.10 Sections in which lines were added to light_system.h.*

| Section | Lines that were added |
|---|---|
| Declarations (prototypes) of public functions | `void lightSystemBrightnessChangeEnable( lightSystem_t light, bool state );` |

### Proposed Exercises

1. How many different colors can be obtained by means of the implemented functionality?

2. How can the number of obtainable colors be increased?

### Answers to the Exercises

1. Each color factor can take 11 intensity values (from 0.0 to 1.0). Therefore, 11 × 11 × 11 = 1331 different colors can be obtained.

2. By means of modifying the 0.1 factor that is used in *lightSystemBrightnessChangeRGBFactor()*. For example, if this value is changed to 0.05, then 21 × 21 × 21 = 9621 different colors can be obtained.

## Example 8.5: Use of the Light Sensor Reading to Control the RGB LED

### Objective

Introduce the basics of a negative feedback control.

### Summary of the Expected Behavior

The brightness of the RGB LEDs is governed by the LDR reading.

**NOTE:** The LDR should be placed as close as possible to the RGB LED, and ambient light should be reduced as much as possible.

### Test the Proposed Solution on the Board

Import the project "Example 8.5" using the URL available in [4], build the project, and drag the *.bin* file onto the NUCLEO board. Set the potentiometer to mid-way through its range. Change the lighting conditions of the LDR. Observe how the RGB LED responds to these changes.

**WARNING:** If the RGB LED does not respond as expected, it is recommended to wait a few seconds until the system stabilizes. If after waiting a few seconds the problem is not solved, in the Proposed Exercises subsection an implementation will be shown that will allow the reader to understand what is happening.

### Discussion of the Proposed Solution

The proposed solution is based on a new module called *ldr_sensor* and the implementation of a *negative feedback control system*. The potentiometer is used to establish a *set point* in the range of 0 to 1, and the light intensity is measured by means of the LDR circuit introduced in Figure 8.6, obtaining a value in the range of 0 to 1. The duty cycle of the RGB LED is increased or decreased in order to make the reading of the LDR sensor as similar as possible to the set point established using the potentiometer.

### Implementation of the Proposed Solution

A new module called *ldr_sensor* is created, with its implementation shown in Code 8.18 and Code 8.19. Because the implementation is similar to the module *light_level_sensor*, no explanation will be included.

```
1  //=====[Libraries]=============================================================
2
3  #include "arm_book_lib.h"
4
5  #include "smart_home_system.h"
6  #include "ldr_sensor.h"
7
8  //=====[Declaration and initialization of public global objects]===============
9
10 AnalogIn LDR(A2);
11
12 //=====[Implementations of public functions]===================================
13
14 void ldrSensorInit() { }
15
16 void ldrSensorUpdate() { }
17
18 float ldrSensorRead()
19 {
20     return LDR.read();
21 }
```

***Code 8.18 Details of the implementation of ldr_sensor.cpp.***

In Code 8.8, the implementation of *light_level_control.h* is shown. It can be seen that the prototypes of the public functions are declared in lines 14 to 16.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _LDR_SENSOR_H_
4  #define _LDR_SENSOR_H_
5
6  //=====[Declarations (prototypes) of public functions]=========================
7
8  void ldrSensorInit();
9  void ldrSensorUpdate();
10 float ldrSensorRead();
11
12 //=====[#include guards - end]=================================================
13
14 #endif // _LDR_SENSOR_H_
```

*Code 8.19 Details of the implementation of ldr_sensor.h.*

In Code 8.20, the new implementation of the function *lightSystemUpdate()* is shown. The only difference appears on lines 3 and 4, where a negative feedback control system is implemented. The basics of negative feedback control theory are explained in the Under the Hood section. For now, it is enough to explain that the difference between the reading of the LDR sensor and the set point (line 4) is multiplied by a gain factor (line 5) and added to the previous value of the duty cycle. In this way, the duty cycle is increased or decreased until the difference between the LDR sensor reading and the set point becomes negligible.

Note that if the set point or the LDR sensor reading is modified (for example, the potentiometer knob is rotated), then the difference changes, so the duty cycle is modified again in order to reduce this difference.

In order to make the call to *LDRSensorRead()*, the library *ldr_sensor.h* is included in *light_system.cpp*, as can be seen in Table 8.11. In the same table, a variable named *lightSystemLoopGain* is declared and initialized.

```
1  void lightSystemUpdate()
2  {
3      dutyCycle = dutyCycle + lightSystemLoopGain
4                            * (lightLevelControlRead() - ldrSensorRead());
5
6      setDutyCycle( RGB_LED_RED, brightnessRGBLedRedFactor*dutyCycle );
7      setDutyCycle( RGB_LED_GREEN, brightnessRGBLedGreenFactor*dutyCycle );
8      setDutyCycle( RGB_LED_BLUE, brightnessRGBLedBlueFactor*dutyCycle );
9  }
```

*Code 8.20 Details of the new implementation of the function lightSystemUpdate().*

*Table 8.11 Sections in which lines were added to light_system.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "ldr_sensor.h"` |
| Declaration and initialization of private global variables | `static float lightSystemLoopGain = 0.01;` |

## Proposed Exercises

1. How can the code be modified to monitor the variables related to the negative feedback control system?

2. How can the duty cycle value be limited to the range 0 to 1, in order to improve the system behavior?

## Answers to the Exercises

1. Lines 3, 4, and 13 to 34 could be added to the function *lightSystemUpdate()*, as shown in Code 8.21. Ten positions were assigned to str (line 4) for safety reasons because *dutyCycle* is not limited in size, as discussed below.

```
 1  void lightSystemUpdate()
 2  {
 3      static int i = 0;
 4      char str[10];
 5
 6      dutyCycle = dutyCycle + lightSystemLoopGain
 7                            * (lightLevelControlRead() - ldrSensorRead());
 8
 9      setDutyCycle( RGB_LED_RED, brightnessRGBLedRedFactor*dutyCycle );
10      setDutyCycle( RGB_LED_GREEN, brightnessRGBLedGreenFactor*dutyCycle );
11      setDutyCycle( RGB_LED_BLUE, brightnessRGBLedBlueFactor*dutyCycle );
12
13      if (i > 100) {
14          i=0;
15
16          pcSerialComStringWrite("SP: ");
17          sprintf( str, "%0.4f", lightLevelControlRead() );
18          pcSerialComStringWrite( str );
19          pcSerialComStringWrite(" | ");
20          pcSerialComStringWrite("LDR: ");
21          sprintf( str, "%0.4f", ldrSensorRead() );
22          pcSerialComStringWrite( str );
23          pcSerialComStringWrite(" | ");
24          pcSerialComStringWrite("Duty: ");
25          sprintf( str, "%0.4f", dutyCycle );
26          pcSerialComStringWrite( str );
27          pcSerialComStringWrite(" | ");
28          pcSerialComStringWrite("Added: ");
29          sprintf( str, "%0.4f", lightSystemLoopGain
30                                 * (lightLevelControlRead() - ldrSensorRead()) );
31          pcSerialComStringWrite( str );
32          pcSerialComStringWrite("\r\n");
33      }
34      i++;
35  }
```

***Code 8.21 Details of the new implementation of the function lightSystemUpdate().***

In this way, an output similar to the one presented in Figure 8.14 should appear on the serial terminal.

**NOTE:** The line numbers in Figure 8.14 have been added for pedagogical purposes only.

**WARNING:** Ambient light should be reduced as much as possible when testing this program.

In Figure 8.14, as shown in Code 8.21, *SP* stands for *set point* (the reading of the potentiometer), *LDR* shows the reading of the LDR, *Duty* shows the *dutyCycle* (where 0 stands for 0% and 1 for 100%), and *Added* shows the value added in each call of the variable *dutyCycle*.

**NOTE:** These parameters are shown once every 100 calls of *lightSystemUpdate()* (approximately once every second), as defined on line 12 of Code 8.21.

There are some interesting things to highlight in the output presented in Figure 8.14:

- From lines 1 to 5, the loop appears to be stable despite the noise of the LDR reading.
- From lines 6 to 11, the LDR is exposed to external light. This can be identified by an increase in its value. The negative feedback control system tries to compensate for this change by reducing *dutyCycle*, but because there are no limits, starting from line 8 the duty cycle moves into negative values.

**WARNING:** Duty cycles cannot have negative values. The negative values are a consequence of the implementation of the negative feedback control system. The interface PwmOut assigns 0 in these cases.

- From lines 12 to 21, the LDR is not exposed to external light. This can be identified by a decrease in its value. The negative feedback control system is able to minimize the error, and the duty cycle presents values between 0 and 1 starting from line 17.
- From lines 21 to 27, the LDR is again exposed to external light, and the potentiometer is turned to the extremes of its rotation. The negative feedback control system tries to compensate for this change by increasing *dutyCycle* but, again because there are no limits, starting from line 23 the duty cycle takes values bigger than 1.

**WARNING:** Duty cycles cannot have values beyond 100%. Values beyond 1 are a consequence of the implementation of the negative feedback control system. The interface PwmOut assigns 1 in these cases.

```
1   SP: 0.2369 | LDR: 0.2046 | Duty: 0.2678 | Added: 0.0010
2   SP: 0.2374 | LDR: 0.1951 | Duty: 0.3121 | Added: 0.0010
3   SP: 0.2361 | LDR: 0.1832 | Duty: 0.3605 | Added: 0.0002
4   SP: 0.2352 | LDR: 0.2042 | Duty: 0.4100 | Added: 0.0001
5   SP: 0.2369 | LDR: 0.1954 | Duty: 0.4613 | Added: -0.0017
6   SP: 0.2354 | LDR: 0.4303 | Duty: 0.3605 | Added: -0.0032
7   SP: 0.2381 | LDR: 0.5067 | Duty: 0.1212 | Added: -0.0021
8   SP: 0.2374 | LDR: 0.4281 | Duty: -0.0811 | Added: -0.0019
9   SP: 0.2369 | LDR: 0.4300 | Duty: -0.2743 | Added: -0.0019
10  SP: 0.2342 | LDR: 0.4313 | Duty: -0.4704 | Added: -0.0020
11  SP: 0.2347 | LDR: 0.4379 | Duty: -0.6724 | Added: -0.0020
12  SP: 0.2352 | LDR: 0.0525 | Duty: -0.8133 | Added: 0.0018
13  SP: 0.2359 | LDR: 0.0364 | Duty: -0.6107 | Added: 0.0020
14  SP: 0.2359 | LDR: 0.0344 | Duty: -0.4080 | Added: 0.0020
15  SP: 0.2357 | LDR: 0.0371 | Duty: -0.2055 | Added: 0.0020
16  SP: 0.2369 | LDR: 0.0371 | Duty: -0.0041 | Added: 0.0020
17  SP: 0.2386 | LDR: 0.1438 | Duty: 0.1236 | Added: 0.0014
18  SP: 0.2371 | LDR: 0.1683 | Duty: 0.2022 | Added: 0.0012
19  SP: 0.2364 | LDR: 0.1768 | Duty: 0.2594 | Added: 0.0011
20  SP: 0.2376 | LDR: 0.1766 | Duty: 0.3355 | Added: 0.0002
21  SP: 0.2357 | LDR: 0.1766 | Duty: 0.4040 | Added: -0.0017
22  SP: 0.9990 | LDR: 0.4965 | Duty: 0.7392 | Added: 0.0071
23  SP: 0.9961 | LDR: 0.5026 | Duty: 1.2705 | Added: 0.0050
24  SP: 0.9949 | LDR: 0.5021 | Duty: 1.7722 | Added: 0.0050
25  SP: 0.9988 | LDR: 0.5023 | Duty: 2.2741 | Added: 0.0050
26  SP: 0.9993 | LDR: 0.5016 | Duty: 2.7758 | Added: 0.0050
27  SP: 0.9978 | LDR: 0.5023 | Duty: 3.2778 | Added: 0.0049
```

***Figure 8.14 Output of serial terminal generated by proposed example 8.4.***

2. The duty cycle values can be limited within the 0 to 1 range by means of including lines 9 and 10 shown in Code 8.22. This implementation is a first step towards *saturation arithmetic*, which is a version of arithmetic in which all operations such as addition and multiplication are limited to a fixed range between a minimum and maximum value. This is very important in the context of *control systems*, such as the one implemented in this example.

```
 1  void lightSystemUpdate()
 2  {
 3      static int i = 0;
 4      char str[100] = "";
 5
 6      dutyCycle = dutyCycle + lightSystemLoopGain
 7                              * (lightLevelControlRead() - ldrSensorRead());
 8
 9      if ( dutyCycle > 1 ) dutyCycle = 1;
10      if ( dutyCycle < 0 ) dutyCycle = 0;
11
12      setDutyCycle( RGB_LED_RED, brightnessRGBLedRedFactor*dutyCycle );
13      setDutyCycle( RGB_LED_GREEN, brightnessRGBLedGreenFactor*dutyCycle );
14      setDutyCycle( RGB_LED_BLUE, brightnessRGBLedBlueFactor*dutyCycle );
15
16      if (i > 100) {
17          i=0;
18
19          pcSerialComStringWrite("SP: ");
20          sprintf( str, "%0.4f", lightLevelControlRead() );
21          pcSerialComStringWrite( str );
22          pcSerialComStringWrite(" | ");
23          pcSerialComStringWrite("LDR: ");
24          sprintf( str, "%0.4f", ldrSensorRead() );
25          pcSerialComStringWrite( str );
26          pcSerialComStringWrite(" | ");
27          pcSerialComStringWrite("Duty: ");
28          sprintf( str, "%0.4f", dutyCycle );
29          pcSerialComStringWrite( str );
30          pcSerialComStringWrite(" | ");
31          pcSerialComStringWrite("Added: ");
32          sprintf( str, "%0.4f", lightSystemLoopGain
33                                  * (lightLevelControlRead() - ldrSensorRead()) );
34          pcSerialComStringWrite( str );
35          pcSerialComStringWrite("\r\n");
36      }
37      i++;
38  }
```

***Code 8.22 Details of the new implementation of the function lightSystemUpdate().***

### Example 8.6: Playback of an Audio Message using the PWM Technique

#### Objective

Introduce the basics of how to obtain analog signals using the PWM technique.

#### Summary of the Expected Behavior

Play back a “Welcome to the Smart Home System” message during the smart home system power up.

#### Test the Proposed Solution on the Board

Import the project “Example 8.6” using the URL available in [4], build the project, and drag the *.bin* file onto the NUCLEO board. Plug headphones into the audio jack. The “Welcome to the Smart Home System” message should be heard every time the NUCLEO board is restarted.

### Discussion of the Proposed Solution

The proposed solution is based on a new module named *audio* and the usage of the PWM technique together with an appropriate low pass filter and a digitized audio signal, as discussed in subsection 8.2.2. The new module has three files: *audio.h*, *audio.cpp*, and *welcome_message.h*. The information about the digitized audio signal is stored in *welcome_message.h*, as explained below.

### Implementation of the Proposed Solution

In Code 8.23, the new implementation of *smart_home_system.cpp* is shown. The *audio* module is included in line 17, and the function *audioInit()* is called on line 23.

In Code 8.24, the implementation of *audio.h* is shown. The only public function of this module is *audioInit()*.

The implementation of *audio.cpp* is shown in Code 8.25. On lines 6 and 8, the files *welcome_message.h* and *audio.h* are included. On line 12, AUDIO_SAMPLE_DURATION is defined with the value of 125. On line 16, an object of type *PwmOut* named *audioOut* is declared and assigned to PE_6. On line 20, the private function *welcomeMessage()* is declared. The implementation of the public function *audioInit()* is shown on line 24, which only makes a call to *welcomeMessage()* on line 25 and then returns.

The implementation of the private function *welcomeMessage()* is shown on line 32. On line 34, the float variable *audioDutyCycle* is declared and initialized at zero. On line 36, the period of *audioOut* is set to 25 microseconds, as shown in Figure 8.13. On line 38, the integer variable *i* is declared, and on line 39 it is used in a *for* loop.

On line 40, *audioDutyCycle* is assigned with the value of *welcomeMessageData[i]* divided by 255. Note that the (float) cast is used, otherwise this value will always be zero because *welcomeMessageData* is an array of unsigned char, as will be seen below.

On line 41, the duty cycle of *audioOut* is set to *audioDutyCycle*, and on line 42, the Mbed OS function *wait_us()* is called in order to introduce a delay of length AUDIO_SAMPLE_DURATION (125 microseconds). In this way, five periods of the PWM are generated with the same duty cycle, as illustrated in Figure 8.13.

In Code 8.26, the first lines of *welcome_message.h* are shown. On line 1, the *constant* integer variable *welcomeMessageLength* is defined. The *const* keyword is used to prevent overriding the variable's value. On line 3, the first part of the constant array of type unsigned char named *welcomeMessageData* is shown.

```
//=====[Libraries]=============================================================

#include "arm_book_lib.h"

#include "smart_home_system.h"

#include "alarm.h"
#include "user_interface.h"
#include "fire_alarm.h"
#include "intruder_alarm.h"
#include "pc_serial_com.h"
#include "event_log.h"
#include "motion_sensor.h"
#include "motor.h"
#include "gate.h"
#include "light_system.h"
#include "audio.h"

//=====[Implementations of public functions]===================================

void smartHomeSystemInit()
{
    audioInit();
    userInterfaceInit();
    alarmInit();
    fireAlarmInit();
    intruderAlarmInit();
    pcSerialComInit();
    motorControlInit();
    gateInit();
    lightSystemInit();
}

void smartHomeSystemUpdate()
{
    userInterfaceUpdate();
    fireAlarmUpdate();
    intruderAlarmUpdate();
    alarmUpdate();
    eventLogUpdate();
    pcSerialComUpdate();
    lightSystemUpdate();
    delay(SYSTEM_TIME_INCREMENT_MS);
}
```

***Code 8.23 Details of the new implementation of the smart_home_system.cpp.***

```
//=====[#include guards - begin]===============================================

#ifndef _AUDIO_H_
#define _AUDIO_H_

//=====[Libraries]=============================================================

void audioInit();

//=====[#include guards - end]=================================================

#endif // _AUDIO_H_
```

***Code 8.24 Details of the implementation of the audio.h.***

```
//=====[Libraries]=============================================================

#include "mbed.h"
#include "arm_book_lib.h"

#include "welcome_message.h"

#include "audio.h"

//=====[Declaration of private defines]========================================

#define AUDIO_SAMPLE_DURATION   125

//=====[Declaration and initialization of public global objects]===============

PwmOut audioOut(PE_6);

//=====[Declarations (prototypes) of private functions]========================

static void welcomeMessage();

//=====[Implementations of public functions]===================================

void audioInit()
{
    welcomeMessage();
    return;
}

//=====[Implementations of private functions]==================================

static void welcomeMessage()
{
    float audioDutyCycle = 0.0;

    audioOut.period(0.000025f);

    int i = 0;
    for( i=1; i<welcomeMessageLength; i++ ) {
        audioDutyCycle = (float) welcomeMessageData[i]/255;
        audioOut.write(audioDutyCycle);
        wait_us(AUDIO_SAMPLE_DURATION);
    }

    return;
}
```

***Code 8.25 Details of the implementation of audio.cpp.***

```
const int welcomeMessageLength=24000;

const unsigned char welcomeMessageData[] = {128, 128, 128, 128, 128, 128, ...
```

***Code 8.26 Details of the implementation of welcome_message.h.***

## Proposed Exercise

1. How can the welcome message be modified?

### Answer to the Exercise

1. The file *welcome_message.h* should be modified with the data corresponding to the new message.

**TIP:** Many online "text to speech" tools are available on the internet, as well as many wav to C converters. These enable different messages to be generated, even in different languages.

## 8.3 Under the Hood

### 8.3.1 Fundamentals of Control Theory

In this chapter, the brightness of an RGB LED was controlled by means of the NUCLEO board considering a *set point reference* established using a potentiometer and reading the light intensity using an LDR. This implementation can be analyzed using *control theory*. In Figure 8.15, a diagram is shown of a *negative feedback control system*. It is based on a feedback loop, which controls the process variable by comparing it with a desired value (the reference) and applying the difference (measured error) as an error signal to generate a control output to reduce or eliminate the error.

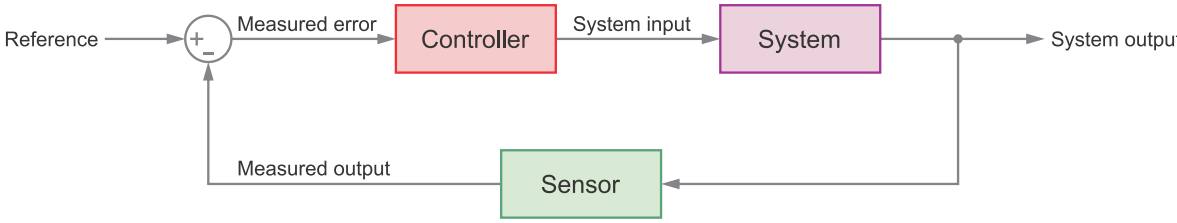



*Figure 8.15 Diagram of a negative feedback control system.*

The model shown in Figure 8.15 can be applied to the setup used in Example 8.6, as shown in Figure 8.16, where the duty cycle of the PWM signal was obtained by adding the current value of the duty cycle to the product of a given gain and the difference between the reading of the analog conversion of A0 and A2:

$$dutyCycle = dutyCycle + lightSystemLoopGain * (lightLevelControlRead - LDRSensorRead) \quad (1)$$

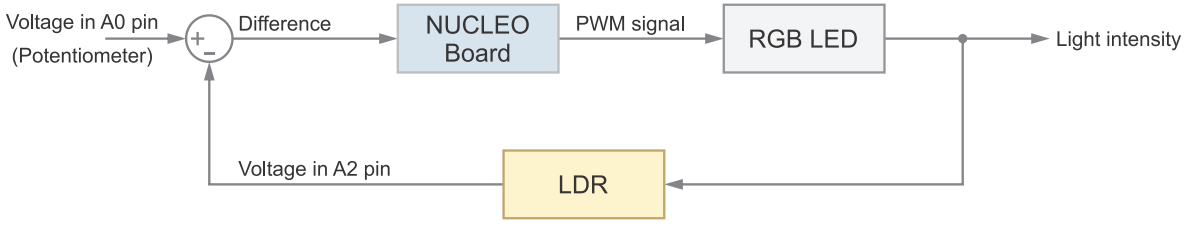



*Figure 8.16 Diagram of the negative feedback control system implemented in Example 8.5.*

In Example 8.6, different values can be tested with the controller gain, and it will be seen that, depending on the values, the response of the controller can be appropriate, or can be unstable or too slow.

If the control system is implemented using a microcontroller, as in the example above, it is referred to as *digital control*, and its behavior is adjusted by the values of the parameters used in the computation. If the control system is implemented using analog components, such as *operational amplifiers*, it is called *analog control*, and the system behavior depends on the values of resistors, capacitors, etc. Usually, a digital controller is more resistant to noise, more power efficient, and needs less maintenance, because digital devices do not tend to degrade or get damaged over time or need to be calibrated, which is often the case with analog devices.

### Proposed Exercise

1. In the Example 8.5 implementation, is the *measured error* signal obtained outside the controller?

### Answer to the Exercise

1. No, it can be seen that in Example 8.5 the *measured error* is computed by the microcontroller.

## 8.4 Case Study

### 8.4.1 Smart City Bike Lights

In this chapter, an RGB LED and a light sensor were connected to the NUCLEO board. In this way, the brightness of the RGB LED was controlled using PWM. An example of smart city bike lights, built with Mbed with similar features, can be found in [7]. In Figure 8.17, the smart city bike light is shown mounted on a bike (red rear light, on the left), and the set of front and rear lights is shown with the smartphone application (on the right).


*Figure 8.17 On the left, smart city bike light mounted on a bike. On the right, rear and front lights and the mobile app.*

The smart city bike light system can adjust its brightness level based on ambient light to conserve battery life. It also flashes brighter and faster depending on a set of conditions to make sure that the cyclist stands out. It is also provided with accelerometers to monitor various conditions such as swerving, sudden braking, road surface condition, and falls. It contains two different types of LEDs (focused and dispersed beam) to make it visible up to 3 km away and also gives 270° of side visibility. The mobile app lets the cyclist personalize light settings and gives low battery alerts straight to the phone.

The smart city bike light uses the PWM technique introduced in this chapter to adjust the brightness level. The sensor used in this chapter to measure ambient light is also very similar to the sensor used by the smart city bike light. Moreover, the system is connected to a smartphone using Bluetooth Low Energy in the same way as will be shown for the smart home system in Chapter 10.

### Proposed Exercise

1. The smart city bike light states that it has 300 lumens of luminous flux in the rear and 400 lumens in the front. How does this compare with the maximum light intensity that an RGB LED such as the one used in this chapter can provide?

### Answer to the Exercise

1. The luminous flux $\Phi_v$ in lumens (lm) is related to the luminous intensity $I_v$ in candela (cd) and the apex angle $\theta$ in degrees (°) by means of the following formula:

$$\Phi_{v(lm)} = I_{v(cd)} \times ( 2\pi(1 - \cos(\theta/2)) ) \qquad (2)$$

Considering that the RGB LED has three LEDs and that altogether can provide up to 5 cd over a typical viewing angle of 45°, as shown in [1], the luminous flux results in about 2.5 lumens. This value is about a hundred times smaller than the luminous flux provided by the smart city bike light rear and front lights.

## References

[1] “RGB LED Pinout, Features, Circuit & Datasheet”. Accessed July 9, 2021. https://components101.com/diodes/rgb-led-pinout-configuration-circuit-datasheet

[2] “LDR Pinout, Working, Applications & Datasheet”. Accessed July 9, 2021. https://components101.com/resistors/ldr-datasheet

[3] “Low-pass filter - Wikipedia”. Accessed July 9, 2021. https://en.wikipedia.org/wiki/Low-pass_filter#RC_filter

[4] “GitHub - armBookCodeExamples/Directory”. Accessed July 9, 2021. https://github.com/armBookCodeExamples/Directory/

[5] "pinout_labels - | Mbed". Accessed July 9, 2021. https://os.mbed.com/teams/ST/wiki/pinout_labels

[6] "mbed-os/PeripheralPinMaps.h at master · ARMmbed/mbed-os · GitHub". Accessed July 9, 2021. https://github.com/ARMmbed/mbed-os/blob/master/targets/TARGET_STM/TARGET_STM32F4/TARGET_STM32F429xI/TARGET_NUCLEO_F429ZI/PeripheralPinMaps.h

[7] "Smart City Bike Lights | Mbed". Accessed July 9, 2021. https://os.mbed.com/built-with-mbed/smart-city-bike-lights/

# Chapter 9

File Storage on SD Cards and
Usage of Software Repositories

# 9.1 Roadmap

## 9.1.1 What You Will Learn

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

- Describe how to connect an SD card to the NUCLEO board using an SPI bus interface.
- Develop programs to manage files on an SD card with the NUCLEO board.
- Implement a revision control system using repositories.
- Summarize the fundamentals of filesystems.

## 9.1.2 Review of Previous Chapters

In the previous chapters, many sensors and actuators were included in the smart home system. Different events were detected, for example over temperature or the presence of intruders. These events were reported using the serial terminal, as well as the display, a range of lights, and the siren, which was simulated using a buzzer. However, once the system is turned off, there is no record of the events that took place. This can be inconvenient, as after a fire alarm or an intruder detection, the system manager might wish to analyze what happened even if there was a power outage in between.

## 9.1.3 Contents of This Chapter

In this chapter, an SD card (*Secure Digital memory card*) is used to store a copy of the events log of the smart home system. In this way, events can be recorded over time on the SD card, even after turning off the power supply of the smart home system. In addition, the files stored in the SD card can be read by any device provided with an SD card reader, such as a PC or a smartphone.

For this purpose, the concept of a *filesystem* will be introduced as a way to organize the storage capacity of the SD card into *files* and *folders*. The files on the SD card will be able to be created, written, read, modified, and deleted.

# 9.2 File Storage with the NUCLEO Board

## 9.2.1 Connect an SD Card to the Smart Home System

In this chapter, a micro-SD card, such as the one described in [1], is connected to the smart home system, as shown in Figure 9.1. The aim of this setup is to store the events log on the SD card (for the sake of brevity, we use "SD card" to refer to the micro-SD card).

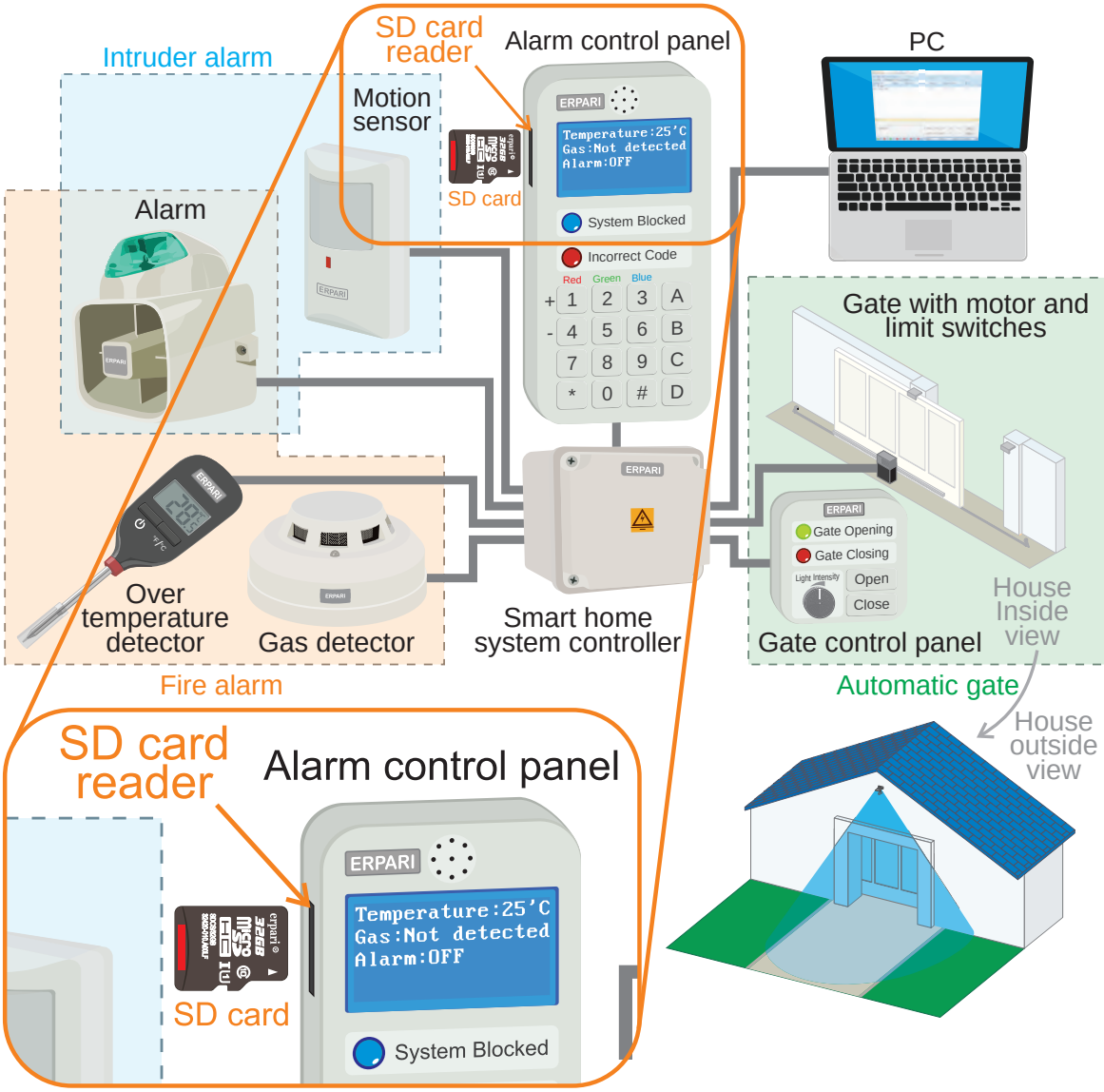



*Figure 9.1 The smart home system is now connected to an SD card.*

The connections that should be made are shown in Figure 9.2 and summarized in Table 9.1. The reader may notice that an SPI bus is used to connect the NUCLEO board with the SD card.

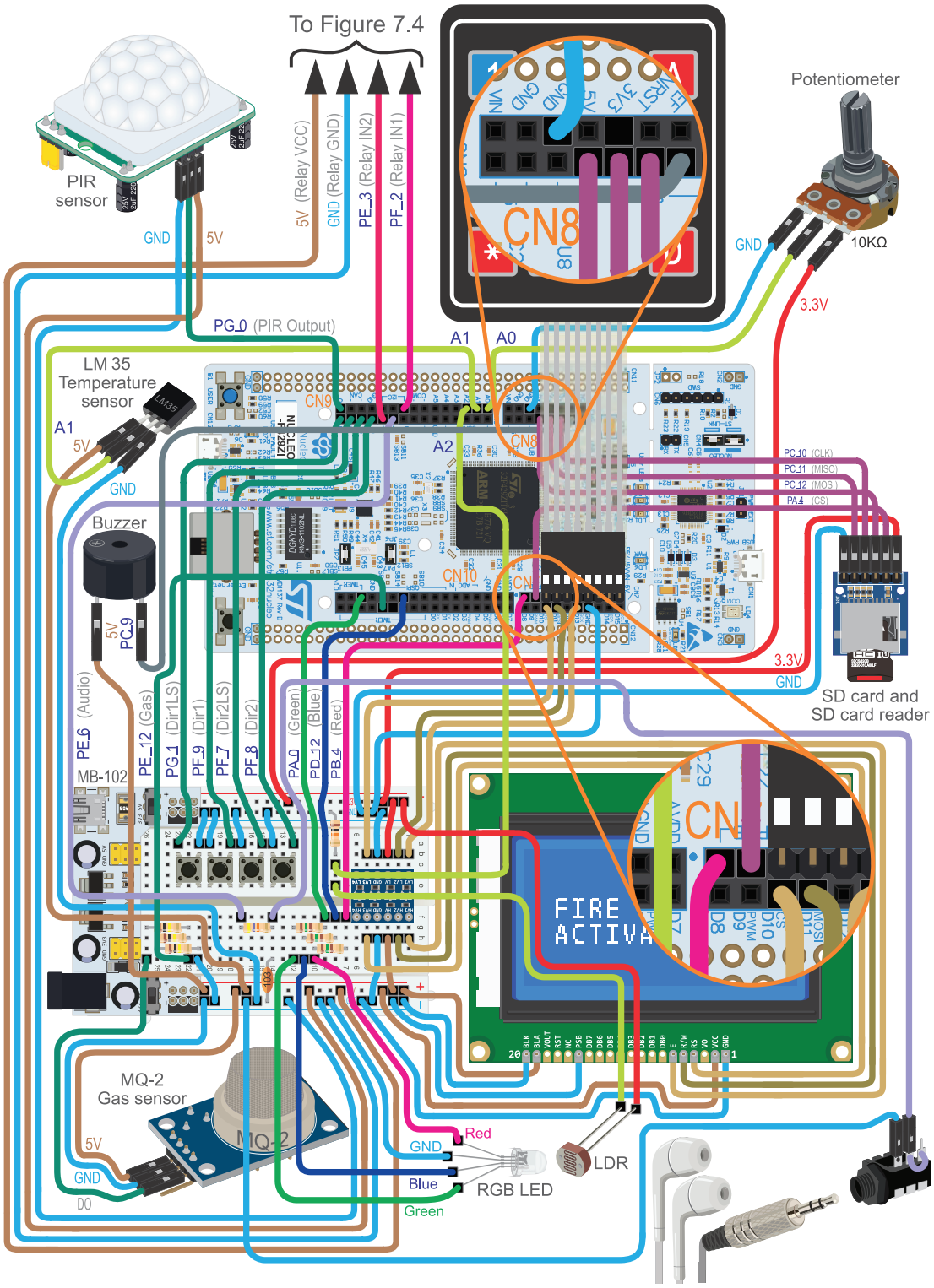



*Figure 9.2 The smart home system is now connected to an SD card.*

*Table 9.1 Summary of the connections between the NUCLEO board and the SD card.*

| NUCLEO board | SD card |
|---|---|
| 3.3 V | 3V3 |
| PA_4 | CS |
| PC_12 | MOSI |
| PC_10 | CLK |
| PC_11 | MISO |
| GND | GND |

**NOTE:** In Figure 8.5 it can be seen that PC_10, PC_11, PC_12, and PA_4 correspond to SPI3_SCK, SPI3_MISO, SPI3_MOSI, and SPI3_CS, respectively. In Chapter 6, in order to connect the graphical display, the pins PA_5, PA_6, PA_7, and PD_14 (SPI1_SCK, SP1_MISO, SPI1_MOSI, and SPI1_CS) were used. The graphical LCD display does not ignore the SPI MOSI and SCK signals even if its CS signal is inactive. Therefore, if the same SPI bus was used for both devices, the display would show glitches every time the program used the SD card.

In Figure 9.3, the details of the SD card module pins are shown. It can be seen that they correspond to the SPI bus signals that were introduced in section 6.2.4. Figure 9.3 also shows how to properly insert the SD card into the SD card module.


*Figure 9.3 Details of the SD card module pins and how to insert the SD card into the module.*

**WARNING:** It is important to use an SD card properly formatted as FAT32. To format the SD card, it is recommended to use a notebook provided with an SD card slot and the *format tool* of its operating system.

**WARNING:** Some SD card modules have a different pinout. Follow the 3.3V, CS, MOSI, CLK, MISO and GND labels of the module when making the connections. Be sure to use a module that can be powered using 3.3 V and that supports the SD card memory size you are using.

To test if the SD card is working correctly, the *.bin* file of the program "Subsection 9.2.1" should be downloaded from the URL available in [2] and loaded onto the NUCLEO board. Press "w" on the PC keyboard to create a file called *Hello.txt* on the SD card connected to the NUCLEO board. Then press "*l*" to get a list of all the files in the root directory of the SD card. The file *Hello.txt* should be in the listing, as well as other files and folders contained on the SD card.

**TIP:** Ignore all the other elements of the setup during the proposed test (Alarm LED, display, etc.).

### 9.2.2 A Filesystem to Control how Data is Stored and Retrieved

Data are usually stored on devices such as SD cards or hard disk drives. Without appropriate organization, the data placed on those devices would be one single body of useless bits with no way to determine where each element of data begins and ends. In order to organize the data, a *filesystem* is used.

In a filesystem, each group of data is called a *file*. The logical rules and structure used to manage the data and their names is called a filesystem. Typically, a filesystem consists of two or three layers, as shown in Figure 9.4. The *logical layer* provides access to files and directories, among other operations. It is usually accessed by means of an *Application Programming Interface* (API), using functions such as *open*, *read*, *write*, *close*, etc. The reader might notice that the modularization implemented in Chapter 5 and many of the names of the functions introduced in that chapter follow this logic. Examples include *eventLogRead()* and *eventLogWrite()*.

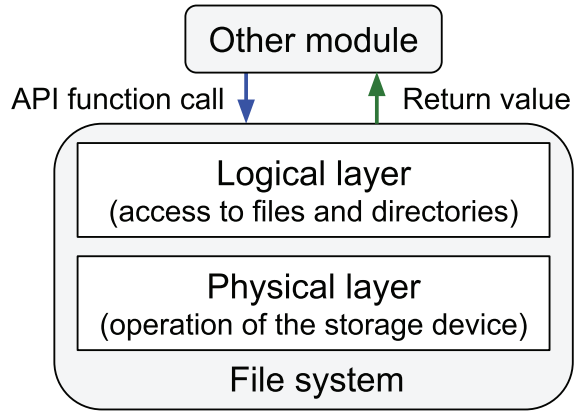



***Figure 9.4 Diagram of a typical organization of a filesystem.***

The second layer shown in Figure 9.4 is the *physical layer*. This layer is concerned with the physical operation of the storage device, for example an SD card. It is responsible for the physical placement of data in specific locations on the storage medium, and for retrieving the data when needed.

The filesystem is responsible for organizing files and directories and keeping track of which areas of the media belong to which file and which are not being used. The data are usually stored in *allocation units* of a given size. This results in unused space when a file is not an exact multiple of the allocation unit. Choosing an allocation size based on the average size of the files expected to be in the filesystem can minimize the amount of unusable space.

There are different kinds of filesystems, each having different advantages and disadvantages. The family of *File Allocation Table* (FAT) filesystems is simpler than other technologies and is supported by almost all operating systems for personal computers. For that reason, it is used in this chapter to store data on the SD card. It is based on a table (the file allocation table) stored on the device in which the areas associated with each file are identified.

### Example 9.1: Create a File with the Event Log on the SD Card

#### Objective

Introduce the usage of filesystems and repositories.

#### Summary of the Expected Behavior

When pressing the "w" key on the PC keyboard, a *.txt* text file with a copy of the current event log (containing up to twenty events) should be created on the SD card.

#### Test the Proposed Solution on the Board

Import the project "Example 9.1" using the URL available in [2], 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. Use the Fire alarm test button (B1 button) to activate the alarm. Press "e" on the PC keyboard to get the date and time of the alarm activation. Press "w" on the PC keyboard to write a file with the events log onto the SD card connected to the NUCLEO board. A message indicating that the file was successfully written on the SD card should be shown on the serial terminal, as shown in Figure 9.5.

```
Storing event 1 in file 2021_07_09_05_25_26.txt
Storing event 2 in file 2021_07_09_05_25_26.txt
Storing event 3 in file 2021_07_09_05_25_26.txt
Storing event 4 in file 2021_07_09_05_25_26.txt
File successfully written
```

***Figure 9.5 Example of events storage messages.***

**NOTE:** Jan 1, 1970, 00:00 hours will be the date and time if the RTC is not configured, as was explained in Chapter 4.

The content of the event file after storing the four events corresponding to Figure 9.5 is shown in Figure 9.6. It can be seen that the content of the *.txt* file shown in Figure 9.6 is the same as the message that is shown on the serial terminal if the letter "e" is pressed on the PC keyboard. This *.txt* file can be opened using a PC or a smartphone if the SD card is inserted into those devices, provided they are properly formatted as FAT32.

```
Event = ALARM_ON
Date and Time = Fri Jul 9 05:25:24 2021

Event = GAS_DET_ON
Date and Time = Fri Jul 9 05:25:24 2021

Event = GAS_DET_OFF
Date and Time = Fri Jul 9 05:25:25 2021

Event = ALARM_OFF
Date and Time = Fri Jul 9 05:25:25 2021
```

***Figure 9.6 Example of the content of an event file stored on the SD card as a .txt file.***

## Discussion of the Proposed Solution

The proposed solution is based on a new module named *sd_card*. This module is composed of two files, *sd_card.cpp* and *sd_card.h*, following the modularized structure discussed in previous chapters. In addition, some Mbed OS libraries to manage FAT filesystems and SD devices that are available in [3] are included in *sd_card.cpp* (*FATFileSystem.h*, *SDBlockDevice.h*, and *mbed_retarget.h*). Also, the *mbed_app.json* file is used by the Mbed OS to configure the SD card, so it was modified as shown in Code 9.1.

```
{
    "target_overrides": {
        "*": {
            "platform.stdio-convert-newlines": 1,
            "target.features_add": ["STORAGE"],
            "target.components_add": ["SD"],
            "sd.INIT_FREQUENCY": 350000,
            "target.printf_lib": "std"
        }
    }
}
```

***Code 9.1 Content of the mbed_app.json file.***

The line "target.printf_lib": "std" was already in the *mbed_app.json* file that was introduced in Chapter 3. By means of the other parameters that are now on the *mbed_app.json* file, the SD card is configured. For more information, see [3].

## Implementation of the Proposed Solution

In Table 9.2 and Table 9.3, the sections where lines were added to *pc_serial_com.h* and *pc_serial_com.cpp* are shown. It can be seen that the library sd_card.h is now included, as well as the new functions *pcSerialComIntWrite()* and *commandEventLogSaveToSdCard()*. In Table 9.4, the lines that were added in *pcSerialComCommandUpdate()* and *availableCommands()* in order to implement the "w" command are shown.

*Table 9.2 Sections in which lines were added to pc_serial_com.h.*

| Section | Lines that were added |
|---|---|
| Declarations (prototypes) of public functions | `void pcSerialComIntWrite( int number );` |

*Table 9.3 Sections in which lines were added to pc_serial_com.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "sd_card.h"` |
| Declarations (prototypes) of private functions | `static void commandEventLogSaveToSdCard();` |

*Table 9.4 Functions in which lines were added in pc_serial_com.cpp.*

| Section | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'w': case 'W': commandEventLogSaveToSdCard(); break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'w' or 'W' to store the events log in the SD card\r\n" );` |

In Code 9.2, the implementation of the function *pcSerialComIntWrite()* is shown. This function is used to show on the serial terminal the number of the event that has been stored on the SD card.

```
1  void pcSerialComIntWrite( int number )
2  {
3      char str[4] = "";
4      sprintf( str, "%d", number );
5      pcSerialComStringWrite( str );
6  }
```

*Code 9.2 Implementation of the function pcSerialComIntWrite().*

In Code 9.3, the implementation of the function *commandEventLogSaveToSdCard()* is shown. This function is called when the "w" key is pressed on the PC keyboard. As can be seen in Code 9.3, it only calls the function *eventLogSaveToSdCard()*.

```
1  static void commandEventLogSaveToSdCard()
2  {
3      eventLogSaveToSdCard();
4  }
```

*Code 9.3 Implementation of the function commandEventLogSaveToSdCard().*

In Table 9.5 and Table 9.6, the sections in which lines were added to *event_log.h* and *event_log.cpp* are shown. It can be seen that *commandEventLogSaveToSdCard()* has been declared as a public function, and that the library sd_card has been included.

***Table 9.5 Sections in which lines were added to event_log.h.***

| Section | Lines that were added |
|---|---|
| Declarations (prototypes) of public functions | `bool eventLogSaveToSdCard();` |

***Table 9.6 Sections in which lines were added to event_log.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "sd_card.h"` |

In Code 9.4, the implementation of the new function *eventLogSaveToSdCard()* is shown. On lines 3 and 4, two char arrays, *fileName* and *eventStr* are declared and initialized. The former is used to store the name of the file, while the latter is used to store a string corresponding to the event to be stored. The Boolean variable *eventsStored* on line 5 is used to indicate whether events have been stored. On lines 7 and 8, two more variables that are used in this example are declared, *seconds* and *i*.

On line 10, the RTC of the NUCLEO board is read, and the current time is stored in the variable *seconds*. On line 12, the name of the new file to be created is generated by means of the function *strftime()* provided by Mbed OS. For this purpose, the value of the variable *seconds* is used. This value is converted into the "YYYY MM DD HH MM SS" format by means of the function *localtime()*. The value of SD_CARD_FILENAME_MAX_LENGTH (that is defined in *sd_card.h*) is used to limit the number of characters that are used in *fileName*. On line 14, the file extension *.txt* is appended to the fileName by means of s*trcat()*.

The *for* loop on line 16 is used to read all the events that are stored in memory, and one after the other these events are stored in the string *eventStr* (line 17).

On line 18, the function *sdCardWriteFile()* is called to write the event log onto the SD card. This function receives two parameters, the filename and a string corresponding to the event to be stored. There is an *if* statement on line 18 as the function *sdCardWriteFile()* returns true if it can successfully write the event in the file, and false if not.

Lines 19 to 23 are used to display a message on the serial terminal showing the event that has been written, as shown in Figure 9.5.

Note that if there are no events in the log, then the function *eventLogNumberOfStoredEvents()* returns 0 on line 16 and, therefore, the *for* loop is never executed. If this is the case, the variable *eventsStored* remains false and the message "There are no events to store or SD card is not available" is shown, as can be seen on line 31 of Code 9.4.

```
1  bool eventLogSaveToSdCard()
2  {
3      char fileName[SD_CARD_FILENAME_MAX_LENGTH] = "";
4      char eventStr[EVENT_STR_LENGTH] = "";
5      bool eventsStored = false;
6
7      time_t seconds;
8      int i;
9
10     seconds = time(NULL);
11
12     strftime( fileName, SD_CARD_FILENAME_MAX_LENGTH,
13               "%Y_%m_%d_%H_%M_%S", localtime(&seconds) );
14     strcat( fileName, ".txt" );
15
16     for (i = 0; i < eventLogNumberOfStoredEvents(); i++) {
17         eventLogRead( i, eventStr );
18         if ( sdCardWriteFile( fileName, eventStr ) ){
19             pcSerialComStringWrite("Storing event ");
20             pcSerialComIntWrite(i+1);
21             pcSerialComStringWrite(" in file ");
22             pcSerialComStringWrite(fileName);
23             pcSerialComStringWrite("\r\n");
24             eventsStored = true;
25         }
26     }
27
28     if ( eventsStored ) {
29         pcSerialComStringWrite("File successfully written\r\n\r\n");
30     } else {
31         pcSerialComStringWrite("There are no events to store ");
32         pcSerialComStringWrite("or SD card is not available\r\n\r\n");
33     }
34
35     return true;
36 }
```

***Code 9.4 Implementation of the function eventLogSaveToSdCard().***

Code 9.5 shows the content of the *sd_card.h* file. On line 8 it can be seen that the #define SD_CARD_FILENAME_MAX_LENGTH that is used in the function *eventLogSaveToSdCard()* has the value 32. It can also be seen on lines 12 and 13 that two public functions are declared, *sdCardInit()* and *sdCardWriteFile()*.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _SD_CARD_H_
4  #define _SD_CARD_H_
5
6  //=====[Libraries]=============================================================
7
8  #define SD_CARD_FILENAME_MAX_LENGTH 32
9
10 //=====[Declarations (prototypes) of public functions]=========================
11
12 bool sdCardInit();
13 bool sdCardWriteFile( const char* fileName, const char* writeBuffer );
14
15 //=====[#include guards - end]=================================================
16
17 #endif // _SD_CARD_H_
```

***Code 9.5 Content of the sd_card.h file.***

In Code 9.6, the first part of the file *sd_card.cpp* is shown. From lines 3 to 15 all the libraries that are used in the *sd_card* module can be seen. It can also be seen that *mbed_retarget.h*, *FATFileSystem.h*, and *SDBlockDevice.h* are included. These are libraries that are used to work with files on the SD card.

On lines 19 to 22 the pins used to connect the SD card are defined. These pins are related to SPI3, and it is important to note that an alternative pin (PA_4_ALT0) is used on line 22 for the *chip select* pin (SPI3_CS).

**NOTE:** It is not necessary to use the SPI port-specific pin for chip select; a DigitalOut can be used and controlled by software. In Chapter 6, PD_14 was used instead of pin PA_4, which is the SPI1 port-specific chip select pin. In this case, the specific pin for SPI3 is used in order to show a usage example of the alternative pins that were introduced in Chapter 8. More information about alternative pins and port-specific pins for peripherals can be found in [4] and [5].

Finally, on lines 26 and 28, two objects are declared; the former is used in the communication with the SD card, and the latter is used to implement the filesystem.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "sd_card.h"
7
8  #include "event_log.h"
9  #include "date_and_time.h"
10 #include "pc_serial_com.h"
11
12 #include "FATFileSystem.h"
13 #include "SDBlockDevice.h"
14
15 #include "platform/mbed_retarget.h"
16
17 //=====[Declaration of private defines]========================================
18
19 #define SPI3_MOSI   PC_12
20 #define SPI3_MISO   PC_11
21 #define SPI3_SCK    PC_10
22 #define SPI3_CS     PA_4_ALT0
23
24 //=====[Declaration and initialization of public global objects]===============
25
26 SDBlockDevice sd( SPI3_MOSI, SPI3_MISO, SPI3_SCK, SPI3_CS );
27
28 FATFileSystem sdCardFileSystem("sd", &sd);
```

***Code 9.6 Content of the sd_card.cpp file (Part 1/2).***

**NOTE:** An SPI object is not declared as in Chapter 6 despite the fact that the SD card is connected to the NUCLEO board using an SPI bus. Recall that in Chapter 6 the objects *SPI spiSt7920(SPI1_MOSI, SPI1_MISO, SPI1_SCK)* and *DigitalOut spiSt7920ChipSelect(SPI1_CS)* were declared, the SPI bus was configured by means of *SPI_ST9720.format(8,3)* and *SPI_ST9720.frequency(1000000)*, and this was used by *SPI_ST9720.write()*. In this chapter, the objects *SDBlockDevice sd(SPI3_MOSI, SPI3_MISO, SPI3_SCK, SPI3_CS)* and *FATFileSystem sdCardFileSystem("sd"*, &sd) are declared to control the SD card using the SPI bus, following the format established by Mbed OS.

In Code 9.7, the second part of the *sd_card.cpp* file is shown. Lines 1 to 16 show the implementation of the function *sdCardInit()*. It can be seen that a message is first displayed indicating that it is looking for a filesystem (line 3). On line 4, *mount()* is used to get the information of the FAT filesystem of the SD card into the object *sdCardFileSystem*. On line 5, the function *opendir()*, which is declared and implemented in the Mbed OS, is used to get the list of directories on the SD card. If the list of directories can be successfully read, then it is stored in the object *sdCardListOfDirectories*; otherwise NULL is stored in *sdCardListOfDirectories*. On line 6, the value of *sdCardListOfDirectories* is checked in order to determine if there is a FAT filesystem mounted on the SD card. If so, the message shown on line 7 is shown and the directory is closed by means of line 8.

```
 1  bool sdCardInit()
 2  {
 3      pcSerialComStringWrite("Looking for a filesystem in the SD card... \r\n");
 4      sdCardFileSystem.mount(&sd);
 5      DIR *sdCardListOfDirectories = opendir("/sd/");
 6      if ( sdCardListOfDirectories != NULL ) {
 7          pcSerialComStringWrite("Filesystem found in the SD card.\r\n");
 8          closedir(sdCardListOfDirectories);
 9          return true;
10      } else {
11          pcSerialComStringWrite("Filesystem not mounted. \r\n");
12          pcSerialComStringWrite("Insert an SD card and ");
13          pcSerialComStringWrite("reset the NUCLEO board.\r\n");
14          return false;
15      }
16  }
17
18  bool sdCardWriteFile( const char* fileName, const char* writeBuffer )
19  {
20      char fileNameSD[SD_CARD_FILENAME_MAX_LENGTH+4] = "";
21
22      fileNameSD[0] = '\0';
23      strcat( fileNameSD, "/sd/" );
24      strcat( fileNameSD, fileName );
25
26      FILE *sdCardFilePointer = fopen( fileNameSD, "a" );
27
28      if ( sdCardFilePointer != NULL ) {
29          fprintf( sdCardFilePointer, "%s", writeBuffer );
30          fclose( sdCardFilePointer );
31          return true;
32      } else {
33          return false;
34      }
35  }
```

***Code 9.7 Content of the sd_card.cpp file (Part 2/2).***

If an appropriate FAT filesystem is not found on the SD card, then lines 10 to 15 are executed in order to indicate this to the user.

**WARNING:** The messages on lines 11 to 13 will be displayed in any of the following situations:

- The SD card is not connected.
- The SD card module is not properly connected.
- The SD card is not working properly (i.e., it is damaged).
- The filesystem format of the SD Card is not FAT32 as expected.

On line 18 of Code 9.7, the implementation of the function *sdCardWriteFile()* is shown. The first parameter that this function receives (*fileName*) is the filename, and the second parameter (*writeBuffer*) is the data to be written. Lines 22 to 24 are used to write the prefix "/sd/" in fileNameSD, which is necessary in order to indicate that this is the root folder of the SD card. On line 26, the object *sdCardFilePointer* of type FILE is declared and is assigned a pointer to the file that is opened by means of *fopen()*. The parameter "a" on line 26 is used to indicate that new content will be appended to the opened file. It is important to note that the use of the "a" parameter implies that if the file doesn't exist, then it must be created.

**NOTE:** Recall that a pointer is an object that stores a memory address, usually corresponding to a variable. Pointers will be discussed in detail in Chapter 10 and an example will be implemented using pointers.

Line 28 assesses whether the file was correctly opened and, if so, the content of *writeBuffer* is written to the file by means of *fprintf()* (line 29). Then, the file is closed (line 30) and the *Boolean* value true is returned (line 31). If the content of *writeBuffer* was not successfully appended to the file, then false is returned (line 33).

Lastly, in Code 9.8 the new implementation of *smartHomeSystemInit()* is shown. It can be seen that *sdCardInit()* was added on line 12.

```
 1  void smartHomeSystemInit()
 2  {
 3      audioInit();
 4      userInterfaceInit();
 5      alarmInit();
 6      fireAlarmInit();
 7      intruderAlarmInit();
 8      pcSerialComInit();
 9      motorControlInit();
10      gateInit();
11      lightSystemInit();
12      sdCardInit();
13  }
```

***Code 9.8 New implementation of the function smartHomeSystemInit.***

### Proposed Exercise

1. What should be modified in order to connect the SD card to a different set of pins of the NUCLEO board?

### Answer to the Exercise

1. The pins' assignment in *sd_card.cpp* (line 28 of Code 9.6) should be modified to use the SPI_4. For example, the following assignment can be used:

```
#define SPI4_MOSI   PE_14
#define SPI4_MISO   PE_5
#define SPI4_SCK    PE_2
#define SPI4_CS     PE_4

SDBlockDevice sd( SPI4_MOSI, SPI4_MISO, SPI4_SCK, SPI4_CS );
```

## Example 9.2: Save a File on the SD Card with only New Events that were not Previously Saved

### Objective

Introduce functionality regarding the management of data storage in files.

### Summary of the Expected Behavior

When pressing the key "w" on the PC keyboard, a *.txt* file with a copy of events that were not saved previously is stored on the SD card. In this way, multiple copies of the same events in different files are avoided.

### Test the Proposed Solution on the Board

Import the project "Example 9.2" using the URL available in [2], build the project, and drag the *.bin* file onto the NUCLEO board. Repeat the steps in Example 9.1 in order to write a log file onto the SD card such as the one shown in Figure 9.6. Use the Fire alarm test button (B1 button) to activate the alarm again. Press "e" on the PC keyboard to get the date and time of the alarm activation. Press "w" on the PC keyboard to write the events log onto the SD card. A message indicating that the file was successfully written on the SD card with only the new events that were not previously stored should be shown on the serial terminal, as in Figure 9.7.

```
Storing event 5 in file 2021_07_09_08_12_54.txt
Storing event 6 in file 2021_07_09_08_12_54.txt
Storing event 7 in file 2021_07_09_08_12_54.txt
Storing event 8 in file 2021_07_09_08_12_54.txt
New events successfully stored in the SD card
```

***Figure 9.7 Example of events storage messages.***

The content of the event file after storing the new events corresponding to Figure 9.7 is shown in Figure 9.8. The *.txt* file has only the new events corresponding to the second time the Fire alarm test

button was pressed (i.e., event 5 to event 8). Note that in Example 9.1, a new file was created each time the "w" key was pressed on the PC keyboard, logging all events, not only new events.

```
Event = ALARM_ON
Date and Time = Fri Jul 9 08:12:51 2021

Event = GAS_DET_ON
Date and Time = Fri Jul 9 08:12:51 2021

Event = GAS_DET_OFF
Date and Time = Fri Jul 9 08:12:53 2021

Event = ALARM_OFF
Date and Time = Fri Jul 9 08:12:53 2021
```

***Figure 9.8 Example of the content of an event file stored on the SD card as a .txt file.***

Finally, press "w" again. It will show a message "No new events to store in SD card." In this way, it is seen that now a file is created only if there are events that were not previously stored.

## Discussion of the Proposed Solution

Due to the modularization that has been followed in the program structure, the proposed modification can be done by means of only changing the *event_log.cpp* file. The corresponding details are shown below.

## Implementation of the Proposed Solution

In order to identify which events were already stored on the SD card, a new *member* is included in the data structure *systemEvent_t*, which is defined in the *event_log.cpp* file. This member is named *storedInSd*, as shown in Code 9.9. The value *storedInSd* of an element will be set as false if the corresponding event was not stored on the SD card and will be set to true when it has been successfully stored on the SD card.

```
1  typedef struct systemEvent {
2      time_t seconds;
3      char typeOfEvent[EVENT_LOG_NAME_MAX_LENGTH];
4      bool storedInSd;
5  } systemEvent_t;
```

***Code 9.9 New declaration of the data structure systemEvent_t.***

Code 9.10 shows the new implementation of the function *eventLogWrite()*. It may be noted that only line 14 has been incorporated. This line is used to set to false the Boolean member *storedInSd* of the corresponding element of *arrayOfStoredEvents* (indicated by *eventsIndex*) each time a new event is added to the log.

The new implementation of *eventLogSaveToSdCard()* is shown in Code 9.11. An *if* statement has been included on line 17 in order to evaluate the value of *arrayOfStoredEvents[i].storedInSd*. If the value is false, then the event is stored on the SD card and then is set to true (line 20) in order to avoid storing it again in the future. The messages on lines 32 and 34 were also modified to indicate whether events were stored or not.

```
void eventLogWrite( bool currentState, const char* elementName )
{
    char eventAndStateStr[EVENT_LOG_NAME_MAX_LENGTH] = "";

    strcat( eventAndStateStr, elementName );
    if ( currentState ) {
        strcat( eventAndStateStr, "_ON" );
    } else {
        strcat( eventAndStateStr, "_OFF" );
    }

    arrayOfStoredEvents[eventsIndex].seconds = time(NULL);
    strcpy( arrayOfStoredEvents[eventsIndex].typeOfEvent, eventAndStateStr );
    arrayOfStoredEvents[eventsIndex].storedInSd = false;
    if ( eventsIndex < EVENT_LOG_MAX_STORAGE - 1 ) {
        eventsIndex++;
    } else {
        eventsIndex = 0;
    }

    pcSerialComStringWrite(eventAndStateStr);
    pcSerialComStringWrite("\r\n");
}
```

*Code 9.10 New implementation of the function eventLogWrite().*

```
bool eventLogSaveToSdCard()
{
    char fileName[SD_CARD_FILENAME_MAX_LENGTH] = "";
    char eventStr[EVENT_STR_LENGTH] = "";
    bool eventsStored = false;

    time_t seconds;
    int i;

    seconds = time(NULL);

    strftime( fileName, SD_CARD_FILENAME_MAX_LENGTH,
              "%Y_%m_%d_%H_%M_%S", localtime(&seconds) );
    strcat( fileName, ".txt" );

    for (i = 0; i < eventLogNumberOfStoredEvents(); i++) {
        if ( !arrayOfStoredEvents[i].storedInSd ) {
            eventLogRead( i, eventStr );
            if ( sdCardWriteFile( fileName, eventStr ) ){
                arrayOfStoredEvents[i].storedInSd = true;
                pcSerialComStringWrite("Storing event ");
                pcSerialComIntWrite(i+1);
                pcSerialComStringWrite(" in file ");
                pcSerialComStringWrite(fileName);
                pcSerialComStringWrite("\r\n");
                eventsStored = true;
            }
        }
    }

    if ( eventsStored ) {
        pcSerialComStringWrite("New events successfully stored in the SD card\r\n\r\n");
    } else {
        pcSerialComStringWrite("No new events to store in the SD card\r\n\r\n");
    }
    return true;
}
```

*Code 9.11 New implementation of the function eventLogSaveToSdCard().*

### Proposed Exercise

1. How can the code be modified in order to name the files using the year, month, and day?

### Answer to the Exercise

1. Line 13 of Code 9.11 can be replaced by:

```
strftime( fileName, SD_CARD_FILENAME_MAX_LENGTH,
        "%Y_%m_%d_%H_%M_%S", localtime(&seconds));
```

## Example 9.3: Get the List of Event Log Files Stored on the SD Card

### Objective

Introduce more advanced functionality of the filesystem.

### Summary of the Expected Behavior

When pressing the "l" key on the PC keyboard, a list of the event log files stored on the SD card should be displayed on the serial monitor.

### Test the Proposed Solution on the Board

Import the project "Example 9.3" using the URL available in [2], build the project, and drag the *.bin* file onto the NUCLEO board. Press "l" on the PC keyboard to get the list of all the event log files stored on the SD card. A message similar to the one shown in Figure 9.9 should be displayed on the serial monitor.

```
Printing all filenames:
hello.txt
image.jpg
1970_01_01_00_04_13.txt
2021_07_09_05_25_26.txt
2021_07_09_08_12_54.txt
```

***Figure 9.9 Example of the file listing that is shown on the PC.***

### Discussion of the Proposed Solution

The proposed solution is based on the function *sdCardListFiles()*, which is used to retrieve the listing of the files that are stored on the SD card. The list of the files is then shown on the serial monitor.

## Implementation of the Proposed Solution

Table 9.7 shows that the private function *commandSdCardListFiles()* was added to *pc_serial_com.cpp*. In Table 9.8, the lines that were added in *pcSerialComCommandUpdate()* and *availableCommands()* in order to implement the "l" command are shown.

***Table 9.7 Sections in which lines were added to pc_serial_com.cpp.***

| Section | Lines that were added |
|---|---|
| Declarations (prototypes) of private functions | `static void commandSdCardListFiles();` |

***Table 9.8 Functions in which lines were added in pc_serial_com.cpp.***

| Section | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'l': case 'L': commandSdCardListFiles(); break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'l' or 'L' to list all the files " );`<br>`pcSerialComStringWrite( "in the root directory of the SD card\r\n" );` |

In Code 9.12, the implementation of the function *commandSdCardListFiles()* is shown. On line 3, the array of char *fileListBuffer* is declared with the appropriate size to store a list of up to ten files (SD_CARD_MAX_FILE_LIST is defined as 10) in the situation of all the filenames having a length of 32 bytes (SD_CARD_FILENAME_MAX_LENGTH is defined as 32). On line 4, *sdCardListFiles()* is called in order to load up to a maximum of ten filenames into *fileListBuffer*. On line 6, the list of files is sent to the serial terminal, followed by "\r\n", as can be seen on line 7.

```
1  static void commandSdCardListFiles()
2  {
3      char fileListBuffer[SD_CARD_MAX_FILE_LIST*SD_CARD_FILENAME_MAX_LENGTH] = "";
4      sdCardListFiles( fileListBuffer,
5                       SD_CARD_MAX_FILE_LIST*SD_CARD_FILENAME_MAX_LENGTH );
6      pcSerialComStringWrite( fileListBuffer );
7      pcSerialComStringWrite( "\r\n" );
8  }
```

***Code 9.12 Implementation of the function commandSdCardListFiles().***

In Table 9.9, the sections where lines were added to *sd_card.h* are shown. It can be seen that SD_CARD_MAX_FILE_LIST was defined, and the new public function *sdCardListFiles()* has been declared. The corresponding code is shown in Code 9.13.

*Table 9.9 Sections in which lines were added to sd_card.h.*

| Section | Lines that were added |
|---|---|
| Declaration of private defines | `#define SD_CARD_MAX_FILE_LIST 10` |
| Declarations (prototypes) of public functions | `bool sdCardListFiles( char* fileNamesBuffer, int fileNamesBufferSize );` |

```
 1  bool sdCardListFiles( char* fileNamesBuffer, int fileNamesBufferSize )
 2  {
 3      int NumberOfUsedBytesInBuffer = 0;
 4      struct dirent *sdCardDirectoryEntryPointer;
 5
 6      DIR *sdCardListOfDirectories = opendir("/sd/");
 7
 8      if ( sdCardListOfDirectories != NULL ) {
 9          pcSerialComStringWrite("Printing all filenames:\r\n");
10          sdCardDirectoryEntryPointer = readdir(sdCardListOfDirectories);
11
12          while ( ( sdCardDirectoryEntryPointer != NULL ) &&
13                  ( NumberOfUsedBytesInBuffer + strlen(sdCardDirectoryEntryPointer->d_name)
14                      < fileNamesBufferSize) ) {
15              strcat( fileNamesBuffer, sdCardDirectoryEntryPointer->d_name );
16              strcat( fileNamesBuffer, "\r\n" );
17              NumberOfUsedBytesInBuffer = NumberOfUsedBytesInBuffer +
18                                          strlen(sdCardDirectoryEntryPointer->d_name);
19              sdCardDirectoryEntryPointer = readdir(sdCardListOfDirectories);
20          }
21
22          closedir(sdCardListOfDirectories);
23
24          return true;
25      } else {
26          pcSerialComStringWrite("Insert an SD card and ");
27          pcSerialComStringWrite("reset the NUCLEO board.\r\n");
28          return false;
29      }
31  }
```

*Code 9.13 Implementation of the function sdCardListFiles().*

On line 3 of Code 9.13, it can be seen that a variable called *NumberOfUsedBytesInBuffer* is declared and initialized to zero. This variable will be used to keep track of the number of bytes used to store the list of files on the SD card. On line 4, a pointer named *sdCardDirectoryEntryPointer* of the type dirent (directory entry) is declared.

On line 6, the root directory of the SD card is opened by means of the function *opendir()*, and a pointer to it is stored in the object *sdCardListOfDirectories*. Line 8 evaluates if the root directory was successfully opened by means of evaluating the content of *sdCardListOfDirectories*. If the directory was successfully opened, then the message on line 9 is shown on the serial monitor. Then, the list of files in the directory is retrieved by means of *readdir()* on line 10 and stored in *sdCardDirectoryEntryPointer*.

The *while* loop shown on line 12 is used to (i) check that *sdCardDirectoryEntryPointer* is not NULL (in that case it implies that all the files were already explored) and (ii) that the memory usage is below the limit given by the expression *"(bufferNumberUsedBytes + strlen(sdCardDirectoryEntryPointer->d_name)*

*< fileNamesBufferSize)"*. This expression assesses if the number of bytes that has already been used (*NumberOfUsedBytesInBuffer*) plus the size in bytes of the next filename to be read (given by *strlen(sdCardDirectoryEntryPointer->d_name)*) is smaller than the maximum number of available bytes (*fileNamesBufferSize*).

**NOTE:** The (->) arrow operator is used to dereference the address a pointer contains (in this example sdCardDirectoryEntryPointer points to the next directory entry) to get or set the value stored in the variable itself. This operator will be discussed in detail in Chapter 10.

On lines 15 and 16, the file names are written one after the other into *fileNamesBuffer*. On line 17, the value of the *NumberOfUsedBytesInBuffer* is updated by means of adding the size of the filename (i.e., *strlen(sdCardDirectoryEntryPointer->d_name)*) to be appended to the list of files. On line 19, the pointer *sdCardDirectoryEntryPointer* is pointed to the next file in the directory in order to be ready to start a new iteration of the *while* loop.

After finishing the *while* loop, the directory is closed on line 22 and the value *true* is returned on line 24.

If the statement on line 8 is false (i.e., the root directory was not successfully opened), then the statements on lines 26 and 27 are executed in order to instruct the user to insert an SD card, just like in the previous examples.

### Proposed Exercise

1. What will happen if the SCK signal of the SD card module is disconnected before pressing the "l" key?

### Answer to the Exercise

1. The message "Insert an SD card and reset the board." will be shown on the PC screen.

**Example 9.4: Choose and Display One of the Event Log Files Stored on the SD Card**

### Objective

Introduce more advanced functionality regarding the filesystem.

### Summary of the Expected Behavior

When pressing the "o" key on the PC keyboard, a filename can be entered. After pressing the Enter key on the PC keyboard, the content of the corresponding file is shown on the serial terminal.

### Test the Proposed Solution on the Board

Import the project "Example 9.4" using the URL available in [2], build the project, and drag the *.bin* file onto the NUCLEO board. Press "l" on the PC keyboard to get a list of all the event log files stored on the SD card. A message with the listing should be displayed on the PC. Then, press "o" on the PC

keyboard and type in the name of the file that will be opened, and then press Enter. The contents of the file should be shown on the serial terminal.

In Figure 9.10, the result of executing the "o" command is shown for two different situations. First, the name of a file that does exist (*2021_01_27_17_05_06.txt)* is entered and then the name of a file that does not exist (*2021_01_27_17_00_00.txt*) is entered.

```
Please enter the file name
2021_01_27_17_05_06.txt
Opening file: /sd/2021_01_27_17_05_06.txt
The file content is:
Event = ALARM_ON
Date and Time = Wed Jan 27 17:04:47 2021
Event = MOTION_ON
Date and Time = Wed Jan 27 17:04:47 2021
Event = MOTION_OFF
Date and Time = Wed Jan 27 17:04:51 2021
Event = ALARM_OFF
Date and Time = Wed Jan 27 17:04:54 2021

Please enter the file name
2021_01_27_17_00_00.txt
File not found
```

***Figure 9.10 Two examples of opening a file: first, when the file exists, and second, when the file does not exist.***

### Discussion of the Proposed Solution

The proposed solution is based on the function *sdCardReadFile()*, which is used to read the content of a file on the SD card that is opened once its name has been entered by means of the PC keyboard.

### Implementation of the Proposed Solution

In Table 9.10, the sections where lines were added to *pc_serial_com.cpp* are shown. It can be seen that two private global variables are declared: an integer variable named *numberOfCharsInFileName* and a char array named *fileName* of size SD_CARD_FILENAME_MAX_LENGTH (i.e., 32, which was introduced in Code 9.5). In addition, four new private functions are declared: *pcSerialComCharWrite()*, *pcSerialComGetFileName()*, *pcSerialComShowSdCardFile()*, and *commandGetFileName()*. The implementation of these functions is discussed below.

In Table 9.11, the lines that were added in *pcSerialComCommandUpdate()* and *availableCommands()* in order to implement the "o" command are shown.

*Table 9.10 Sections in which lines were added to pc_serial_com.cpp.*

| Section | Lines that were added |
|---|---|
| Declaration and initialization of private global variables | `static int numberOfCharsInFileName = 0;`<br>`static char fileName[SD_CARD_FILENAME_MAX_LENGTH] = "";` |
| Declarations (prototypes) of private functions | `static void pcSerialComCharWrite( char chr );`<br>`static void pcSerialComGetFileName( char receivedChar );`<br>`static void pcSerialComShowSdCardFile( char * fileName );`<br>`static void commandGetFileName();` |

*Table 9.11 Functions in which lines were added in pc_serial_com.cpp.*

| Function | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'o': case 'O': commandGetFileName(); break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'o' or 'O' to show an SD Card file contents\r\n" );` |

The implementation of the function *commandGetFileName()*, which is called when the "o" key is pressed, is presented in Code 9.14. On line 3, it can be seen that a message is shown in order to indicate that a filename should be entered. Then, on line 4, the variable *pcSerialComMode* is assigned by PC_SERIAL_GET_FILE_NAME. This new mode is introduced in the new definition of *pcSerialComMode_t*, as can be seen on line 2 of Code 9.15. The new mode is used in a similar way to the other modes used in *pcSerialComUpdate()*, as can be seen on lines 6 to 8 of Code 9.16.

Finally, it is important to mention that the value of *numberOfCharsInFileName* is set to zero on line 5 of *commandGetFileName()*, as can be seen in Code 9.14. This is done in order to prepare the variable *numberOfCharsInFileName* to receive a new filename.

**NOTE:** When *pcSerialComMode* is in the PC_SERIAL_GET_FILE_NAME mode, the smart home system will not accept new commands until the filename is entered.

```
1  static void commandGetFileName()
2  {
3      pcSerialComStringWrite( "Please enter the file name \r\n" );
4      pcSerialComMode = PC_SERIAL_GET_FILE_NAME ;
5      numberOfCharsInFileName = 0;
6  }
```

*Code 9.14 Implementation of the function commandGetFileName().*

```
1  typedef enum{
2      PC_SERIAL_GET_FILE_NAME,
3      PC_SERIAL_COMMANDS,
4      PC_SERIAL_GET_CODE,
5      PC_SERIAL_SAVE_NEW_CODE,
6  } pcSerialComMode_t;
```

*Code 9.15 New declaration of the type definition pcSerialComMode_t.*

```
 1  void pcSerialComUpdate()
 2  {
 3      char receivedChar = pcSerialComCharRead();
 4      if( receivedChar != '\0' ) {
 5          switch ( pcSerialComMode ) {
 6              case PC_SERIAL_GET_FILE_NAME:
 7                  pcSerialComGetFileName( receivedChar );
 8              break;
 9              case PC_SERIAL_COMMANDS:
10                  pcSerialComCommandUpdate( receivedChar );
11              break;
12              case PC_SERIAL_GET_CODE:
13                  pcSerialComGetCodeUpdate( receivedChar );
14              break;
15              case PC_SERIAL_SAVE_NEW_CODE:
16                  pcSerialComSaveNewCodeUpdate( receivedChar );
17              break;
18              default:
19                  pcSerialComMode = PC_SERIAL_COMMANDS;
20              break;
21          }
22      }
23  }
```

***Code 9.16 New implementation of the function pcSerialComUpdate().***

In Code 9.17, the implementation of the function *pcSerialComGetFileName()*, which is called on line 7 of the function *pcSerialComUpdate()*, is shown. On lines 3 and 4, it can be seen that the entered character is checked to find out if it is '\r' (i.e., the "Enter" key on the PC keyboard), and it is also checked to see if the length of the filename is smaller than the maximum filename length. If so, then PC_SERIAL_COMMANDS is assigned to *pcSerialComMode* in order to be ready to receive new commands. Then, a null character is written at the last position of fileName in order to finalize the string (line 6), and *numberOfCharsInFileName* is set to zero in order to be ready to get a new filename the next time the function *pcSerialComGetFileName()* is called. Finally, the *pcSerialComShowSdCardFile()* function is used to display the contents of the file on the serial terminal.

In the event that the key pressed is not "Enter" and the length of the filename is smaller than the maximum filename length, the *else* statement shown on line 9 of Code 9.17 is executed. It can be seen that the received character is stored in the last position of fileName (line 10), the received character is printed on the serial terminal (line 11), and then the number of characters in the filename is incremented.

```
1  static void pcSerialComGetFileName( char receivedChar )
2  {
3      if ( (receivedChar == '\r') &&
4           (numberOfCharsInFileName < SD_CARD_FILENAME_MAX_LENGTH) ) {
5          pcSerialComMode = PC_SERIAL_COMMANDS;
6          fileName[numberOfCharsInFileName] = '\0';
7          numberOfCharsInFileName = 0;
8          pcSerialComShowSdCardFile( fileName );
9      } else {
10         fileName[numberOfCharsInFileName] = receivedChar;
11         pcSerialComCharWrite( receivedChar );
12         numberOfCharsInFileName++;
13     }
14 }
```

***Code 9.17 Implementation of the function pcSerialComGetFileName().***

The implementation of *pcSerialComShowSdCardFile()* is shown in Code 9.18. On line 3, the array of char *fileContentBuffer* is declared with the appropriate size to store a file with twenty events, as discussed below. Then, "\r\n" is written to the PC screen in order to start a new line. After this, the new function *sdCardReadFile()* that is included in the *sd_card* module is called. This function copies the content of the file whose name is indicated by *fileName* to *fileContentBuffer*, up to a maximum number of bytes indicated by its third parameter, EVENT_STR_LENGTH*EVENT_LOG_MAX_STORAGE (line 6). If the file exists, then *sdCardReadFile()* returns true, lines 7 to 9 are executed, and the file content is shown on the serial monitor. If that file doesn't exist, then *sdCardReadFile()* returns false, and lines 7 to 9 are not executed.

```
1  static void pcSerialComShowSdCardFile( char* fileName )
2  {
3      char fileContentBuffer[EVENT_STR_LENGTH*EVENT_LOG_MAX_STORAGE] = "";
4      pcSerialComStringWrite( "\r\n" );
5      if ( sdCardReadFile( fileName, fileContentBuffer,
6                           EVENT_STR_LENGTH*EVENT_LOG_MAX_STORAGE ) ) {
7          pcSerialComStringWrite( "The file content is:\r\n");
8          pcSerialComStringWrite( fileContentBuffer );
9          pcSerialComStringWrite( "\r\n" );
10     }
11 }
```

***Code 9.18 Implementation of the function pcSerialComShowSdCardFile().***

The implementation of *pcSerialComCharWrite()* is shown on Code 9.19. Note that its parameter, *chr*, is of type char, instead of the parameter of the function *pcSerialComStringWrite()*, which is of type const char*, as was discussed in previous chapters. Because of this, *pcSerialComStringWrite()* cannot be used on line 11 of Code 9.17 instead of *pcSerialComCharWrite()*.

```
1  static void pcSerialComCharWrite( char chr )
2  {
3      char str[2] = "";
4      sprintf (str, "%c", chr);
5      uartUsb.write( str, strlen(str) );
6  }
```

***Code 9.19 Implementation of the function pcSerialComCharWrite().***

In Table 9.12, the lines that were added to *sd_card.h* are shown. It can be seen that the public function *sdCardReadFile()* has been included.

***Table 9.12 Sections in which lines were added to sd_card.h.***

| Section | Lines that were added |
|---|---|
| Declarations (prototypes) of public functions | `bool sdCardReadFile( const char* fileName, char * readBuffer, int readBufferSize );` |

In Code 9.20, the implementation of the function *sdCardReadFile()* as can be seen in *sd_card.cpp* is shown. Lines 3 to 8 are similar to previous examples. The aim of these lines is to append /sd/ to the filename and store this in the string *fileNameSD*. For that reason, this string is declared with four more positions (i.e., +4).

On line 10, the chosen file is opened and a pointer to the file is stored in *sdCardFilePointer*. It should be noted that the parameter "r" is used when the file is opened, which states that the file is to be opened with read privileges only. If the file has been successfully opened, then *sdCardFilePointer* is not NULL and lines 13 to 24 are executed.

By means of lines 13 to 15, the name of the file that is being opened is shown. The *while* loop on line 18 is used to read all of the characters in the file sequentially and copy them to the *readBuffer* array until the end of the file is reached (in that event the return value by *feof(sdCardFilePointer)* becomes true and the *while* loop is finished) or the size of the read buffer has been reached (in that event the *while* loop is finished, too).

The function *fread()* on line 19 has the following four parameters: a pointer to a block of memory where the read bytes are stored (*&readBuffer[i]* in this example), the size in bytes of each element to be read (1 in this example, because the elements are of type char), the number of elements to read (1 in this example, because the characters of the file are read one after the other), and, finally, a pointer to a FILE object that specifies the input stream (*sdCardFilePointer* in this example).

On line 22, the null character is written into *readBuffer* in order to indicate the end of the data, and on line 23 the file is closed.

If the file does not exist, then *sdCardFilePointer* is assigned the NULL value. In that case, the *else* condition of the *if* statement on line 12 is executed in order to indicate that the file was not found.

**NOTE:** In summary, in this example the reading of the file was ended because of one of these reasons:

i) It was not possible to open the file (i.e., *sdCardFilePointer == NULL*)
ii) The end of the file was reached (i.e., *feof(sdCardFilePointer) == true*)
iii) The maximum size of the buffer was reached (i.e., *i == readBufferSize - 1*)

For the sake of simplicity, the situation of an error on the file, which is assessed using *ferror()*, is not considered in Code 9.19.

```
bool sdCardReadFile( const char* fileName, char * readBuffer, int readBufferSize )
{
    char fileNameSD[SD_CARD_FILENAME_MAX_LENGTH+4] = "";
    int i;

    fileNameSD[0] = '\0';
    strcat( fileNameSD, "/sd/" );
    strcat( fileNameSD, fileName );

    FILE *sdCardFilePointer = fopen( fileNameSD, "r" );

    if ( sdCardFilePointer != NULL ) {
        pcSerialComStringWrite( "Opening file: " );
        pcSerialComStringWrite( fileNameSD );
        pcSerialComStringWrite( "\r\n" );

        i = 0;
        while ( ( !feof(sdCardFilePointer) ) && ( i < readBufferSize - 1 ) ) {
           fread( &readBuffer[i], 1, 1, sdCardFilePointer );
           i++;
        }
        readBuffer[i-1] = '\0';
        fclose( sdCardFilePointer );
        return true;
    } else {
        pcSerialComStringWrite( "File not found\r\n" );
        return false;
    }
}
```

***Code 9.20 Implementation of the function sdCardReadFile().***

### Proposed Exercises

1. What will happen if an event log file with no events is selected?

2. How can the code be modified in order to automatically generate a daily backup of the events?

### Answers to the Exercises

1. The smart home system will not store event log files with no events.

2. The *eventLogUpdate()* function can be modified in order to keep track of the current day, and once it detects that the day has changed, it generates a new file with the corresponding events of the day.

## 9.3 Under the Hood

### 9.3.1 Fundamentals of Software Repositories

All the programs used in this book are imported from [2] to the Keil Studio Cloud. This program sharing is done by means of a set of links and buttons, which are the *front end* (the presentation layer) of a *software repository* that is being used in the *back end* (the data access layer) to support the file sharing.

A software repository is a storage location for software. At the user side, a *package manager* (such as some tools provided by Keil Studio Cloud) usually helps to manage the repositories. The server side is typically provided by organizations, either free of charge or for a subscription fee. For example, major Linux distributions have many repositories around the world that mirror the main repository.

A repository provides a revision control system, which includes a historical record of changes in the repository together with the corresponding committed objects. The user of a repository usually has access to basic information about the repository, such as that shown in Table 9.13.

***Table 9.13 Summary of typical information available about a repository.***

| Information | Used to |
|---|---|
| Versions available | Indicate current and previous versions |
| Dependencies | Indicate other elements that the current element depends on |
| Dependants | Specify other elements that depend on the current element |
| License | Govern the use or redistribution of the element |
| Build date and time | Be able to trace each specific version |
| Metrics | Indicate some properties of the element |

Some functionality that is provided by most repositories is to add, edit, and delete content; show the elements using different sorting properties; search for different types of elements and data; provide access control; and import and export elements.

By means of this functionality, a repository can evolve as shown in Figure 9.11. It can be seen that there is a main project (indicated in green) with different *tags* (1, 3, and 7 in Figure 9.11) from which branches (indicated in yellow) can be created. The branches can be merged with the main project, as shown in red, in order to create a new version of the software. They could, alternatively, become a discontinued development branch, such as the one indicated in violet.

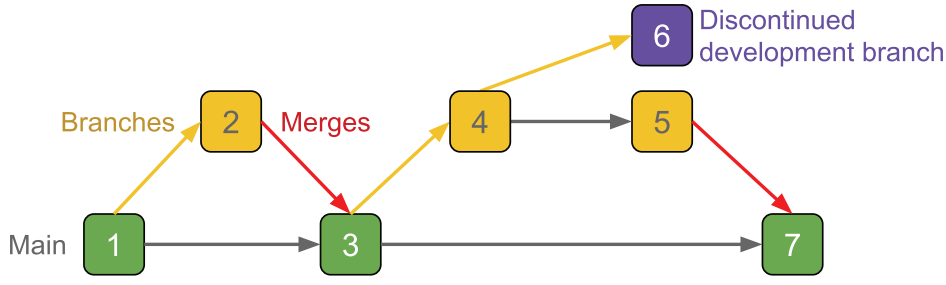



***Figure 9.11 Diagram of a typical evolution of a repository.***

The use of repositories provides many advantages, among which the following can be highlighted:

- Allows many programmers to work in the same repository without interrupting one another.
- Indicates when changes in the repository (*commits*) have been introduced by other programmers.

The Keil Studio Cloud provides an easy-to-use environment where most of these things can be done, as described in [6]:

- Set a remote repository
- Branch a repository
- Edit files from a repository
- Stage changes of a repository
- Commit to a repository
- Push to a repository

### Proposed Exercises

1. How can the revision history of a given project be accessed using Keil Studio Cloud?

2. How can the changes between selected revisions be seen using Keil Studio Cloud?

3. How can changes made on a program be committed?

4. How can a program be shared with other people?

**WARNING:** In order to do some of the proposed exercises, it is important to set credentials for GitHub in Keil Studio Cloud as described step-by-step in [6].

### Answers to the Exercises

1. The project should be selected as the active project and then the "History" view selected (its shortcut is Alt+H). The "Revision History" window will open, showing all the information corresponding to the commits that were made along the project history.

2. In the "Revision History" window, click once on any of the commits shown in order to select it. Click for a second time on the same commit in order to display the list of changed files. Double-click on one of the selected files. A window will open where the changes between revisions are shown in different colors.

3. The reader is encouraged to make a change in Example 9.4; for example, in *sdCardReadFile()*, change the line *pcSerialComStringWrite( "File not found\r\n" );* to *pcSerialComStringWrite( "File not found in the SD Card\r\n" )*. Then, select the "Source Control" window (Ctrl+Shift+G). Select the file *sd_card.cpp*

and press the "+" sign. The file should be listed in the "Staged changes" list. Enter a commit message and press the "Commit" button (its drawing is a check mark). Press the "More Actions..." button (represented by three small dots). A menu will be displayed. Select "Push". A message indicating that the reader does not have permission to push will be shown.

In order to commit the changes, the reader can fork to their own GitHub account but will first need to connect the Mbed account to their GitHub account, following the instructions shown by Keil Studio Cloud. If it was not previously connected, follow the step-by-step instructions indicated in [6]. Once the Mbed account is connected to the GitHub account of the reader, press "Push" again. A message indicating that the reader does not have permission to push will be shown. This time, the option to "Fork on GitHub" will be shown. After doing so, a message indicating that the fork has been created will be shown. In this way, the changes will be committed to the GitHub repository of the reader.

4. The reader should use a web browser to access GitHub and open the repository related to the project. Then, be sure that the repository access is configured as "public" and copy the URL of the repository in order to share the created repository with other people.

## 9.4 Case Study

### 9.4.1 Repository Usage in Mbed-Based Projects

In this chapter, the fundamentals of repositories and their usage was introduced. It is interesting to explore how repositories are used in Mbed-based projects. As an example, the game console provided with a graphical LCD display [7] that was introduced in the Case Study of Chapter 6 can be used.

A representation of the system is shown in Figure 9.12. On that web page, similar to the one in [8], a basic example of the game console "Pokitto" is shown, named "Hello World!" It can be seen that the current version is indicated by a label (where it says "Files at revision"). All the previous versions are available in the "History" tab.

At the bottom of Figure 9.12, it can be seen that there are three files in the repository: *My_settings.h*, *PokittoLib.lib*, and *main.cpp*. This file structure is similar to the one used in the first chapters of this book. By clicking on the "Revisions" link at the right of each file, the details of the corresponding revisions can be seen.

In the "Repository details" frame at the right of Figure 9.12, some interesting information can be seen. For example, it shows how many forks have been made of this repository, as well as how many commits were made to the repository. At the bottom of the frame, the whole repository can be downloaded.

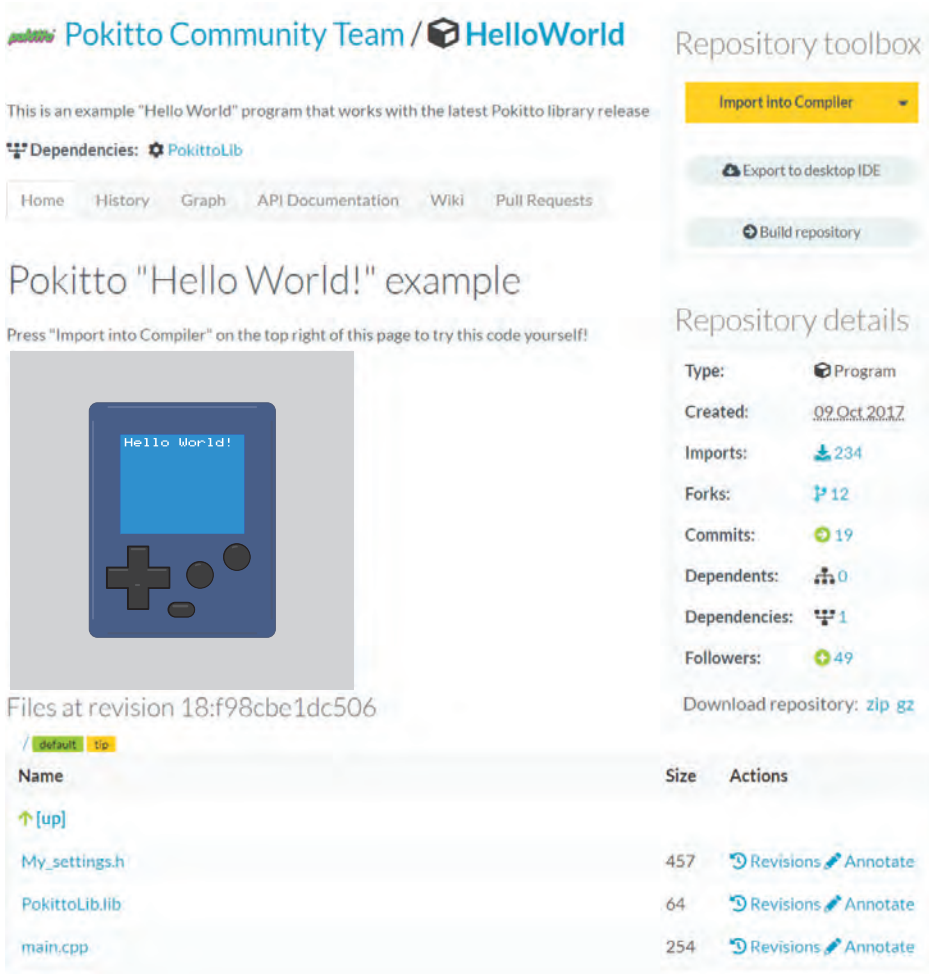



*Figure 9.12 Repository of an example of a game console based on Mbed.*

## Proposed Exercises

1. Are there any other examples of Pokitto programs available in repositories on the Mbed OS web page?

2. How could a copy of the Pokitto "Hello World!" example be copied into the reader's personal repository?

## Answers to the Exercises

1. About thirty examples are available in [9], all having the same organization as shown in Figure 9.12.

2. In order to make a copy of the example "Hello World!", the "Import into Compiler" button must be pressed.

## References

[1] "microSD Card Pinout, Features & Datasheet". Accessed July 9, 2021. https://components101.com/misc/microsd-card-pinout-datasheet

[2] "GitHub - armBookCodeExamples/Directory". Accessed July 9, 2021. https://github.com/armBookCodeExamples/Directory/

[3] "Data storage - API references and tutorials | Mbed OS 6 Documentation". Accessed July 9, 2021. https://os.mbed.com/docs/mbed-os/v6.12/apis/data-storage.html

[4] "pinout_labels - | Mbed". Accessed July 9, 2021. https://os.mbed.com/teams/ST/wiki/pinout_labels

[5] "mbed-os/PeripheralPinMaps.h at master · ARMmbed/mbed-os · GitHub". Accessed July 9, 2021. https://github.com/ARMmbed/mbed-os/blob/master/targets/TARGET_STM/TARGET_STM32F4/TARGET_STM32F429xI/TARGET_NUCLEO_F429ZI/PeripheralPinMaps.h

[6] "Arm Keil Studio Cloud User Guide". Accessed July 9, 2021. https://developer.arm.com/documentation/102497/1-5/Source-control/Work-with-Git

[7] "Game Console | Mbed". Accessed July 9, 2021. https://os.mbed.com/built-with-mbed/game-console/

[8] "HelloWorld - This is an example "Hello World" program that w... | Mbed". Accessed July 9, 2021. https://os.mbed.com/teams/Pokitto-Community-Team/code/HelloWorld/

[9] "Pokitto Community Team | Mbed". Accessed July 9, 2021. https://os.mbed.com/teams/Pokitto-Community-Team/

# Chapter 10

## Bluetooth Low Energy Communication with a Smartphone

# 10.1 Roadmap

## 10.1.1 What You Will Learn

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

- Describe how to connect Bluetooth Low Energy (BLE) modules to the NUCLEO board using a UART.
- Develop programs to exchange data between the NUCLEO board and a smartphone using a BLE connection.
- Summarize the fundamentals of a BLE connection.
- Describe the fundamentals of C++ objects.

## 10.1.2 Review of Previous Chapters

In previous chapters, the smart home system was provided with many functions, implemented by means of a set of sensors and actuators. This functionality was configured by the user using different interfaces, including a matrix keypad, an LCD display, and a PC. These interfaces were appropriate for this project, but often there are other interfaces that are more convenient, more flexible, or allow a better presentation of the information.

## 10.1.3 Contents of This Chapter

In this chapter, it will be explained how to enable communication between the NUCLEO board and a smartphone using a BLE connection. This will be achieved by using an HM-10 module connected to one of the UARTs of the NUCLEO board available in the ST ZIO connectors. In this way, relevant information from the smart home system will be shown on the smartphone. In addition, the gate will be controlled using the smartphone.

The fundamentals of object-oriented programming (OOP) will be introduced. It will be explained how OOP can be used to increase code modularity, reusability, flexibility, and effectiveness. Some details about the Mbed OS library objects that were used in previous chapters will be discussed.

Finally, a new way of implementing delays using interrupts and pointers will be shown. The delay in the main loop will be replaced by a *non-blocking delay*. This will be useful to avoid blocking the processor and keep it executing instructions while waiting for the expiration of the delay time.

# 10.2 Bluetooth Low Energy Communication between a Smartphone and the NUCLEO Board

## 10.2.1 Connect the Smart Home System to a Smartphone

In this chapter, the smart home system will be connected to a smartphone using Bluetooth Low Energy (BLE), as shown in Figure 10.1. The aim of this setup is to monitor relevant information from the smart home system and control the gate from the smartphone.

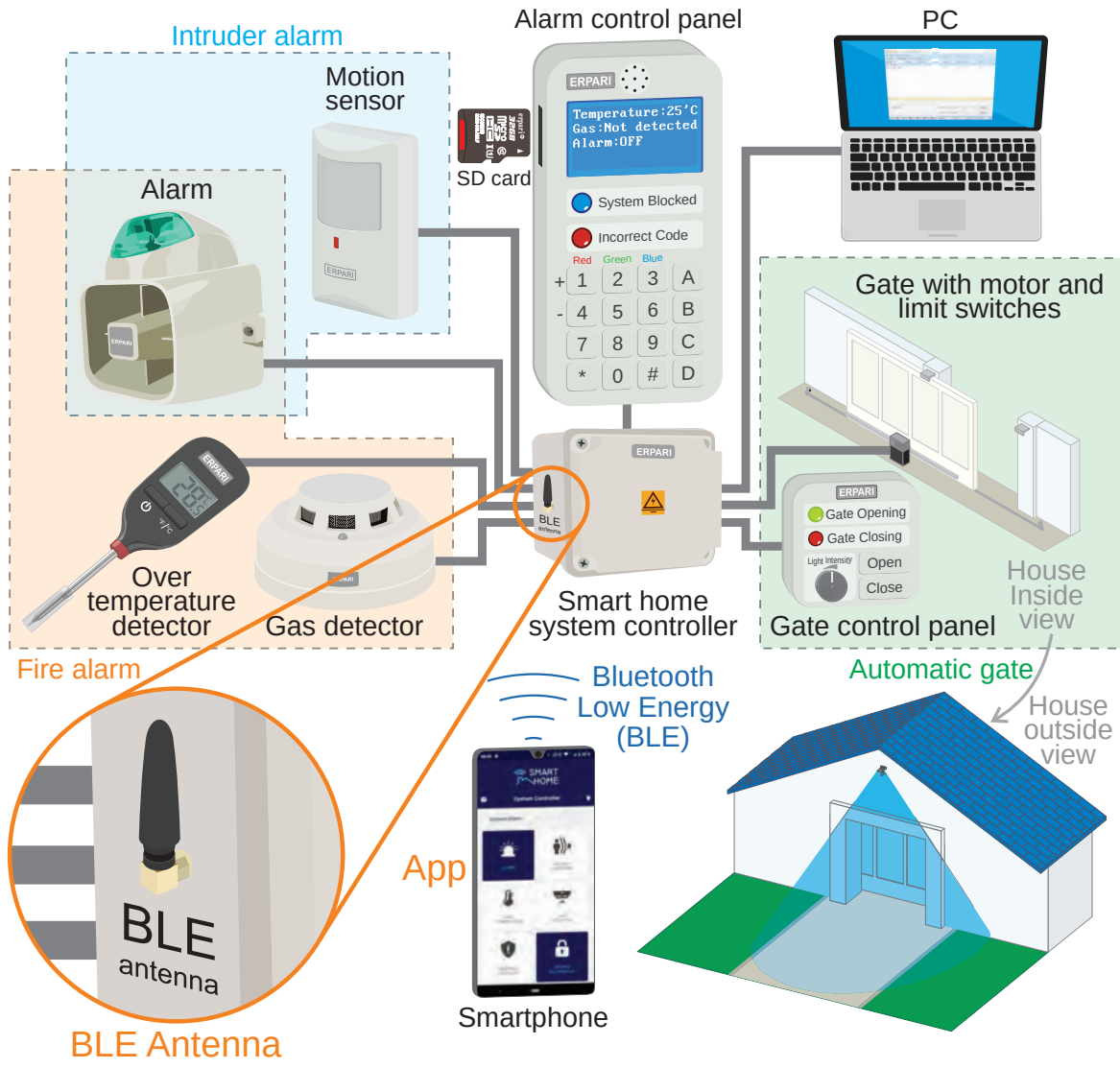



*Figure 10.1 The smart home system will be connected to a smartphone via Bluetooth Low Energy.*

In order to connect the NUCLEO board to a smartphone, two elements are required:

1. A module that provides the NUCLEO board with BLE communication capabilities.

2. A specifically prepared application running on a smartphone.

Figure 10.2 shows how to connect the HM-10 Bluetooth module, which is described in [1], to the NUCLEO board. These connections are summarized in Table 10.1.

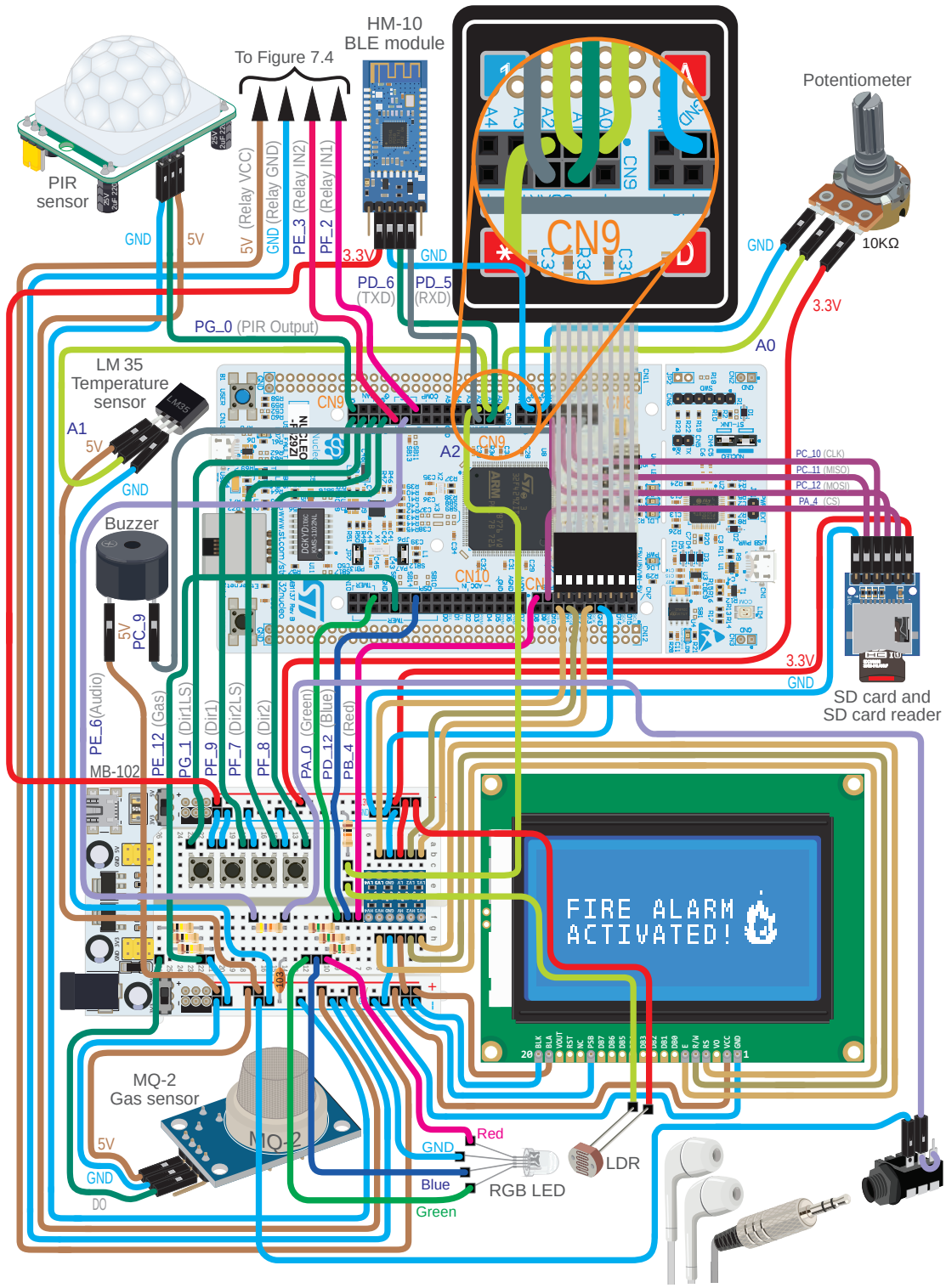



*Figure 10.2 Connections to be made between the NUCLEO board and the HM-10 module.*

*Table 10.1 Summary of the connections between the NUCLEO board and the HM-10 module.*

| NUCLEO board | HM-10 module |
|---|---|
| 3V3 | VCC |
| GND | GND |
| PD_6 (UART2_RX) | TXD |
| PD_5 (UART2_TX) | RXD |

In Figure 10.3, the pins of the HM-10 module are shown. The STATE pin is the *connection status*. It is in LOW when not connected and in HIGH when connected. The BRK pin is the Break pin. When there is an active connection, bringing the BRK pin LOW breaks the connection. These two pins are not used in this book.

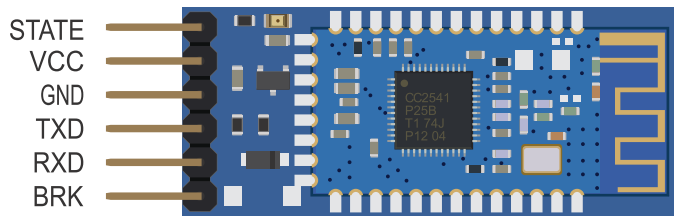



*Figure 10.3 Basic functionality of the HM-10 module pins.*

The HM-10 module is connected to the NUCLEO board by means of the pins PD_5 and PD_6, which are the TXD and RXD signals of a UART on the NUCLEO board; this is shown in Figure 8.5 and Table 10.1. The HM-10 module is responsible for the communication with the smartphone using a BLE connection. Using a UART serial communication, such as the one explained in Chapter 2, the information is exchanged between the NUCLEO board and the HM-10 module.

On the other side of the connection, a BLE module inside the smartphone implements the communication with the HM-10 module. The smartphone routes the messages from the smart home system to the application "Smart Home System App."

To test the HM-10 module, download the *.bin* file of the program "Subsection 10.2.1" from the URL available in [2] and load it onto the NUCLEO board. The application to be used on the smartphone is named "Smart Home System App" and should be downloaded from Google Play or the App Store (depending on the operating system of the smartphone the user owns) and installed on the smartphone as usual.

Open the application. The "Connect to the Smart Home System" screen is displayed. Click on the magnifying glass located at the bottom of the screen (Figure 10.4). Some BLE connections should be listed on the smartphone. Select the "BT05" connection (Figure 10.4). Once the connection is established, different icons will be displayed on the screen, and the status of the connection will be indicated in the top right corner (Figure 10.4). Wave a hand over the PIR sensor. The "ALARM" and "MOTION DETECTOR" indicators of the "Smart Home System App" should change their color. Press the "OPEN THE GATE" and "CLOSE THE GATE" buttons of the "Smart Home System App." The motor should rotate in one direction and then the other. If so, the reader is ready to move forward to the first example of this chapter.

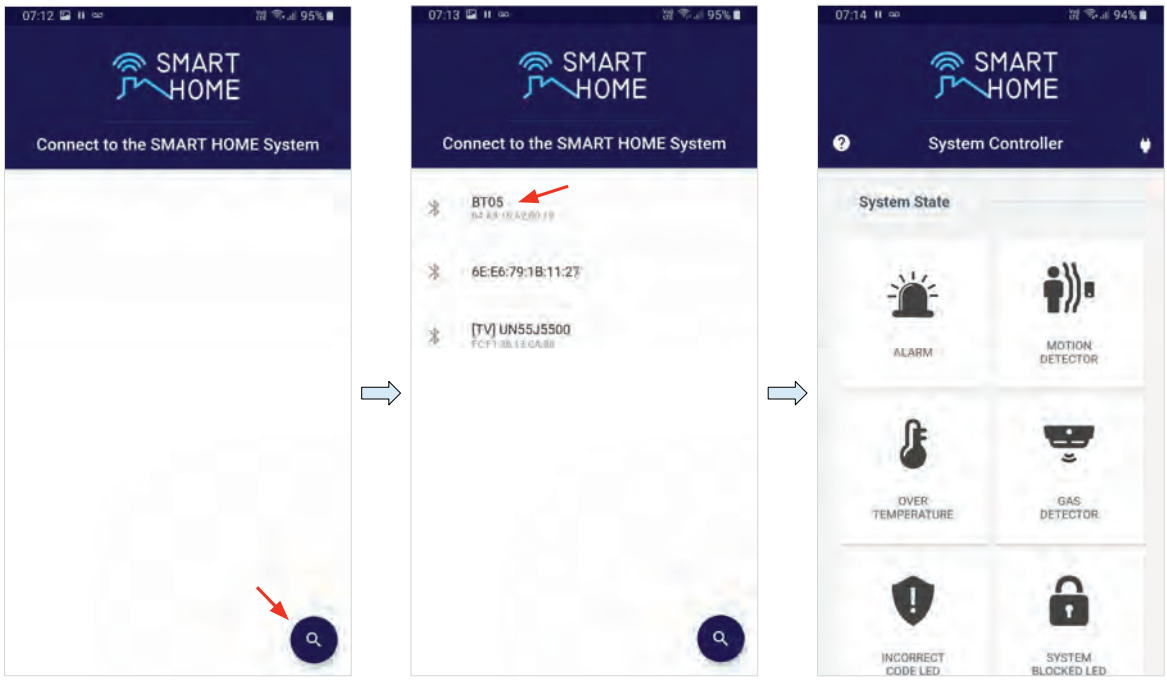



*Figure 10.4 Screenshots of the "Smart Home System App," showing the sequence to connect and use the application.*

**WARNING:** Ignore all the other elements of the setup during the proposed test (Alarm LED, display, etc.).

**TIP:** If the BLE connection is not established, check if all the wires are properly connected. If there are no mistakes in the connections detailed in Table 10.1, try using another HM-10 module or smartphone.

### 10.2.2 Messages Exchanged with the Smartphone Application

In order to exchange information between the program running on the NUCLEO board and the smartphone application, the list of messages that the smartphone application is prepared to receive and send by means of the BLE connection must be known.

**NOTE:** The code used to implement the application running on the smartphone is beyond the scope of this book and, therefore, is not analyzed. However, the reader may download from Google Play or the App Store a "Bluetooth terminal" application and use it on the smartphone to receive and send the messages detailed below in a similar way as messages are received and sent to the PC using the serial terminal. In that case, the "Smart Home System App" behavior can be emulated using a Bluetooth terminal. "CR+LF" should be configured in "newline receive settings" and "None" in "newline send settings."

All the messages that can be sent from the NUCLEO board to the application are listed in Table 10.2. These messages force a given element in the application to turn on or off.

***Table 10.2 Summary of the messages from the NUCLEO board to the application.***

| Message | Meaning |
|---|---|
| ALARM_ON | Turn on the state indicator of the alarm in the application |
| ALARM_OFF | Turn off the state indicator of the alarm in the application |
| GAS_DET_ON | Turn on the state indicator of the gas detector in the application |
| GAS_DET_OFF | Turn off the state indicator of the gas detector in the application |
| OVER_TEMP_ON | Turn on the state indicator of the over temperature detector in the application |
| OVER_TEMP_OFF | Turn off the state indicator of the over temperature detector in the application |
| LED_IC_ON | Turn on the state indicator of the Incorrect code LED in the application |
| LED_IC_OFF | Turn off the state indicator of the Incorrect code LED in the application |
| LED_SB_ON | Turn on the state indicator of the System blocked LED in the application |
| LED_SB_OFF | Turn off the state indicator of the System blocked LED in the application |
| MOTION_ON | Turn on the state indicator of the motion sensor in the application |
| MOTION_OFF | Turn off the state indicator of the motion sensor in the application |

All the messages that can be sent from the application to the NUCLEO board are listed in Table 10.3. The aim of these messages is to inform the board that a given button has been pressed or released in the application or to request the current state of all the system events that are shown by the smartphone.

***Table 10.3 Summary of the messages from the application to the NUCLEO board.***

| Message | Meaning |
|---|---|
| O | The "Open the gate" button was pressed in the application |
| C | The "Close the gate" button was pressed in the application |
| b | The application requested the current state of all the system events shown in Table 10.2 |

**NOTE:** In order to simplify the software implementation, only one letter is used in the messages from the application to the NUCLEO board, as can be seen in Table 10.3. Messages having more letters, as in Table 10.2, could be used. However, in that case the program becomes more complex.

In the following examples, the messages in Table 10.2 and Table 10.3 will be gradually introduced as more functionality regarding the smartphone application is incorporated into the smart home system.

**Example 10.1: Control the Gate Opening and Closing from a Smartphone**

### Objective

Introduce the use of a BLE module to receive data sent from a smartphone application to the NUCLEO board.

### Summary of the Expected Behavior

If the "Open the gate" button is pressed in the application, it is reported to the NUCLEO board, which in turn opens the gate if it is not already opened. If the "Close the gate" button is pressed in the application, it is reported to the NUCLEO board, which closes the gate if it is not already closed.

### Test the Proposed Solution on the Board

Import the project "Example 10.1" using the URL available in [2], build the project, and drag the *.bin* file onto the NUCLEO board. Open the application "Smart Home System App" on the smartphone. Connect the application to the NUCLEO board. Press the "Open the gate" button in the application. If the gate is not already opened, then it should be opened by the NUCLEO board. Press the "Close the gate" button in the application. The gate should close.

### Discussion of the Proposed Solution

The proposed solution is based on the BLE communication that is established between the HM-10 module and the application running on the smartphone. The fundamentals of BLE communication are described in the Under the Hood section at the end of this chapter. In this example, only the code that runs on the STM32 microcontroller is explained.

A new module named *ble_com* is included in order to receive commands from the smartphone application.

### Implementation of the Proposed Solution

The function *bleComUpdate()* is included in *smartHomeSystemUpdate()* (line 10 of Code 10.1) to periodically check if a new command is received from the smartphone. In order to implement this call, the library *ble_com.h* is included in *smart_home_system.cpp*, as can be seen in Table 10.4.

```
1  void smartHomeSystemUpdate()
2  {
3      userInterfaceUpdate();
4      fireAlarmUpdate();
5      intruderAlarmUpdate();
6      alarmUpdate();
7      eventLogUpdate();
8      pcSerialComUpdate();
9      lightSystemUpdate();
10     bleComUpdate();
11     delay(SYSTEM_TIME_INCREMENT_MS);
12 }
```

*Code 10.1 Details of the new implementation of smart_home_system.cpp.*

*Table 10.4 Sections in which lines were added to smart_home_system.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "ble_com.h"` |

The new module *ble_com* is shown in Code 10.2 and Code 10.3. The libraries that are included are shown from lines 3 to 6 of Code 10.2. On line 10, the public global object *uartBle* is declared. On line 14, the prototype of the private function *bleComCharRead()* is declared.

The implementation of the public function *bleComUpdate()* is shown on lines 18 to 27 of Code 10.2. The reader may notice that this is similar to the function *pcSerialComCommandUpdate()* of the *pc_serial_com* module. On line 20, the private function *bleComCharRead()* (implemented on lines 31 to 38) is used to read characters that the smartphone application sends. If the application does not send a character, then the function returns the null character ('\0').

On lines 21 to 26, if a new character is received from the smartphone application, a switch statement is used to call *gateOpen()* or *gateClose()* from the module *gate* if the character is "O" or "C", respectively.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4
5  #include "ble_com.h"
6  #include "gate.h"
7
8  //=====[Declaration and initialization of public global objects]===============
9
10 UnbufferedSerial uartBle(PD_5, PD_6, 9600);
11
12 //=====[Declarations (prototypes) of private functions]=======================
13
14 static char bleComCharRead();
15
16 //=====[Implementations of public functions]==================================
17
18 void bleComUpdate()
19 {
20     char receivedChar = bleComCharRead();
```

```
    if( receivedChar != '\0' ) {
        switch (receivedChar) {
            case 'O': gateOpen(); break;
            case 'C': gateClose(); break;
        }
    }
}

//=====[Implementations of private functions]==================================

static char bleComCharRead()
{
    char receivedChar = '\0';
    if( uartBle.readable() ) {
        uartBle.read(&receivedChar,1);
    }
    return receivedChar;
}
```

***Code 10.2 Details of the implementation of the ble_com.h.***

**NOTE:** The program code implemented in this example does not include the response of the "b" message that was introduced in Table 10.3. This will be addressed in Example 10.3.

Finally the implementation of the file *ble_com.h* is shown in Code 10.3.

```
//=====[#include guards - begin]===============================================

#ifndef _BLE_COM_H_
#define _BLE_COM_H_

//=====[Declarations (prototypes) of public functions]=========================

void bleComUpdate();

//=====[#include guards - end]=================================================

#endif // _BLE_COM_H_
```

***Code 10.3 Implementation of the file ble_com.h.***

## Proposed Exercise

1. How can the Fire alarm test button functionality be implemented by means of a button press on the smartphone application?

## Answer to the Exercise

1. A "Fire alarm test button" should be included in the application, and a character (such as "A") should be sent from the smartphone to the NUCLEO board when this button is pressed. The program running on the NUCLEO board should process this message and trigger the corresponding actions in order to set the variables *overTemperatureDetected* and *gasDetected* to ON.

### Example 10.2: Report the Smart Home System State to a Smartphone

#### Objective

Use of a BLE module to send data from the NUCLEO board to the smartphone.

#### Summary of the Expected Behavior

The states of different elements of the smart home system (alarm, gas detector, over temperature detector, Incorrect code indicator, System blocked indicator, and motion sensor) are communicated to the application running on the smartphone.

#### Test the Proposed Solution on the Board

Import the project "Example 10.2" using the URL available in [2], build the project, and drag the *.bin* file onto the NUCLEO board. Make sure that the alarm is not active. Open the smartphone application "Smart Home System App." Connect the application to the NUCLEO board. Press the Fire alarm test button (B1 User button). The state of the alarm should be displayed in the application. Wave a hand over the PIR sensor. The application should notify the user that motion has been detected.

#### Discussion of the Proposed Solution

The proposed solution is again based on the BLE communication established between the HM-10 module and the application running on the smartphone. New functionality is included in the *ble_com* module in order to implement the expected behavior.

#### Implementation of the Proposed Solution

The states of the alarm, gas detector, over temperature detector, Incorrect code indicator, System block indicator, and motion sensor are to be reported. To send the states to the smartphone application, the function *eventLogWrite()* from the module *event_log* is modified. The new implementation can be seen in Code 10.4.

```
1  void eventLogWrite( bool currentState, const char* elementName )
2  {
3      char eventAndStateStr[EVENT_LOG_NAME_MAX_LENGTH]) = "";
4
5      strcat( eventAndStateStr, elementName );
6      if ( currentState ) {
7          strcat( eventAndStateStr, "_ON" );
8      } else {
9          strcat( eventAndStateStr, "_OFF" );
10     }
11
12     arrayOfStoredEvents[eventsIndex].seconds = time(NULL);
13     strcpy( arrayOfStoredEvents[eventsIndex].typeOfEvent, eventAndStateStr );
14     if ( eventsIndex < EVENT_LOG_MAX_STORAGE - 1 ) {
15         eventsIndex++;
16     } else {
17         eventsIndex = 0;
```

```
18      }
19
20      arrayOfStoredEvents[eventsIndex].storedInSd = false;
21
22      pcSerialComStringWrite(eventAndStateStr);
23      pcSerialComStringWrite("\r\n");
24
25      bleComStringWrite(eventAndStateStr);
26      bleComStringWrite("\r\n");
27
28      eventAndStateStrSent = true;
29  }
```

***Code 10.4 New implementation of the function eventLogWrite().***

The function *eventLogWrite()* already sends the messages shown in Table 10.2 to the serial terminal (lines 22 and 23). Following a similar procedure, lines 25 and 26 are added to call the new function *bleComStringWrite()*. In order to implement this call, the library *ble_com.h* is included in *event_log.cpp*, as can be seen in Table 10.5. When the function *eventLogWrite()* is called, the variable *eventAndStateStrSent* is set to true (line 28). This variable is declared in *event_log.cpp*, as can be seen in Table 10.5, and its meaning will be explained in Code 10.5.

The new implementation of the function *eventLogUpdate()* can be seen in Code 10.5. This function is called approximately every 10 milliseconds (depending on the delays executed in the other function calls of *smartHomeSystemUpdate()*). The application running on the smartphone needs some time between received messages in order to process each command. For this reason, *eventLogUpdate()* is modified in order to send only one message in each call. To accomplish this, the variable *eventAndStateStrSent* is set to false in line 3. The event states will be updated only if this variable is false (lines 6 to 41).

As explained for Code 10.4, *eventAndStateStrSent* is set to true when a message is sent to the smartphone. This is because *eventLogWrite()* is called if there are changes in the state of any event. Then, if more than one event changes its state simultaneously, there will be at least a 10-millisecond delay between the corresponding messages. For instance, the ALARM_ON and GAS_DET_ON events occur almost simultaneously. Also, the LED_SB_ON and LED_IC_ON events occur simultaneously. In any of these cases, the corresponding messages will be sent to the smartphone with a gap of at least 10 milliseconds.

```
 1  void eventLogUpdate()
 2  {
 3      eventAndStateStrSent = false;
 4      bool currentState;
 5
 6      if ( !eventAndStateStrSent ) {
 7          currentState = sirenStateRead();
 8          eventLogElementStateUpdate( sirenLastState, currentState, "ALARM" );
 9          sirenLastState = currentState;
10      }
11
12      if ( !eventAndStateStrSent ) {
13          currentState = gasDetectorStateRead();
14          eventLogElementStateUpdate( gasLastState, currentState, "GAS_DET" );
```

```
15          gasLastState = currentState;
16      }
17
18      if ( !eventAndStateStrSent ) {
19          currentState = overTemperatureDetectorStateRead();
20          eventLogElementStateUpdate( tempLastState, currentState, "OVER_TEMP" );
21          tempLastState = currentState;
22      }
23
24      if ( !eventAndStateStrSent ) {
25          currentState = incorrectCodeStateRead();
26          eventLogElementStateUpdate( ICLastState, currentState, "LED_IC" );
27          ICLastState = currentState;
28      }
29
30      if ( !eventAndStateStrSent ) {
31          currentState = systemBlockedStateRead();
32          eventLogElementStateUpdate( SBLastState ,currentState, "LED_SB" );
33          SBLastState = currentState;
34      }
35
36      if ( !eventAndStateStrSent ) {
37          currentState = motionSensorRead();
38          eventLogElementStateUpdate( motionLastState ,currentState, "MOTION" );
39          motionLastState = currentState;
40      }
41  }
```

***Code 10.5 New implementation of the function eventLogUpdate().***

***Table 10.5 Sections in which lines were added to event_log.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "ble_com.h"` |
| Declaration and initialization of private global variables | `static bool eventAndStateStrSent;` |

The implementation of the new public function *bleComStringWrite()* from module ble_com is shown in Code 10.6, and its prototype is declared in *ble_com.h*, as shown in Table 10.6.

```
1  void bleComStringWrite( const char* str )
2  {
3      uartBle.write( str, strlen(str) );
4  }
```

***Code 10.6 New function bleComStringWrite() in ble_com.cpp.***

***Table 10.6 Sections in which lines were added to ble_com.h.***

| Section | Lines that were added |
|---|---|
| Declarations (prototypes) of public functions | `void bleComStringWrite( const char* str );` |

### Proposed Exercise

1. Given that the modules *pc_serial_com* and *ble_com* have many similarities, why was a function named *bleComInit()* not included in *smartHomeSystemInit()*?

### Answer to the Exercise

1. The HM-10 module automatically initializes itself after power on, so no initialization is required. However, as stated in other chapters, this function may have been included in order to follow the same pattern as in other modules even though no functionality is needed in the BLE communication.

## Example 10.3: Implement the Smart Home System State Report Using Objects

### Objective

Introduce the use of object-oriented programming.

**NOTE:** The C language does not support object-oriented programming. The C++ language is a superset of C that introduces the concept of classes and objects, among other features. Given that objects are used in all of the program code of this book (for example, DigitalIn, DigitalOut, etc.), the C language is not sufficient to interpret the code. The IDE (*Integrated Development Environment*, such as Keil Studio Cloud) infers that C++ is being used because the files have the extension *.cpp* ("C plus plus", C++).

### Summary of the Expected Behavior

The behavior of this example will remain exactly the same as that in Example 10.2, but some changes are included to improve the smartphone application functionality. Also, the code will be refactored in order to introduce the use of object-oriented programming.

### Test the Proposed Solution on the Board

Import the project "Example 10.3" using the URL available in [2], build the project, and drag the *.bin* file onto the NUCLEO board. Open the smartphone application "Smart Home System App." Connect the application to the NUCLEO board. Press the Fire alarm test button (B1 User button). The state of the alarm should be displayed in the application. Reset the NUCLEO board. Note that the alarm turns off on the NUCLEO board but in the smartphone application it remains on. Press the button "System Controller" in the smartphone application, which is used to send a request to the NUCLEO board for the updated state of the system events. The alarm should turn off.

### Discussion of the Proposed Solution

Throughout this book, the concept of *object* has been mentioned several times. In this example, a class named *system_event* is created. Several instances of this class are declared to implement part of the functionality of the *event_log* module. Each instance of the *system_event* class is called an object. How to implement methods and create attributes to use the *system_event* objects that are created is shown below.

## Implementation of the Proposed Solution

A new module called *system_event* is created. The implementation is shown in Code 10.7 and Code 10.8. In Code 10.7, the library *event_log.h* is included because the definition of EVENT_LOG_NAME_MAX_LENGTH is used on line 20. The declaration of the new class *systemEvent* can be seen on lines 12 to 21 of file *system_event.h*. The objects of this class will provide the functionality that is needed to implement the events that are logged and reported by the module *event_log*.

The class is divided into *public* (line 17) and *private* (line 22) *members*. The public members (from lines 14 to 17) are accessible outside the class, and the private members (from lines 19 to 21) are neither accessible nor viewable from outside the class. The concept is similar to the concepts of public and private that were introduced in Chapter 5. The variables and functions that belong to the class alone are called *attributes* and *methods*, respectively.

The public method declared on line 14 is a special method that every class must have, called the *constructor*. The constructor is a method with the same name as the class, which is automatically called to initialize an object of a class when it is created. The constructor may execute statements or call functions.

The implementation of the other methods and use of the attributes of this class is shown in Code 10.8 from lines 10 to 39. The difference in the implementation of functions is that the name of the class and "::" should precede the name of the method.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _SYSTEM_EVENT_H_
4  #define _SYSTEM_EVENT_H_
5
6  //=====[Libraries]=============================================================
7
8  #include "event_log.h"
9
10 //=====[Declaration of public classes]=========================================
11
12 class systemEvent {
13     public:
14         systemEvent(const char* eventLabel);
15         void stateUpdate( bool state );
16         bool lastStateRead( );
17         char* getLabel( );
18     private:
19         void lastStateUpdate(bool state);
20         char label[EVENT_LABEL_MAX_LENGTH];
21         bool lastState;
22 };
23
24 //=====[#include guards - end]=================================================
25
26 #endif // _SYSTEM_EVENT_H_
```

***Code 10.7 Details of the implementation of the file system_event.h.***

If the current state of the event is different from the previous state (line 32), then the function *eventLogWrite()* is called (line 33). Finally, the last state of the event is stored (line 35). The keyword *this* is an expression whose value is the memory address of the object on which the member function is being called.

The methods *lastStateRead()*, *getLabel()*, and *lastStateUpdate()* are shown in Code 10.8. The implementation of these methods is very straightforward: *lastStateRead()* returns the last state, while *getLabel()* returns a pointer to the label. The method *lastStateUpdate()* assigns the value of the parameter *state* to the *lastState* private variable of the object of the class *systemEvent*.

In Code 10.9, some sections of the file *event_log.cpp* are shown. The names of the struct and data types previously named *systemEvent* and *systemEvent_t* were renamed to *storedEvents* and *storedEvents_t*, respectively (lines 3 and 7). Line 21 was modified to account for this change. Also, the private function *eventLogElementStateUpdate()* was removed from this module.

The objects related to each of the events shown in Table 10.2 are declared on lines 11 to 16. When each of these objects is created, the constructor is called and the *eventLabel* attribute is assigned to the created object (line 12 of Code 10.8). Then, the private attribute *lastState* is initialized to OFF on line 13 of Code 10.8. In order to make use of the class *systemEvent*, the library *system_event.h* is included in *event_log.cpp*, as indicated in Table 10.7. Also in Table 10.7 the declarations of the function prototype *eventLabelReduce* and the private definition EVENT_LOG_NAME_SHORT_MAX_LENGTH are shown. In Table 10.8, the lines added to *event_log.h* are shown.

***Table 10.7 Sections in which lines were added to event_log.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "system_event.h"` |
| Declarations (prototypes) of private functions | `void eventLabelReduce(char * eventLogReportStr,`<br>`  systemEvent * event);` |
| Declaration of private defines | `#define EVENT_LOG_NAME_SHORT_MAX_LENGTH  22` |

***Table 10.8 Sections in which lines were added to event_log.h.***

| Section | Lines that were added |
|---|---|
| Declarations (prototypes) of public functions | `void eventLogReport();` |
| Declaration of public definitions | `#define EVENT_LABEL_MAX_LENGTH  10` |

```
//=====[Libraries]=============================================================

#include "mbed.h"
#include "arm_book_lib.h"

#include "system_event.h"

//=====[Implementations of public methods]===================================

systemEvent::systemEvent(const char* eventLabel)
{
    strcpy( label, eventLabel );
    lastState = OFF;
}

void systemEvent::stateUpdate( bool state )
{
    if ( state != this->lastStateRead() ) {
        eventLogWrite( state, this->getLabel() );
    }
    this->lastStateUpdate( state );
}

bool systemEvent::lastStateRead( )
{
    return lastState;
}

char* systemEvent::getLabel( )
{
    return label;
}

//=====[Implementations of private methods]==================================

void systemEvent::lastStateUpdate(bool state)
{
    lastState = state;
}
```

*Code 10.8 Details of the implementation of the file system_event.cpp.*

```
//=====[Declaration of private data types]====================================

typedef struct storedEvent {
    time_t seconds;
    char typeOfEvent[EVENT_LOG_NAME_MAX_LENGTH];
    bool storedInSd;
} storedEvent_t;

//=====[Declaration and initialization of public global objects]==============

systemEvent alarmEvent("ALARM");
systemEvent gasEvent("GAS_DET");
systemEvent overTempEvent("OVER_TEMP");
systemEvent ledICEvent("LED_IC");
systemEvent ledSBEvent("LED_SB");
systemEvent motionEvent("MOTION");

//=====[Declaration and initialization of private global variables]===========

static int eventsIndex     = 0;
static storedEvent_t arrayOfStoredEvents[EVENT_LOG_MAX_STORAGE];
static bool eventAndStateStrSent;
```

*Code 10.9 Details of the new implementation of some sections of the file event_log.cpp.*

The new implementation of the public function *eventLogUpdate()* is shown in Code 10.10. The method *stateUpdate()* is called for each system event if *eventAndStateStrSent* is false (lines 5 to 13) and replaces the functionality of the private function *eventLogElementStateUpdate()*, which was removed from *event_log.cpp*.

```
1  void eventLogUpdate()
2  {
3      eventAndStateStrSent = false;
4
5      if ( !eventAndStateStrSent ) alarmEvent.stateUpdate( sirenStateRead() );
6      if ( !eventAndStateStrSent ) gasEvent.stateUpdate(  gasDetectorStateRead() );
7      if ( !eventAndStateStrSent )
8          overTempEvent.stateUpdate(  overTemperatureDetectorStateRead() );
9      if ( !eventAndStateStrSent )
10         ledICEvent.stateUpdate(  incorrectCodeStateRead() );
11     if ( !eventAndStateStrSent )
12         ledSBEvent.stateUpdate( systemBlockedStateRead() );
13     if ( !eventAndStateStrSent ) motionEvent.stateUpdate( motionSensorRead() );
14 }
```

***Code 10.10 Details of the implementation of the ble_com.h.***

By comparing lines 11 to 16 of Code 10.9 and Code 10.10 with Code 7.20, it can be seen how the code modularity, reusability, flexibility, and effectiveness is increased by the use of object-oriented programming (OOP).

**NOTE:** Some other features of OOP, such as polymorphism, inheritance, encapsulation, and abstraction, are beyond the scope of this book and are, therefore, not discussed here.

**NOTE:** The objects defined by the Mbed OS that were introduced in previous chapters, such as DigitalIn, DigitalOut, UnbufferedSerial, etc. are used in a very similar way to the object introduced in this example. All those Mbed OS objects have a constructor and a set of publicly defined methods and can be instantiated as many times as needed, just like the *systemEvent* Object.

In order to improve the experience of using the smartphone application, the NUCLEO board now responds to the character "b" sent from the smartphone application. When this character is received, the function *eventLogReport()* is called, as shown in the new implementation of *bleComUpdate()* (line 8 of Code 10.11).

```
1  void bleComUpdate()
2  {
3      char receivedChar = bleComCharRead();
4      if( receivedChar != '\0' ) {
5          switch (receivedChar) {
6              case 'O': gateOpen(); break;
7              case 'C': gateClose(); break;
8              case 'b': eventLogReport(); break;
9          }
10     }
11 }
```

***Code 10.11 Details of the implementation of the bleComUpdate().***

The implementation of *eventLogReport()* in the module *event_log* is shown in Code 10.12. This function sends a string to the smartphone application that contains the state of the system events, separated by commas. Because there is a limitation in the length of the string, each event label and state is sent using only a character; for instance, the string "AF,GF,TF,IF,SF,MF" means that all the system events are off. From lines 5 to 20, this string (*eventLogReportStr*) is prepared using the function *eventLabelReduce()*, and on lines 22 to 23, *eventLogReportStr* is sent using *bleComStringWrite()*.

```
1  void eventLogReport()
2  {
3      char eventLogReportStr[EVENT_LOG_NAME_SHORT_MAX_LENGTH] = "";
4
5      eventLabelReduce( eventLogReportStr, &alarmEvent );
6      strcat( eventLogReportStr, "," );
7
8      eventLabelReduce( eventLogReportStr, &gasEvent );
9      strcat( eventLogReportStr, "," );
10
11     eventLabelReduce( eventLogReportStr, &overTempEvent );
12     strcat( eventLogReportStr, "," );
13
14     eventLabelReduce( eventLogReportStr, &ledICEvent );
15     strcat( eventLogReportStr, "," );
16
17     eventLabelReduce( eventLogReportStr, &ledSBEvent );
18     strcat( eventLogReportStr, "," );
19
20     eventLabelReduce( eventLogReportStr, &motionEvent );
21
22     bleComStringWrite(eventLogReportStr);
23     bleComStringWrite("\r\n");
24 }
```

***Code 10.12 Details of the implementation of the function eventLogReport().***

The implementation of the function *eventLabelReduce()* is shown in Code 10.13. This function concatenates onto the string *eventLogReportStr* a character that represents the event (lines 4, 6, 8, 10, 12, and 14) by using *strcmp* to compare the label of the event received with each event string (lines 3, 5, 7, 9, 11, and 13). Then, it checks the last state of the event (line 17) and appends a character that represents the state (lines 18 or 20).

```
static void eventLabelReduce(char * eventLogReportStr, systemEvent * event)
{
    if (strcmp(event->getLabel(), "ALARM") == 0) {
        strcat(eventLogReportStr,"A");
    } else if (strcmp(event->getLabel(), "GAS_DET") == 0) {
        strcat(eventLogReportStr,"G");
    } else if(strcmp(event->getLabel(), "OVER_TEMP") == 0) {
        strcat(eventLogReportStr,"T");
    } else if(strcmp(event->getLabel(), "LED_IC") == 0) {
        strcat(eventLogReportStr,"I");
    } else if(strcmp(event->getLabel(), "LED_SB") == 0) {
        strcat(eventLogReportStr,"S");
    } else if(strcmp(event->getLabel(), "MOTION") == 0) {
        strcat(eventLogReportStr,"M");
    }

    if ( event->lastStateRead() ) {
        strcat( eventLogReportStr, "N" );
    } else {
        strcat( eventLogReportStr, "F" );
    }
}
```

***Code 10.13 Details of the implementation of the function eventLabelReduce().***

### Proposed Exercise

1. How can a *systemEvent* object be created in order to monitor the state of the strobe light?

### Answer to the Exercise

1. A possible implementation may be by means of declaring the object in Code 10.9 as:

    systemEvent strobeLightEvent("STROBE_LIGHT");

and then incorporating the following lines in Code 10.10:

    strobeLightEvent.stateUpdate( strobeLightStateRead() );

## Example 10.4: Implement Non-Blocking Delays using Pointers and Interrupts

### Objective

Introduce the use of non-blocking delays and review the concepts of pointer and interrupt.

### Summary of the Expected Behavior

The behavior of this example will remain exactly the same as that in Example 10.3. However, the delay used in the main loop of the program will be replaced by a non-blocking delay.

### Test the Proposed Solution on the Board

Import the project "Example 10.4" using the URL available in [2], build the project, and drag the *.bin* file onto the NUCLEO board. The behavior should be exactly the same as in Example 10.3.

### Discussion of the Proposed Solution

The proposed solutions implemented throughout the examples in the book use blocking delays. This means that the processor waits until the expiration of the delay time without executing other instructions. Using non-blocking delays allows the program to check the condition of the delay time expiration with an *if* statement. In this way, the processor can execute other instructions while waiting for the delay time to expire.

### Implementation of the Proposed Solution

Code 10.14 shows the new implementations of *smartHomeSystemInit()* and *smartHomeSystemUpdate()*. The blocking delay of the function *smartHomeSystemUpdate()* is replaced by the function *nonBlockingDelayRead()*. This function is called using *smartHomeSystemDelay* as a parameter.

The "&" operator before the parameter is called the *reference operator* and it is defined as the "*memory address of…*". This operator brings an important concept: *parameter passing* to functions. In this book, the two most common methods are used: *pass-by-value* and *pass-by-reference*. In the former, a local copy of the variable used as a parameter is created and used inside the function, as introduced in Chapter 3. Therefore, the value of the variable used as a parameter is not modified, only the local copy. In the latter, the memory address of the variable used as a parameter is passed, and the function can change its value. In this way, if the value of the variable used as a parameter is modified inside the function, its value outside the function scope is also modified. Some examples are shown in Table 10.9.

***Table 10.9 Examples of functions with parameters passed by reference and by value.***

| Parameter passing method | Example |
|---|---|
| pass-by-value | `static void setPeriod( lightSystem_t light, float period );` |
| pass-by-reference | `bool sdCardWriteFile( const char* fileName, const char* writeBuffer )` |

The function *nonBlockingDelayRead()* that is used on line 19 of Code 10.14 checks if the time configured on line 14 of Code 10.14 (*SYSTEM_TIME_INCREMENT_MS*) to the variable *smartHomeSystemDelay* using *nonBlockingDelayInit()* is reached and returns true in that case or false otherwise. In this way, the processor is able to execute other instructions while waiting for the delay time to expire.

On line 3, the function *tickInit()* is called to configure the interrupt service routine (ISR) used to account time by the *non_blocking_delay* module. This function is based on a ticker, in a very similar way to the *brightControlInit()* function that was introduced in Example 8.1, as discussed below. The library *non_blocking_delay.h* is included to implement these function calls, and the variable *smartHomeSystemDelay* of type *nonBlockingDelay_t* is declared, as can be seen in Table 10.10.

**NOTE:** Recall the concept of interrupt service routines (ISRs), which was introduced in Chapter 7.

```
 1  void smartHomeSystemInit()
 2  {
 3      tickInit();
 4      audioInit();
 5      userInterfaceInit();
 6      alarmInit();
 7      fireAlarmInit();
 8      intruderAlarmInit();
 9      pcSerialComInit();
10      motorControlInit();
11      gateInit();
12      lightSystemInit();
13      sdCardInit();
14      nonBlockingDelayInit( &smartHomeSystemDelay, SYSTEM_TIME_INCREMENT_MS );
15  }
16
17  void smartHomeSystemUpdate()
18  {
19      if( nonBlockingDelayRead(&smartHomeSystemDelay) ) {
20          userInterfaceUpdate();
21          fireAlarmUpdate();
22          intruderAlarmUpdate();
23          alarmUpdate();
24          eventLogUpdate();
25          pcSerialComUpdate();
26          motorControlUpdate();
27          lightSystemUpdate();
28          bleComUpdate();
29      }
30  }
```

***Code 10.14 New implementation of the functions smartHomeSystemInit and smartHomeSystemUpdate.***

***Table 10.10 Sections in which lines were added to smart_home_system.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "non_blocking_delay.h"` |
| Declaration and initialization of public global objects | `static nonBlockingDelay_t smartHomeSystemDelay;` |

The implementation of *non_blocking_delay.cpp* is shown in Code 10.15 and Code 10.16. The library that is included is shown on line 3 of Code 10.15. On line 7, the global object that will be used for the ISR that is triggered by the ticker is created, and on line 8, the variable *tickCounter* is declared. Finally, the prototype of the function *tickerCallback()* is declared on line 12, and the prototype of *tickRead()* is declared on line 13.

```
1  //=====[Libraries]=============================================================
2
3  #include "non_blocking_delay.h"
4
5  //=====[Declaration and initialization of private global variables]============
6
7  static Ticker ticker;
8  static tick_t tickCounter;
9
10 //=====[Declarations (prototypes) of private functions]=======================
11
12 void tickerCallback();
13 tick_t tickRead();
```

***Code 10.15 Details of the implementation of the file non_blocking_delay.cpp (1/2).***

Code 10.16 shows the implementation of the public functions. On lines 3 to 6, the timer interrupt is configured in the function *tickInit()*. The callback function *tickerCallback* will be called once every millisecond. This function, implemented from lines 40 to 43, increments the variable *tickCounter* by one in each call.

The function used to initialize the non-blocking delays is implemented from lines 8 to 12. As was mentioned before, the first parameter of type *nonBlockingDelay_t* is *passed-by-reference*. For this reason, the operator "*" should be included before *delay* (line 8 of Code 10.16). The operator "*" is called the *dereference operator* and it is defined as "the content pointed to by... ".

Because *delay* is a pointer to the type *nonBlockingDelay_t* and *nonBlockingDelay_t* is a *struct*, its members are accessed using the "->" operator. When line 14 of Code 10.14 is executed, the value *SYSTEM_TIME_INCREMENT_MS* is assigned to the member *duration* (line 10 of Code 10.16), and false is assigned to the member *isRunning* of the content pointed by *delay* (line 11), that is, the variable *smartHomeSystemDelay* (Code 10.14). As has been explained, the function modifies the passed variable directly.

The parameters of the function *nonBlockingDelayRead()* are also *passed-by-reference*. On line 16, a local bool variable named *timeArrived* is declared and initialized with false. If the member *isRunning* of the content pointed by *delay* is true, then the delay time has not expired. This condition is checked by an *if-else* statement. If it is not running (line 19), then the non-blocking delay is started (lines 20 and 21). If it is running (line 22), the elapsed time is obtained as the difference between *tickCounter* and *delay->startTime* (line 23). Line 24 assesses if *elapsedTime* has reached the delay duration. If so, true is assigned to *timeArrived* on line 25, and the member *isRunning* of the content pointed by *delay* is set to false (line 26). Finally, the local variable *timeArrived* is returned by the function *nonBlockingDelayRead()* on line 28.

```
1  //=====[Implementations of public functions]=================================
2
3  void tickInit()
4  {
5      ticker.attach( tickerCallback, ((float) 0.001 ));
6  }
7
8  void nonBlockingDelayInit( nonBlockingDelay_t * delay, tick_t durationValue )
9  {
10    delay->duration = durationValue;
11    delay->isRunning = false;
12 }
13
14 bool nonBlockingDelayRead( nonBlockingDelay_t * delay )
15 {
16    bool timeArrived = false;
17    tick_t elapsedTime;
18
19    if( !delay->isRunning ) {
20       delay->startTime = tickCounter;
21       delay->isRunning = true;
22    } else {
23       elapsedTime = tickCounter - delay->startTime
24       if ( elapsedTime >= delay->duration ) {
25          timeArrived = true;
26          delay->isRunning = false;
27       }
28    }
29
30    return timeArrived;
31 }
32
33 void nonBlockingDelayWrite( nonBlockingDelay_t * delay, tick_t durationValue )
34 {
35    delay->duration = durationValue;
36 }
37
38 //=====[Implementations of private functions]================================
39
40 void tickerCallback( void )
41 {
42     tickCounter++;
43 }
```

***Code 10.16 Details of the implementation of the file non_blocking_delay.cpp (2/2).***

The function *nonBlockingDelayWrite()* implemented from lines 33 to 36 of Code 10.16 assigns the parameter *durationValue* (*passed-by-value* as the second parameter of the function) to the member *duration* of the content pointed by *delay*.

The implementation of *non_blocking_delay.h* is shown in Code 10.17. The library *mbed.h* (line 8) is included because the new data type tick_t is defined using an unsigned integer of 64 bits: *uint64_t* (line 14). This definition will allow the implementation of large delays without overflow.

**TIP:** In some applications, a ticker is used to implement a very precise measurement with a high resolution of time elapsed between events (for example, the time between the transmission of a signal and the reception of the response). In those cases, the ticker interval can be specified in microseconds, as can be seen in [3].

The other data type in this module is defined from lines 16 to 20: *nonBlockingDelay_t*. Finally, the prototypes of public functions are declared from lines 24 to 26.

```
1    //=====[#include guards - begin]===============================================
2
3    #ifndef _NON_BLOCKING_DELAY_H_
4    #define _NON_BLOCKING_DELAY_H_
5
6    //==================[Libraries]=================================================
7
8    #include "mbed.h"
9
10   //=====[Declaration of public data types]======================================
11
12   typedef uint64_t tick_t;
13
14   typedef struct{
15      tick_t startTime;
16      tick_t duration;
17      bool isRunning;
18   } nonBlockingDelay_t;
19
20   //=====[Declarations (prototypes) of public functions]=========================
21
22   void tickInit();
23
24   void nonBlockingDelayInit( nonBlockingDelay_t* delay, tick_t durationValue );
25   bool nonBlockingDelayRead( nonBlockingDelay_t* delay );
26   void nonBlockingDelayWrite( nonBlockingDelay_t* delay, tick_t durationValue );
27
28   //=====[#include guards - end]=================================================
29
30   #endif // _NON_BLOCKING_DELAY_H_
31
32
```

***Code 10.17 Details of the implementation of the file non_blocking_delay.h.***

**NOTE:** The proposed implementation of the non-blocking delay is based on a software module of the *sAPI library*. The *sAPI (simple Application Programming Interface) library* is an open-source library written by Eric Pernia and other collaborators for *Proyecto CIAA*. Many ideas in the sAPI library were used as a starting point for many of the code examples in this book. The reader is encouraged to explore the *sAPI library* in [4], where a broad set of useful functions is available, ranging from step motor drivers to LED dimming code, as well as information about Proyecto CIAA (Computadora Industrial Abierta Argentina, Argentine Open Industrial Computer), which is the context in which the sAPI library was written.

### Proposed Exercise

1. How can a blocking delay be implemented using the tick interrupt?

### Answer to the Exercise

1. A module similar to *non_blocking_delay* might be implemented with a function *blockingDelay()*, as shown in Code 10.18.

```
1  void blockingDelay( tick_t durationMs )
2  {
3     tick_t startTime = tickCounter;
4     tick_t elapsedTime;
5
6     while ( elapsedTime < durationMs ) {
7         elapsedTime = tickCounter - startTime;
8     }
9  }
```

***Code 10.18 Details of the implementation of the function blockingDelay().***

## 10.3 Under the Hood

### 10.3.1 Basic Principles of Bluetooth Low Energy Communication

In this chapter, the connection between the NUCLEO board and the smartphone was made using Bluetooth Low Energy (BLE) by means of an HM-10 module. Firstly, it should be noted that classic Bluetooth and BLE are two different technologies. There are many differences, but the most relevant is that BLE is designed for low energy consumption.

**NOTE:** In BLE, the low energy consumption is achieved by using smaller data packets that are transmitted only when necessary. BLE is not designed for continuous connections and large amounts of data. When large amounts of data need to be transmitted, it is more convenient to use classic Bluetooth, which maintains a continuous connection.

In this chapter, a connection of the type *Central + Peripheral* was used. In this configuration, a peripheral advertises itself at startup and waits for a central device to connect to it. A peripheral is usually a small device like a smart sensor. A central device is usually a smartphone that is scanning for devices. After a peripheral makes a connection, it is called the *subordinate*. After a central makes a connection, it is called the *manager*. This is illustrated in Figure 10.5.

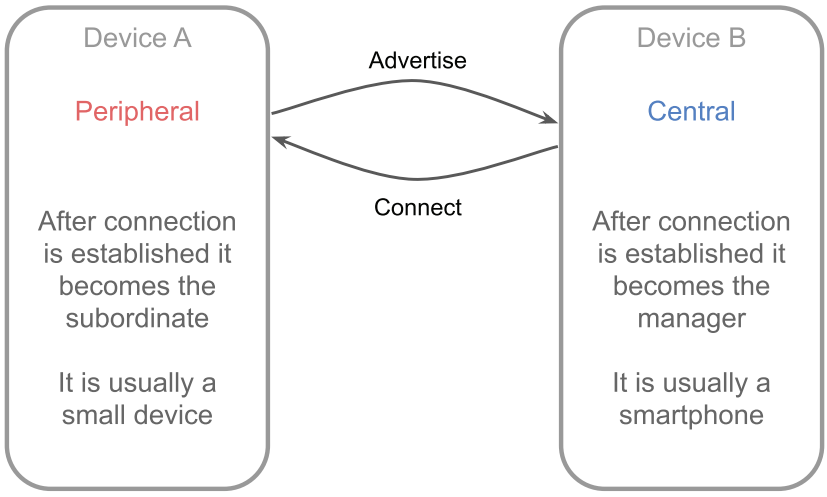



*Figure 10.5 Illustration of the names and behaviors of each device in the BLE startup process.*

After a BLE connection has been established, the more typical names and behaviors are as shown in Figure 10.6. The client is usually the manager, and the server is usually the subordinate.

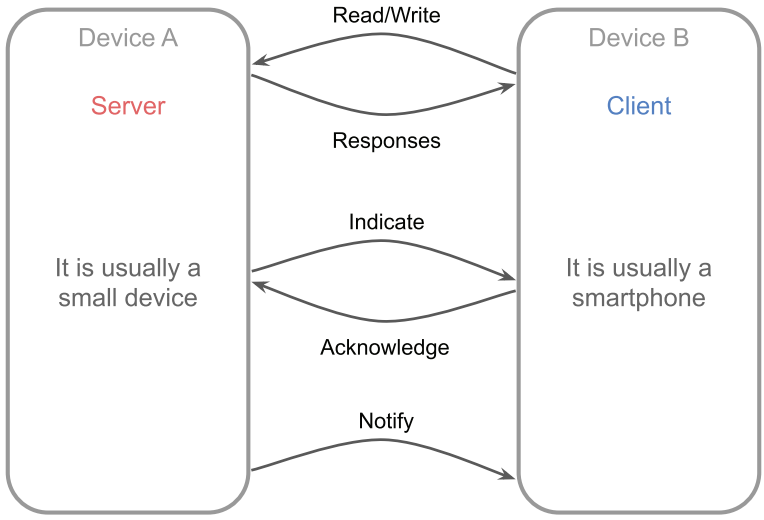



*Figure 10.6 Illustration of the names and behaviors of each device in a typical BLE communication.*

**NOTE:** The data exchange shown in Figure 10.5 and Figure 10.6 is automatically managed by the HM-10 module. Therefore, the C/C++ code to use the BLE communication does not have to tackle this task.

A server provides resources to the client. For example, in this chapter the NUCLEO board with the HM-10 module is the server. It provides the states of the alarm, the gas detector, the over temperature detector, the Incorrect code LED, the System blocked LED, and the motion detector to the client (the smartphone).

**NOTE:** It must be mentioned that a client is usually the manager, but a client could instead be the subordinate. Conversely, a server is usually the subordinate, but a server could be the manager. This role change is not necessary for basic setups and, therefore, is neither used nor explained further in this book.

As can be seen in Figure 10.6, there are three possible ways in which data can be exchanged between the client and the server:

- A client sends *Read/Write* operations to the server, and the server responds with data; if appropriate, the server changes its local data or configuration.
- A server sends an *Indicate* operation to the client, which is acknowledged by the client.
- A server sends a *Notify* operation to the client, which is not acknowledged by the client.

In this chapter, the connection implemented was based on Notify and Read/Write operations over a *personalized service* of the HM-10 module. The messages sent by the NUCLEO board to the smartphone (Table 10.2) correspond to Notify operations. The messages sent by the smartphone to the NUCLEO board (Table 10.3) are Read/Write operations, even if there is no response from the NUCLEO board.

BLE operates in the spectrum range of 2.400–2.4835 GHz, as does classic Bluetooth technology. It uses a different set of channels, however. Instead of classic Bluetooth's 79 1-MHz channels, BLE has 40 2-MHz channels. In order to avoid communication collisions between different clients and servers, each client–server pair should use a different channel. BLE also uses *frequency hopping* to counteract narrowband interference problems.

Within a channel, data is transmitted using Gaussian frequency shift modulation, similar to classic Bluetooth's Basic Rate scheme. The bit rate is between 1 Mbit/s and 2 Mbit/s, depending on the BLE version. Further details are given in Volume 6 Part A (Physical Layer Specification) of the Bluetooth Core Specification V4.0 [5].

**NOTE:** In advanced setups, a device can be the central device to up to eight other devices that are acting as peripherals. A device can also be a central and peripheral simultaneously to different devices.

### Proposed Exercises

1. It was discussed in the proposed exercises of subsection "2.3.1 Basic Principles of Serial Communication" how one device can be sure that another device has received information with no errors. It was concluded that both devices must be involved in the process of checking the integrity of the information. Is it possible to find this kind of process in Figure 10.6?

2. It was concluded in the proposed exercises subsection "2.3.1 Basic Principles of Serial Communication" that using 115,200 bps serial communication, such as that used in Chapter 2, it will take about 86 seconds to transfer a 1 MB file. How much time will it take to transfer a 1 MB file using the BLE connection under optimal conditions?

### Answers to the Exercises

1. In Figure 10.6 it can be seen that the *Indicate* operation has an acknowledgement. This acknowledgement contains information that is used to check the integrity of the information.

2. 1 MB is equal to 8 Mbits. Considering a 1 Mbit/s physical layer bit rate and assuming that this reflects the actual data rate (which is not true), it will take 8 seconds to transfer a 1 MB file. This is about ten times less than the 86 seconds that was obtained in the calculation of subsection 2.3.1 for the UART connection at 115,200 bps.

**WARNING:** It is important to note that optimal conditions (namely, no interference from other devices, maximum possible speed with every device, etc.) are not easy to achieve, so in practical situations the real bit rate is lower than the aforementioned 1 Mbit/s. In the implementation of this chapter, the transfer speed of 1 Mbit/s cannot be reached because the communication with the HM-10 is at 9600 bps.

## 10.4 Case Study

### 10.4.1 Wireless Bolt

In this chapter, the NUCLEO board communicated with a smartphone by means of a BLE connection implemented with an HM-10 module. By using an application, it was possible to access the state of the elements and control the gate through the smartphone. A brief of a commercial "wireless bolt" built with Mbed, containing some similar features, can be found in [5]. An image of this is shown in Figure 10.7.

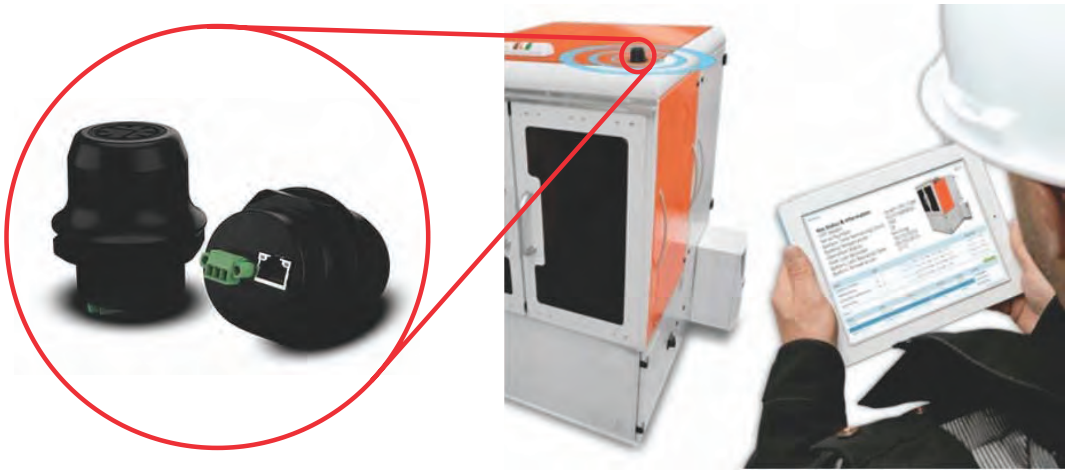


*Figure 10.7 "Anybus wireless bolt" built with Mbed contains elements introduced in this chapter.*

The wireless bolt shown in Figure 10.7 is designed to be mounted on an industrial device, machine, or cabinet and to enable wireless access via Bluetooth or wireless LAN. The system is made up of two elements: the black device shown inside the circle in Figure 10.7, and an application that runs on a tablet, laptop, or smartphone. By connecting the appropriate signals to the wireless bolt and properly configuring the application, it is possible to save the cost of buying an HMI (Human–Machine Interface). Another typical use is connecting the wireless bolt to an existing infrastructure or an external cloud service.

It can be appreciated that the functionality shown in Figure 10.7 is very similar to the functionality implemented in this chapter (to monitor and configure a device from a smartphone application using BLE). In the following chapter, it will be explained how to implement a Wi-Fi connection to the NUCLEO board, as well as looking at some other technologies that are mentioned in [6], such as TCP/UDP.

## Proposed Exercises

1. If the Anybus wireless bolt were to be connected to the smart home system, what elements might it be used to observe?

2. What are the main differences between the smart home system developed so far and the Anybus wireless bolt?

## Answers to the Exercises

1. It might be used to observe the state of the alarm, the gas detector, and the over temperature detector, for example.

2. The smart home system is intended to be used to monitor and control certain specific elements. Conversely, Anybus wireless bolt allows monitoring and control of various elements, according to

need in each case. Furthermore, the smart home system developed so far supports few connectivity options, just USB and Bluetooth, while the Anybus wireless bolt supports many connectivity options. In the next chapter, the connectivity of the smart home system will be increased.

## References

[1] "HM-10 Bluetooth Module Pinout, Features, Interfacing & Datasheet". Accessed July 9, 2021. https://components101.com/wireless/hm-10-bluetooth-module

[2] "GitHub - armBookCodeExamples/Directory". Accessed July 9, 2021. https://github.com/armBookCodeExamples/Directory

[3] "Ticker - API references and tutorials | Mbed OS 6 Documentation". Accessed July 9, 2021. https://os.mbed.com/docs/mbed-os/v6.12/apis/ticker.html

[4] "sAPI library for microcontrollers". Accessed July 9, 2021. https://github.com/epernia/firmware_v3/blob/master/libs/sapi/documentation/api_reference_en.md

[5] Bluetooth Core Specification V4.0 https://www.bluetooth.com/specifications/bluetooth-core-specification/

[6] "Anybus® Wireless Bolt™ | Mbed". Accessed July 9, 2021. https://os.mbed.com/built-with-mbed/anybus-wireless-bolt/

# Chapter 11

## Embedded Web Server over a Wi-Fi Connection

# 11.1 Roadmap

## 11.1.1 What You Will Learn

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

- Describe how to connect a Wi-Fi module to the NUCLEO board.
- Develop programs to serve a web page using the NUCLEO board and a Wi-Fi module.
- Summarize the fundamentals of AT commands and TCP/IP connections.

## 11.1.2 Review of Previous Chapters

In the previous chapter, the smart home system was provided with a BLE (Bluetooth Low Energy) connection by means of which it was possible to get some information on the state of the system using a smartphone, as well as to control the opening and closing of the gate. A limitation is that the BLE range is just a few meters, and sometimes it is useful to monitor the state of the system from a greater distance. This could be, for example, a web browser that is running on a PC or a smartphone. Also, in certain applications, a high data rate over a wireless connection is needed, as well as error-checked delivery of data, error-detection, and retransmission, among other capabilities that are limited if BLE is used.

## 11.1.3 Contents of This Chapter

In this chapter, the process of serving a web page using a Wi-Fi module is carried out step-by-step by the reader in subsection 11.2.2. The reader enters each command one after the other. The process is then gradually automated through the examples. In this way, the aim is to give an insight into the process in order to separate understanding of what should be done (the commands and the logic around those commands) from the automation of those commands and the corresponding logic on the NUCLEO board.

Some basic AT commands will be introduced (the *de facto* standard to communicate with different types of modems, Wi-Fi modules, GPS modules, cellular modules, etc.). Some basic concepts about TCP connections and Wi-Fi communications will also be discussed. In addition, the implementation of a *parser* will be shown, in order to analyze the responses of the Wi-Fi module to the AT commands sent by the NUCLEO board.

# 11.2 Serve a Web Page with the NUCLEO Board

## 11.2.1 Connect a Wi-Fi Module to the Smart Home System

In this chapter, the smart home system is provided with the capability of serving a web page using Wi-Fi, as shown in Figure 11.1. In this way, the information can be accessed using a smartphone or a PC.

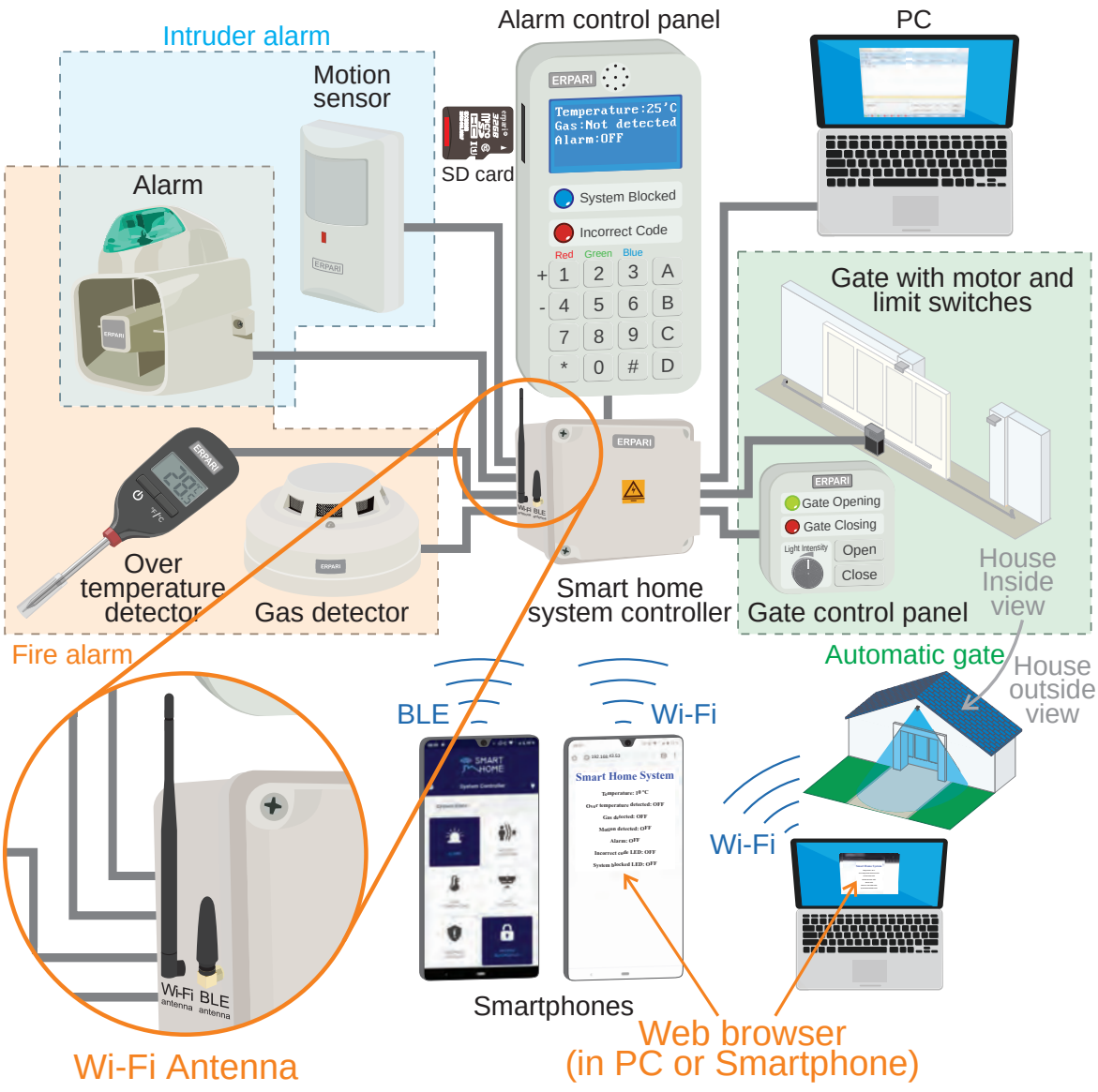



*Figure 11.1 The smart home system is now able to serve a web page.*

The Wi-Fi connection is implemented using an ESP-01 module, which is described in [1] and shown in Figure 11.2, and which is part of a broad family of Wi-Fi modules based on the ESP8266 chipset [2].

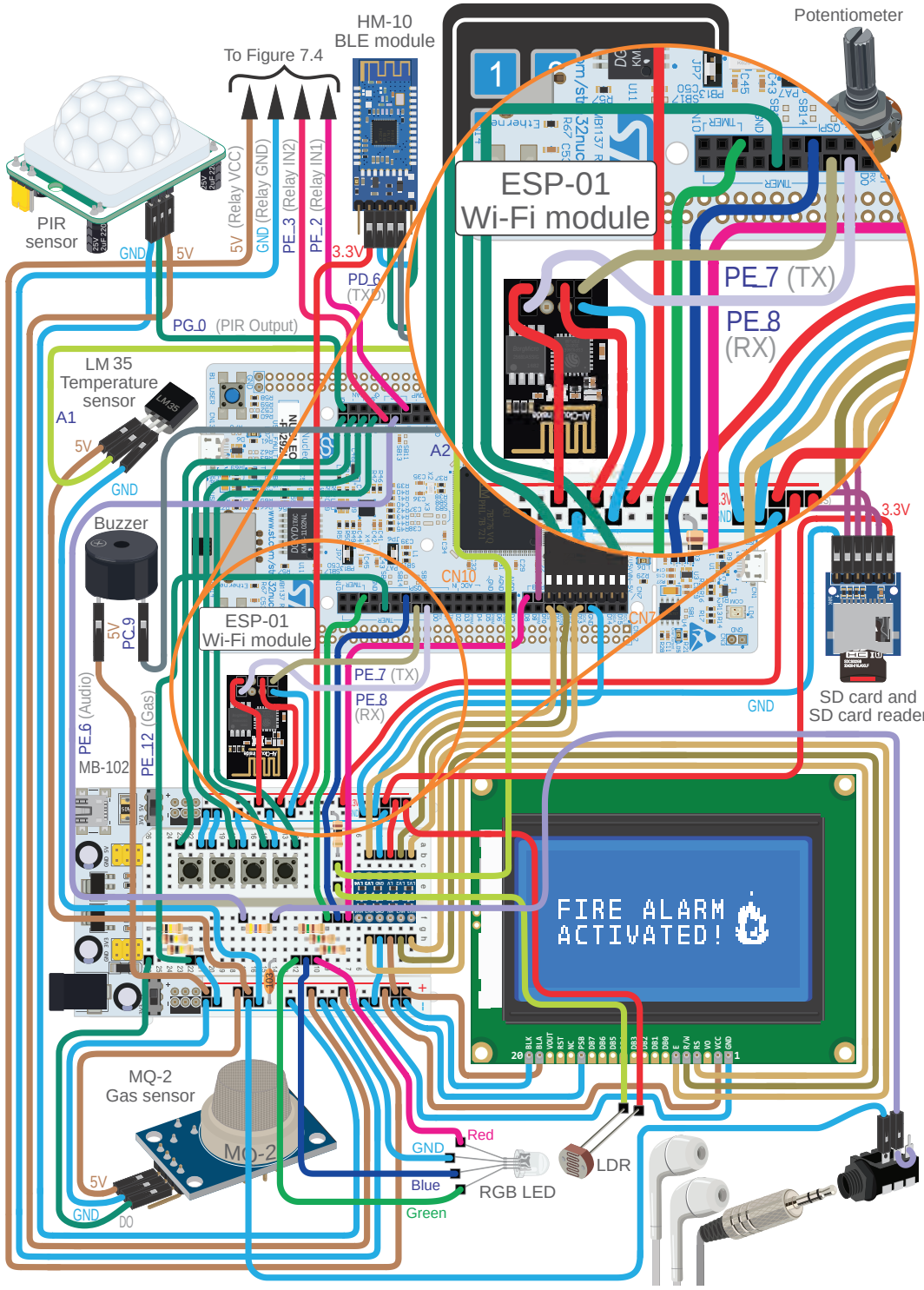



*Figure 11.2 The smart home system is now connected to a ESP-01 module.*

In Figure 11.3, the basic functions of the ESP-01 module pins are shown. Besides the GND and VCC power supply pins, the RST (reset) pin, and the EN (chip enabled) pin, it can be seen that there are two UART pins (RXD and TXD) and two GPIO pins (IO0 and IO2). The UART pins are used in this chapter to connect the ESP-01 module and the NUCLEO board, while the GPIO pins are not used in this book.

**NOTE:** More information about the broad set of functions of the ESP-01 module is available in [1].

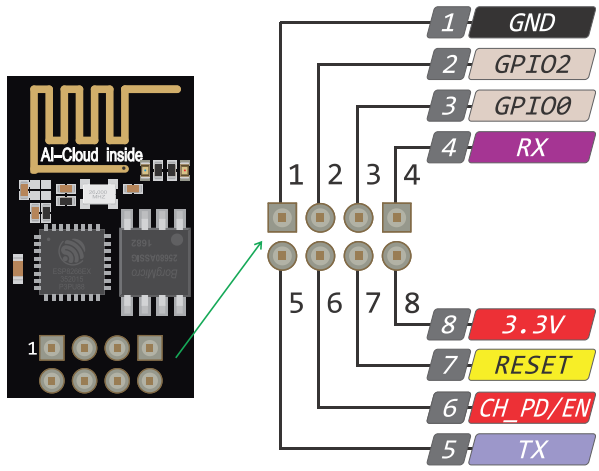



*Figure 11.3 Basic functionality of the ESP-01 module pins.*

The connections between the NUCLEO board and the ESP-01 module are summarized in Table 11.1, while the connections between the ESP-01 module and the breadboard are summarized in Table 11.2.

*Table 11.1 Summary of the connections between the NUCLEO board and the ESP-01 module.*

| NUCLEO board | ESP-01 module |
|---|---|
| PE_8 (UART7_TX) | RXD |
| PE_7 (UART7_RX) | TXD |

*Table 11.2 Summary of other connections that should be made to the ESP-01.*

| ESP-01 module | Breadboard |
|---|---|
| GND | GND |
| EN | 3.3 V |
| VCC | 3.3 V |

**WARNING:** The ESP-01 module has soldered pins on its bottom side. The connections in Figure 11.2 are for illustrative purposes only. The soldered pins on its bottom side must be used to connect the module.

To test if the ESP-01 module is working, the setup shown in Figure 11.4 will be used. In this setup, there is a smartphone or PC with a web browser that is connected to an access point by means of a Wi-Fi connection. The test program that runs on the NUCLEO board uses the connection to a PC, where the serial terminal is running. The program that runs on the NUCLEO board also uses a connection to the ESP-01 module, as shown in Figure 11.4. As discussed below, the ESP-01 module runs a TCP server (*Transmission Control Protocol* server), while the NUCLEO board runs an HTML server, which is implemented in this chapter.

During the test, the user is asked for the SSID (Service Set IDentifier) and password of the access point, and the test program configures the ESP-01 module using these credentials. This allows it to connect to the same access point as the smartphone or PC with the web browser. The ESP-01 module internally runs a TCP server, which receives requests from other devices connected to the access point – in this case the smartphone or PC. The ESP-01 module reports the requests to the test program that runs on the NUCLEO board, which provides the ESP-01 module with the HTML document to use in the response. The HTML document contains relevant information about the smart home system.

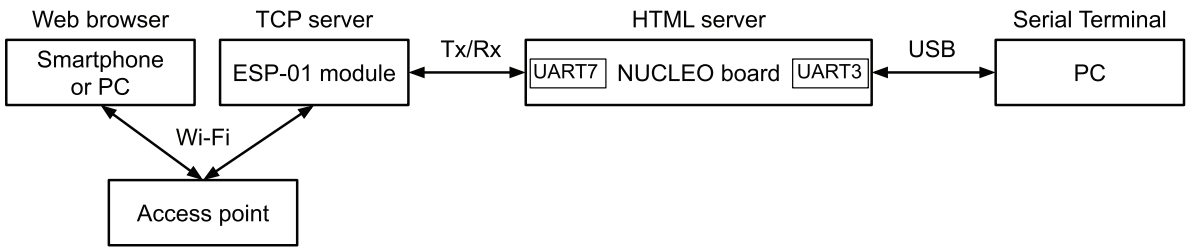



***Figure 11.4 Diagram of the communication that is implemented between the different devices.***

Download the *.bin* file of the program "Subsection 11.2.1" from the URL available in [3] and load it onto the NUCLEO board. This program uses the setup illustrated in Figure 11.4. After powering on, the NUCLEO board will ask the user to enter the SSID and the password of the Wi-Fi access point. It will then indicate the IP (Internet Protocol) address assigned to the ESP-01 module by the Wi-Fi access point, as shown in Figure 11.5.

```
*Subsection 11.2.1 test program*

Please provide the SSID of the Wi-Fi Access Point and press the Enter key
> mySSID
Wi-Fi Access Point SSID configured

Please provide the Password of the Wi-Fi Access Point and press the Enter
key
> **********
Wi-Fi Access Point password configured

Wi-Fi communication started, please wait...

IP address assigned correctly

Enter 192.168.43.53 as the URL in the web browser
```

***Figure 11.5 Steps to follow in the test program used in this subsection.***

Open the web browser on a PC or smartphone connected to the same access point as the ESP-01 module, and in the address bar enter the IP indicated on the serial terminal, as shown in Figure 11.6. A web page such as the one shown in Figure 11.6 should be displayed in the web browser. If so, the ESP-01 module is connected and working properly. Otherwise, review the connections of the module and the access point, and check that the ESP-01 is connected to the same access point as the PC or smartphone where the web browser runs.

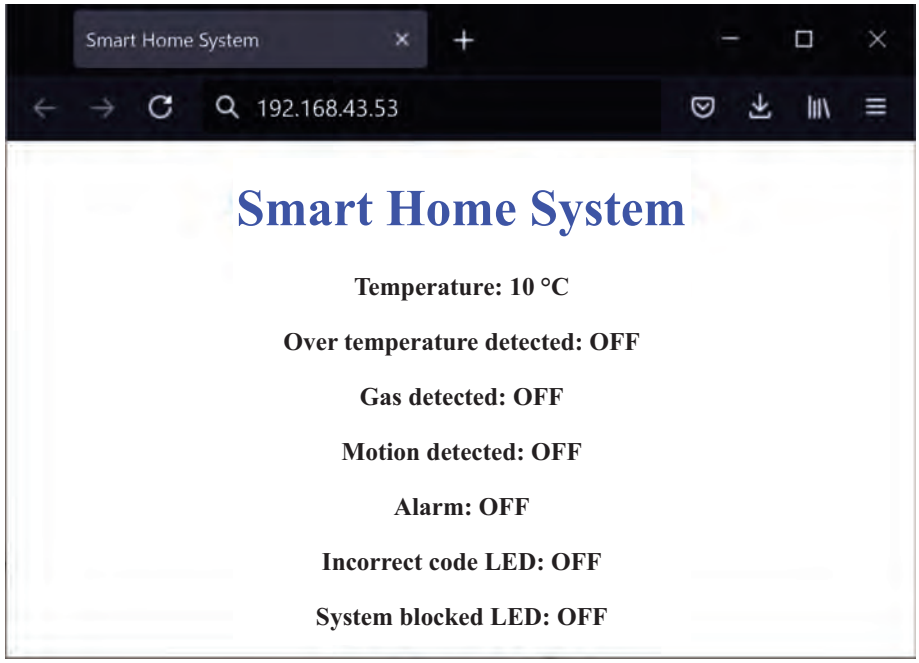



*Figure 11.6 Web page served by the ESP-01 module.*

### 11.2.2 Fundamentals of the Web Server to be Implemented

In this subsection, the fundamentals of web servers are introduced in order to allow the reader to understand the main concepts that are used in the examples that are discussed in this chapter.

The web page that was displayed in subsection 11.2.1 is provided by a *web server* that is composed of a TCP server that runs on the ESP-01 module and an HTML server that runs on the NUCLEO board. The primary function of a web server is to store, process, and deliver web pages to *clients*. Usually, a client is a web browser, which initiates the communication by making a request for a specific web page using the *Hypertext Transfer Protocol* (HTTP). The web server then responds with the web page content or with an error message if it is unable to retrieve the requested web page.

The web pages delivered are most frequently *Hypertext Markup Language (HTML) documents*, which may include images, style sheets, and scripts in addition to the textual content. The received HTML documents are rendered by the web browser. For this purpose, HTML documents can be assisted by technologies such as *Cascading Style Sheets (CSS)* used for describing the presentation of a web page (colors, fonts, etc.) and scripting languages such as *JavaScript* to enable interactive behavior.

It is also important to mention that the ESP8266 chipset, on which the ESP-01 module is based, can be configured as a server, as a client, and as server and client at the same time. When it is configured as a server, it serves HTML documents that in the proposed setup are retrieved by the NUCLEO board, as shown in subsection 11.2.1. When it is configured as a client, it retrieves HTML documents from a server.

The configuration of the ESP8266 chipset is done by means of *AT commands*. These commands were originally defined by Hayes Microcomputer Products in the early 1980s to be used to configure and operate modems. The AT commands consist of short texts that can be combined to produce commands for operations such as changing the parameters of a connection or connecting to a given IP address. In this subsection and in the examples below, some basic AT commands are used to configure the ESP8266 chipset. More information about the ESP8266 AT commands can be found in [4].

In order to get an idea about how all these concepts are used, a step-by-step example is shown, where AT commands are used to implement an embedded web server that serves a basic web page to a web browser.

First, download the *.bin* file of the program "Subsection 11.2.2" from the URL available in [3] and load it onto the NUCLEO board.

Next, the command "*AT*" (which stands for *attention*) should be typed into the serial terminal and the "Enter" key pressed on the PC keyboard. This AT command is forwarded by the NUCLEO board to the ESP-01 module, which should reply "*OK*". The NUCLEO board will forward this "OK" message to the serial terminal, as shown in Figure 11.7. This step is only to confirm that the most basic AT command works.

```
AT

OK
```

***Figure 11.7 The "AT" command (attention) is sent to the ESP-01 module, which replies "OK".***

The user should then type "*AT+CWMODE=1*" (standing for Change Wi-Fi mode) in order to configure the operation mode of the ESP-01 module as a station. This indicates that it should connect to an access point in order to get an IP address, after which it should reply "*OK*", as shown in Figure 11.8.

```
AT+CWMODE=1

OK
```

***Figure 11.8 The "AT+CWMODE=1" command (mode configuration) is sent to the ESP-01 module, which replies "OK".***

The next step is to connect the ESP-01 module to an available Wi-Fi access point, by means of entering the corresponding SSID and password. For example, if the SSID name is "mySSID" and the

password is "abcd1234", then the user should type: *AT+CWJAP="mySSID","abcd1234"* (note: for clarity, the quotes around commands issued to the ESP=01 module are omitted here and in the remainder of this section). This stands for *Join Access Point*, as shown in Figure 11.9. The ESP-01 module should reply "*WIFI CONNECTED*", "*WIFI GOT IP*", and "*OK*", and the NUCLEO board should forward these messages to the serial terminal, as shown in Figure 11.9.

```
AT+CWJAP="mySSID","abcd1234"
WIFI CONNECTED
WIFI GOT IP

OK
```

***Figure 11.9 The "AT+CWJAP" command (Join Access Point) is sent to the ESP-01 module.***

The user should then type *AT+CIFSR* in order to retrieve the IP address that has been assigned to the ESP-01 module. The response of the ESP-01 module will be forwarded by the NUCLEO board and will look as in Figure 11.10, where the IP address 192.168.43.53 has been assigned to the ESP-01 module.

```
AT+CIFSR
+ CIFSR:STAIP,"192.168.43.53"
+ CIFSR:STAMAC,"84:f3:eb:b7:34:84"

OK
```

***Figure 11.10 The "AT+CIFSR" command (Get IP Address) is sent to the ESP-01 module.***

**NOTE:** In Figure 11.10, STAIP stands for Station IP, and STAMAC stands for Station MAC. The MAC (*Media Access Control*) address is a unique identifier assigned to each network interface controller.

Next, the command *AT+CIPMUX=1* should be entered in order to enable multiple connections with the ESP-01 module using the assigned IP. The ESP-01 module should reply "OK", as shown in Figure 11.11.

```
AT+CIPMUX=1

OK
```

***Figure 11.11 The "AT+CIPMUX=1" command to enable multiple connections is sent to the ESP-01 module.***

The command *AT+CIPSERVER=1,80* should then be typed into the serial terminal to create a TCP server on the ESP-01 module. The "1" indicates that the command is to create a TCP server, and "80" is the port number assigned. This is the default HTTP port number. This TCP server is able to receive and respond to requests from different clients, for example web browsers. The ESP-01 module replies "OK", as shown in Figure 11.12.

```
AT+CIPSERVER=1,80

OK
```

*Figure 11.12 The "AT+CIPSERVER=1,80" command (creates a TCP server) is sent to the ESP-01 module.*

**NOTE:** The TCP server can only be created if multiple connections were first activated (AT+CIPMUX=1).

At this point, the TCP server is already running on the ESP-01 module. Therefore, whenever a client sends a request to the server IP address (in this case 192.168.43.53), the TCP server will keep a record that a TCP connection request has been received and will also keep some details of the request content.

To assess if a request has been received by the TCP server, the command *AT+CIPSTATUS* must be sent to the ESP-01 module. Figure 11.13 shows the response when no request has been received yet. All the possible values of STATUS are shown in Table 11.3. When the ESP-01 module has an assigned IP address and has received a TCP connection request, it will return "STATUS:3". In this way, it can be confirmed that a request has been received by the TCP server that is embedded in the ESP-01 module.

```
AT+CIPSTATUS

STATUS:2

OK
```

*Figure 11.13 The "AT+CIPSTATUS" command shows the connection status of the ESP-01 module.*

*Table 11.3 Summary of the AT+CIPSTATUS return values.*

| STATUS value | Meaning |
|---|---|
| 0 | The ESP-01 module is not initialized |
| 1 | The ESP-01 module is initialized, but Wi-Fi connection has not been started yet |
| 2 | The ESP-01 module is connected to an access point and has an assigned IP address |
| 3 | The ESP-01 module has an assigned IP address and has received a TCP connection request |
| 4 | All of the TCP/UDP/SSL connections of the ESP device station are disconnected |
| 5 | The ESP-01 module is not connected to an access point |

To show how the process works, the next step is to connect a PC or a smartphone to the same Wi-Fi access point as the ESP-01 module. The IP address assigned to the ESP-01 module (i.e., 192.168.43.53 in this case, as shown in Figure 11.10) should be entered into the address bar of a web browser on the PC or smartphone, as shown in Figure 11.14.

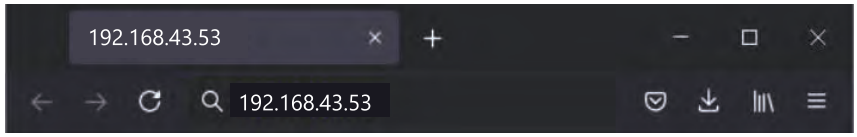



*Figure 11.14 A request to the ESP-01 module is sent by a web browser.*

The TCP server that is running on the ESP-01 module receives this request and informs the NUCLEO board that it has received a request by means of sending the messages that are shown in Figure 11.15.

```
0,CONNECT

+IPD,0,479:GET / HTTP/1.1
Host: 192.168.43.53
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/85.0.4183.102 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,
 image/webp,image/apng,/;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,es-AR;q=0.8,es;q=0.7
```

*Figure 11.15 The ESP-01 module indicates that a network connection with ID of 0 has been established.*

**NOTE:** The messages shown in Figure 11.15 are sent to the NUCLEO board by the ESP-01 module using a UART (as in Figure 11.4) and are then displayed on the serial terminal. This is only because the program that is running on the NUCLEO board forwards every character that it receives from the ESP-01 module to the PC by means of another UART (again, recall Figure 11.4).

In the first line of Figure 11.15, the ESP-01 module indicates that a TCP connection with ID 0 has been established. The other lines between *+IPD* and *q=0.7* are details about the connection that has been established and are not discussed here in order to keep this explanation as short as possible. For more information, please refer to [4].

At this point, if the command *AT+CIPSTATUS* is sent to the ESP-01 module, the response shown in Figure 11.16 is obtained. By means of "STATUS: 3", it is indicated that a TCP connection request has been received by the TCP server (recall Table 11.3). The details in Figure 11.16 follow the next sequence: +CIPSTATUS:<link ID>,<"type">,<"remote IP">,<remote port>,<local port>,<tetype>, as shown in [5].

```
AT+CIPSTATUS

STATUS:3
+CIPSTATUS:0,"TCP","192.168.77.198",60297,80,1.1

OK
```

***Figure 11.16 The "AT+CIPSTATUS" command shows the connection status of the ESP-01 module.***

The way in which the NUCLEO board indicates to the ESP-01 module how to respond to the web browser request is by loading the answer into the TCP server. In this case, the response will be an HTML document that must be loaded into the TCP server. In this step-by-step example, this is done by means of *AT+CIPSEND=0,52*, where "0" is the ID (identifier) of the connection with the TCP server and "52" is the length of the message in bytes that will be sent to the TCP server. Then, the ESP-01 module replies "OK" and sends the prompt symbol, "*>*", to the NUCLEO board in order to indicate that it is waiting for the HTML document, as shown in Figure 11.17.

```
AT+CIPSEND=0,52

OK
>
```

***Figure 11.17 The "AT+CIPSEND=0,52" command (sends data) is sent to the ESP-01 module, and it responds ">".***

The next step is to load the HTML document into the TCP server. In this particular example, the program that is loaded on the NUCLEO board will sequentially send the 52 bytes of the HTML document to the TCP server. The characters these bytes represent are shown in Code 11.1 and are sent when the "h" key is pressed on the PC keyboard.

```
<!doctype html> <html> <body> Hello! </body> </html>
```

***Code 11.1 The code of the HTML document that is loaded in the TCP server when the "h" key is pressed.***

The first part of the code, <*!doctype html*>, is used to indicate that it is an HTML document. Then, the tag <*html*> is used to indicate the beginning of the HTML code, and the <*body*> tag is used to indicate the beginning of the body of the HTML code. This is the part of the HTML document that the web browser renders on the screen. In this case, the body is just "Hello!" The last tags, (<*/body*> and <*/html*>), are used to close the previous tags.

Once the "h" key has been pressed on the PC keyboard, the 52 bytes are sent from the NUCLEO board to the ESP-01 module. The response of the ESP-01 module will be forwarded to the serial terminal using UART3 and, if everything goes well, the response will be as shown in Figure 11.18. This indicates that the 52 bytes were received correctly by the TCP server that is running on the ESP-01 module.

```
Recv 52 bytes

SEND OK

OK
```

*Figure 11.18 The "AT+CIPSEND=0,52" command (sends data) is sent to the ESP-01 module.*

Finally, the TCP connection between the ESP-01 module and the NUCLEO board should be closed by means of typing *AT+CIPCLOSE=0*, as shown in Figure 11.19. This is done in order to close the TCP connection with the web browser. In this way, the TCP server knows that no more data will be sent as a response to the web browser request.

```
AT+CIPCLOSE=0

0,CLOSED

OK
```

*Figure 11.19 The "AT+CIPCLOSE=0" command (close a TCP connection) is sent to the ESP-01 module.*

Now the TCP server has the response that should be sent to the web browser (i.e., the HTML document that has just been loaded to it) and sends the 52 bytes to the web browser. As a consequence, the web browser receives the 52 bytes, identifies it as an HTML document (because of the tag <!doctype html>), and displays the text "Hello!" as shown in Figure 11.20.

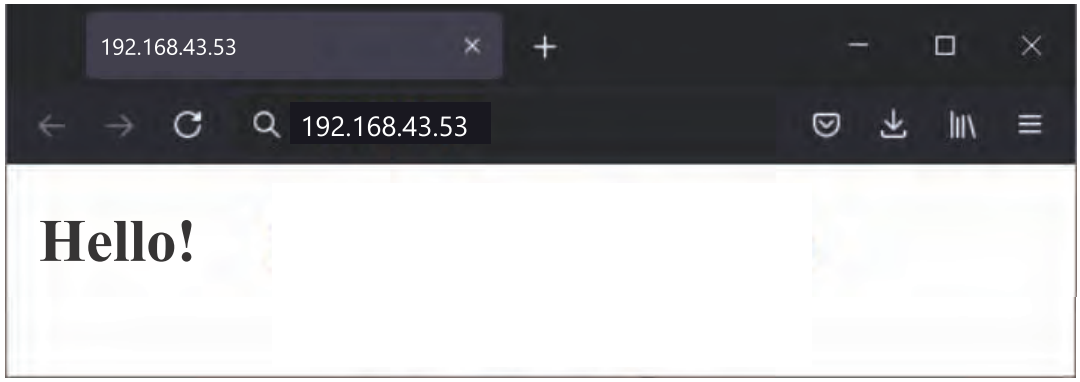



*Figure 11.20 Web page served by the ESP-01 module.*

**NOTE:** By means of using more complex HTML code, together with other resources such as JavaScript and CSS, more appealing and meaningful web pages can be created. These topics are beyond the scope of this book.

**WARNING:** Do not worry if a "Not secure" message is indicated by the web browser. The web page has no harmful elements. This message can be avoided if a TLS (Transport Layer Security) connection is provided, which is beyond the scope of this book.

In the following examples, all the steps that were followed in this section are automated by means of a new software module that is gradually incorporated into the smart home system program.

**Example 11.1: Implement the AT Command to Detect the Wi-Fi Module**

### Objective

Introduce the Finite-State Machine (FSM) that is used to control the ESP-01 module using AT commands.

### Summary of the Expected Behavior

The NUCLEO board will send the command "AT" to the ESP-01 module and will wait five seconds to receive the reply "OK". If this reply is received correctly by the NUCLEO board, then "AT command responded correctly" will be shown on the serial terminal. If this message is not received within 10 seconds, then "AT command not responded correctly" will be shown on the serial terminal.

### Test the Proposed Solution on the Board

Import the project "Example 11.1" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. If the ESP-01 module is working correctly, the message "AT command responded correctly" should appear on the serial terminal after 10 seconds. If this does not happen, check the connections, the ESP-01 module, and the access point and try again by pressing "a" on the PC keyboard.

### Discussion of the Proposed Solution

The proposed solution is based on a new module named *wifi_com*. This module will manage all the communications with the ESP-01 module.

### Implementation of the Proposed Solution

Code 11.2 shows the new implementation of *smartHomeSystemInit()* and *smartHomeSystemUpdate()*. On line 14, the *wifi_com* module is initialized by means of *wifiComInit()*. On line 15, the non-blocking delay is initialized using SYSTEM_TIME_INCREMENT_MS, which is equal to 10 milliseconds, as in Example 10.4.

```
 1  void smartHomeSystemInit()
 2  {
 3      tickInit();
 4      audioInit();
 5      userInterfaceInit();
 6      alarmInit();
 7      fireAlarmInit();
 8      intruderAlarmInit();
 9      pcSerialComInit();
10      motorControlInit();
11      gateInit();
12      lightSystemInit();
13      sdCardInit();
14      wifiComInit();
15      nonBlockingDelayInit( &smartHomeSystemDelay, SYSTEM_TIME_INCREMENT_MS );
16  }
17
18  void smartHomeSystemUpdate()
19  {
20      if( nonBlockingDelayRead(&smartHomeSystemDelay) ) {
21          userInterfaceUpdate();
22          fireAlarmUpdate();
23          intruderAlarmUpdate();
24          alarmUpdate();
25          eventLogUpdate();
26          pcSerialComUpdate();
27          motorControlUpdate();
28          lightSystemUpdate();
29          bleComUpdate();
30      }
31      wifiComUpdate();
32  }
```

***Code 11.2 New implementation of the functions smartHomeSystemInit() and smartHomeSystemUpdate().***

Given that the communication with the ESP-01 module will be established using a UART at a relatively high speed (115,200 bps) and there will be a relatively large number of bytes, the ESP-01 module must be read as fast as possible. For this purpose, the function *wifiComUpdate()* is called on line 31 outside the *if* statement that checks the non-blocking delay. In this way, *smartHomeSystemUpdate()* will call the functions from line 21 to line 29 every 10 milliseconds, while *wifiComUpdate()* will be called at a much higher rate. The reader is encouraged to compare how *smartHomeSystemUpdate()* is implemented in Code 11.2 and in Example 10.4.

In Table 11.4, the new library *wifi_com.h* that was added to *smart_home_system.cpp* is shown. The implementation of *wifi_com.h* is shown in Code 11.3. It can be seen that the public functions *wifiComRestart()*, *wifiComInit()*, and *wifiComUpdate()* are declared from line 8 to line 10.

***Table 11.4 Sections in which lines were added to smart_home_system.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "wifi_com.h"` |

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _WIFI_COM_
4  #define _WIFI_COM_
5
6  //=====[Declarations (prototypes) of public functions]=======================
7
8  void wifiComRestart();
9  void wifiComInit();
10 void wifiComUpdate();
11
12 //=====[#include guards - end]=================================================
13
14 #endif /* _WIFI_COM_ */
```

*Code 11.3 Implementation of wifi_com.h.*

The library *wifi_com.h* was added to *pc_serial_com.cpp*, as shown in Table 11.5. The table also shows that the private function *commandRestartWifiCom()* was declared in *pc_serial_com.cpp*. In order to implement this function, the lines shown in Table 11.6 were added in *pcSerialComCommandUpdate()* and *availableCommands()*. In this way, the command "a" is incorporated into the *pc_serial_com* module. This command calls the function *commandRestartWifiCom()*.

The implementation of *commandRestartWifiCom()* is shown in Code 11.4. This function sends "Wi-Fi communication restarted" to *uartUsb* (line 3) and calls the function *wifiComRestart()* of the *wifi_com* module (line 4).

*Table 11.5 Sections in which lines were added to pc_serial_com.cpp.*

| Section | Lines that were added |
|---|---|
| Libraries | `#include "wifi_com.h"` |
| Declarations (prototypes) of private functions | `static void commandRestartWifiCom();` |

*Table 11.6 Functions in which lines were added in pc_serial_com.cpp.*

| Function | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'a': case 'A': commandRestartWifiCom(); break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'a' or 'A' to restart the  Wi-Fi communication\r\n" );` |

```
1    static void commandRestartWifiCom()
2    {
3        pcSerialComStringWrite( "Wi-Fi communication restarted \r\n" );
4        wifiComRestart();
5    }
```

*Code 11.4 Implementation of commandRestartWifiCom().*

From Code 11.5 to Code 11.7, the implementation of *wifi_com.cpp* is shown. On lines 3 to 8 of Code 11.5, the libraries used by the *wifi_com* module are included. On line 12, DELAY_5_SECONDS is defined as 5000. A data type named *wifiComState_t* is declared in lines 16 to 22. This data type is used to implement an FSM that is used to control the ESP-01 module. A serial object is declared on line 27 in order to implement the communication with the ESP-01 module.

Line 30 declares a private array of char named *responseOk*. Because it is declared using the reserved word *const*, its content cannot be modified later in the program. Line 32 declares a pointer to a char type named *wifiComExpectedResponse*, which will be used to point to a string holding the expected response.

```
1  //=====[Libraries]=============================================================
2
3  #include "arm_book_lib.h"
4
5  #include "wifi_com.h"
6
7  #include "non_blocking_delay.h"
8  #include "pc_serial_com.h"
9
10 //=====[Declaration of private defines]========================================
11
12 #define DELAY_5_SECONDS         5000
13
14 //=====[Declaration of private data types]=====================================
15
16 typedef enum {
17    WIFI_STATE_INIT,
18    WIFI_STATE_SEND_AT,
19    WIFI_STATE_WAIT_AT,
20    WIFI_STATE_IDLE,
21    WIFI_STATE_ERROR
22 } wifiComState_t;
23
24 //=====[Declaration and initialization of public global objects]===============
25
26 UnbufferedSerial uartWifi( PE_8, PE_7, 115200 );
27
28 //=====[Declaration and initialization of private global variables]============
29
30 static const char responseOk[] = "OK";
31
32 static const char* wifiComExpectedResponse;
33 static wifiComState_t wifiComState;
34
35 static nonBlockingDelay_t wifiComDelay;
36
37 //=====[Declarations (prototypes) of private functions]=======================
38
39 static bool isExpectedResponse();
40 bool wifiComCharRead( char* receivedChar );
41 void wifiComStringWrite( const char* str );
```

***Code 11.5 Details of the implementation of the file wifi_com.cpp (Part 1/3).***

**NOTE:** The UART communication with the ESP-01 module uses the default configuration, which is 8 bits, no parity, and one stop bit. For this reason, these parameters are not configured.

On line 33, the variable *wifiComState* of type *wifiComState_t*, is declared, while on line 35, the variable *wifiComDelay* of type *nonBlockingDelay_t* is declared. The prototypes of the private functions *isExpectedResponse()*, *wifiComCharRead()*, and *wifiComStringWrite()* are declared from line 39 to line 41.

In Code 11.6, the implementation of *wifiComInit()* is shown on line 3. This function sets *wifiComState* to WIFI_STATE_INIT. On line 8, the implementation of *wifiComRestart()* can be seen, which only sets *wifiComState* to WIFI_STATE_INIT.

Line 13 shows the implementation of *wifiComUpdate()*. On line 15, *receivedCharWifiCom* is declared, which is a char variable that preserves its value between one call and another of *wifiComUpdate()* because it is declared as static. The FSM starts on line 17, with a switch over *wifiComState*. If its value is WIFI_STATE_INIT, then a non-blocking delay of five seconds is configured, and *wifiComState* is assigned with the value WIFI_STATE_SEND_AT.

```
1  //=====[Implementations of public functions]=================================
2
3  void wifiComInit()
4  {
5      wifiComState = WIFI_STATE_INIT;
6  }
7
8  void wifiComRestart()
9  {
10     wifiComState = WIFI_STATE_INIT;
11 }
12
13 void wifiComUpdate()
14 {
15    switch (wifiComState) {
16
17       case WIFI_STATE_INIT:
18          nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
19          wifiComState = WIFI_STATE_SEND_AT;
20       break;
21
22       case WIFI_STATE_SEND_AT:
23          if (nonBlockingDelayRead(&wifiComDelay)) {
24             wifiComStringWrite( "AT\r\n" );
25             wifiComExpectedResponse = responseOk;
26             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
27             wifiComState = WIFI_STATE_WAIT_AT;
28          }
29       break;
30
31       case WIFI_STATE_WAIT_AT:
```

```
32            if (isExpectedResponse()) {
33               nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
34               pcSerialComStringWrite("AT command responded ");
35               pcSerialComStringWrite("correctly\r\n");
36               wifiComState = WIFI_STATE_IDLE;
37            }
38            if (nonBlockingDelayRead(&wifiComDelay)) {
39               pcSerialComStringWrite("AT command not responded ");
40               pcSerialComStringWrite("correctly\r\n");
41               wifiComState = WIFI_STATE_ERROR;
42            }
43         break;
44
45         case WIFI_STATE_IDLE:
46         case WIFI_STATE_ERROR:
47         break;
48      }
49   }
```

***Code 11.6 Details of the implementation of the file wifi_com.cpp (Part 2/3).***

On line 22, the state WIFI_STATE_SEND_AT is implemented. If the non-blocking delay of five seconds has elapsed, then the command AT is written (line 24) and "OK" is assigned to *wifiComExpectedResponse* (line 25). A non-blocking delay of five seconds is initialized on line 26, and *wifiComState* is set to WIFI_STATE_WAIT_AT on line 27.

The implementation of WIFI_STATE_WAIT_AT is shown from line 31 to line 43. First, it is assessed whether there is a response and whether it is the expected response (line 32). If so, in lines 33 to 36 a new five-second non-blocking delay is started, the corresponding message is sent to the PC, and *wifiComState* is assigned to WIFI_STATE_IDLE. Otherwise, it is checked on line 38 if the five-second delay has expired. If so, the corresponding message is sent to the PC, and *wifiComState* is set to WIFI_STATE_ERROR. On lines 45 and 46, it can be seen that no actions are assigned to the WIFI_STATE_IDLE or WIFI_STATE_ERROR states.

Code 11.7 shows the implementation of some of the remaining functions of *wifi_com.cpp*. On line 3, it can be seen that *wifiComCharRead()* first checks if a character was received on *uartWifi* (connected to the ESP8266) (line 6), and if so, it writes the corresponding content on the memory address pointed by *receivedChar* (line 8). The returned value of *wifiComCharRead()* depends on whether there was a character available to be read (line 9) or not (line 11).

The function *wifiComStringWrite()* on line 14 is used to write the string pointed by its parameter *str* to the *uartWifi* object.

The implementation of *isExpectedResponse()* is shown between lines 19 and 37. First, three variables are declared: *responseStringPositionIndex*, to track the index of the position in the string corresponding to the response (note that it is declared as static); *charReceived*, to store the received char; and the Boolean variable *moduleResponse*, which is assigned the false state.

```
1  //=====[Implementations of private functions]=================================
2
3  bool wifiComCharRead( char* receivedChar )
4  {
5      char receivedCharLocal = '\0';
6      if( uartWifi.readable() ) {
7          uartWifi.read(&receivedCharLocal,1);
8          *receivedChar = receivedCharLocal;
9          return true;
10     }
11     return false;
12 }
13
14 void wifiComStringWrite( const char* str )
15 {
16     uartWifi.write( str, strlen(str) );
17 }
18
19 static bool isExpectedResponse()
20 {
21    static int responseStringPositionIndex = 0;
22    char charReceived;
23    bool moduleResponse = false;
24
25    if( wifiComCharRead(&charReceived) ){
26       if (charReceived == wifiComExpectedResponse[responseStringPositionIndex]) {
27          responseStringPositionIndex++;
28          if (wifiComExpectedResponse[responseStringPositionIndex] == '\0') {
29             responseStringPositionIndex = 0;
30             moduleResponse = true;
31          }
32       } else {
33          responseStringPositionIndex = 0;
34       }
35    }
36    return moduleResponse;
37 }
```

***Code 11.7 Details of the implementation of the file wifi_com.cpp (Part 3/3).***

Line 25 assesses whether there is a char available to be read on the *uartWifi* object. If so, line 26 assesses whether the received char is equal to the char that is expected at the corresponding position of the string *wifiComExpectedResponse*, and on line 27, *responseStringPositionIndex* is incremented by one. Line 28 assesses whether the current position of *wifiComExpectedResponse* is the null character. If so, *responseStringPositionIndex* is set to zero, and *moduleResponse* is set to the true state. If the received char is not the expected char (line 32), *responseStringPositionIndex* is set to zero. Finally, on line 36, *moduleResponse* is returned. Its value will be true if the expected response was received (recall line 30), and false otherwise.

## Proposed Exercise

1. How can more AT commands be added to the *wifi_com* module?

### Answer to the Exercise

1. In order to add more AT commands, the corresponding states should be added to *wifiComState_t* (line 16 of Code 11.5), and the corresponding state should be incorporated into *wifiComUpdate()*. This is shown in Example 11.2.

**Example 11.2: Configure the Credentials to Connect to the Wi-Fi Access Point**

### Objective

Include AT commands in the FSM in order to implement the connection with the Wi-Fi access point.

### Summary of the Expected Behavior

The NUCLEO board will send the commands "*AT*", "*AT+CWMODE=1*", "*AT+CWJAP*", and "*AT+CIFSR*" to the ESP-01 module (recall Section 11.2.2). It will be indicated on the serial terminal if the expected responses are received correctly by the NUCLEO board or not.

### Test the Proposed Solution on the Board

Import the project "Example 11.2" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Press "d" on the PC keyboard to set the Wi-Fi SSID of the access point that is to be used, and press "r" to set the Wi-Fi password. Then, press "a" to restart the Wi-Fi communication. If everything has worked correctly, after a few seconds the message "IP address assigned correctly" should appear on the serial terminal, and by pressing "p", the assigned IP address will be shown on the serial terminal. If this does not happen, check the connections, the access point, and the Wi-Fi credentials, and press "a" again to retry.

### Discussion of the Proposed Solution

The proposed solution is based on new states that are incorporated into the FSM of the *wifi_com* module. These new states implement the steps shown in Figure 11.8 to Figure 11.10.

### Implementation of the Proposed Solution

Table 11.7 shows the lines that were added to *pc_serial_com.cpp*. It can be seen that the private variables *numberOfCharsInApCredentials*, *ApSsid*, and *ApPassword* are created to manage the access point (AP) credentials. AP_SSID_MAX_LENGTH and AP_PASSWORD_MAX_LENGTH are defined in *wifi.h*, as shown below. Also, five new prototypes of private functions are declared in order to configure the SSID and the password and to get the assigned IP address.

In Table 11.8, the new lines that were added to *pcSerialComCommandUpdate()* and *availableCommands()* are shown. These lines are used to inform the user how to set the AP credentials and get the assigned IP address.

*Table 11.7 Sections in which lines were added to pc_serial_com.cpp.*

| Section | Lines that were added |
|---|---|
| Declaration and initialization of private global variables | `static int numberOfCharsInApCredentials = 0;`<br>`static char ApSsid[AP_SSID_MAX_LENGTH] = "";`<br>`static char ApPassword[AP_PASSWORD_MAX_LENGTH] = "";` |
| Declarations (prototypes) of private functions | `static void pcSerialComGetWiFiComApSsid( char receivedChar );`<br>`static void pcSerialComGetWiFiComApPassword( char receivedChar );`<br>`static void commandSetWifiComApSsid();`<br>`static void commandSetWifiComApPassword();`<br>`static void commandGetWifiComAssignedIp();` |

*Table 11.8 Functions in which lines were added in pc_serial_com.cpp.*

| Function | Lines that were added |
|---|---|
| static void pcSerialComCommandUpdate( char receivedChar ) | `case 'd': case 'D': commandSetWifiComApSsid(); break;`<br>`case 'r': case 'R': commandSetWifiComApPassword(); break;`<br>`case 'p': case 'P': commandGetWifiComAssignedIp(); break;` |
| static void availableCommands() | `pcSerialComStringWrite( "Press 'd' or 'D' to set Wi-Fi AP SSID\r\n" );`<br>`pcSerialComStringWrite( "Press 'r' or 'R' to set Wi-Fi AP Password\r\n" );`<br>`pcSerialComStringWrite( "Press 'p' or 'P' to get Wi-Fi assigned IP\r\n" );` |

The new declaration of the user-defined type *pcSerialComMode_t* is shown in Code 11.8. Two new valid values are incorporated: PC_SERIAL_GET_WIFI_AP_CREDENTIALS_SSID and PC_SERIAL_GET_WIFI_AP_CREDENTIALS_PASSWORD.

```
1  typedef enum{
2      PC_SERIAL_GET_FILE_NAME,
3      PC_SERIAL_COMMANDS,
4      PC_SERIAL_GET_CODE,
5      PC_SERIAL_SAVE_NEW_CODE,
6      PC_SERIAL_GET_WIFI_AP_CREDENTIALS_SSID,
7      PC_SERIAL_GET_WIFI_AP_CREDENTIALS_PASSWORD,
8  } pcSerialComMode_t;
```

*Code 11.8 New declaration of the type definition pcSerialComMode_t.*

The new implementation of *pcSerialComUpdate()* is shown in Code 11.9. The new program code is between lines 18 and 23. It is used to get the AP credentials using the functions *pcSerialComGetWiFiComApSsid()* and *pcSerialComGetWiFiComApPassword()*, which are discussed below.

```
void pcSerialComUpdate()
{
    char receivedChar = pcSerialComCharRead();
    if( receivedChar != '\0' ) {
        switch ( pcSerialComMode ) {
            case PC_SERIAL_GET_FILE_NAME:
                pcSerialComGetFileName( receivedChar );
                break;
            case PC_SERIAL_COMMANDS:
                pcSerialComCommandUpdate( receivedChar );
                break;
            case PC_SERIAL_GET_CODE:
                pcSerialComGetCodeUpdate( receivedChar );
                break;
            case PC_SERIAL_SAVE_NEW_CODE:
                pcSerialComSaveNewCodeUpdate( receivedChar );
                break;
            case PC_SERIAL_GET_WIFI_AP_CREDENTIALS_SSID:
                pcSerialComGetWiFiComApSsid( receivedChar );
                break;
            case PC_SERIAL_GET_WIFI_AP_CREDENTIALS_PASSWORD:
                pcSerialComGetWiFiComApPassword( receivedChar );
                break;
            default:
                pcSerialComMode = PC_SERIAL_COMMANDS;
                break;
        }
    }
}
```

***Code 11.9 Implementation of new cases in Serial_com pcSerialComUpdate().***

In Code 11.10, the new functions that are incorporated into *pc_serial_com.cpp* are shown. The functions *pcSerialComGetWiFiComApSsid()* and *pcSerialComGetWiFiComApPassword()* are both called from *pcSerialComUpdate()*, as was shown in Code 11.9. These functions are used to ask the user for the AP credentials, which are stored in *ApSsid* and *ApPassword*. The credentials are configured into the ESP-01 module using the functions *wifiComSetWiFiComApSsid()* and *wifiComSetWiFiComApPassword()*, which are discussed below. The number of characters is stored in *numberOfCharsInApCredentials*.

The implementation of the functions that were mentioned in Table 11.8 are shown in Code 11.10. These functions are used to ask the user for the SSID and the password of the AP, and also to report the assigned IP address. To get the assigned IP address, the function *wifiComGetIpAddress()* is used, which is discussed below. Notice that *pcSerialComMode* is set in *commandSetWifiComApSsid()* and *commandSetWifiComApPassword()*, and *numberOfCharsInApCredentials* is set to zero in both functions.

```
1  static void commandSetWifiComApSsid()
2  {
3      pcSerialComStringWrite("\r\nPlease provide the SSID of the Wi-Fi ");
4      pcSerialComStringWrite("Access Point and press the Enter key\r\n");
5      pcSerialComStringWrite("> ");
6      pcSerialComMode = PC_SERIAL_GET_WIFI_AP_CREDENTIALS_SSID;
7      numberOfCharsInApCredentials = 0;
8  }
9
10 static void commandSetWifiComApPassword()
11 {
12     pcSerialComStringWrite("\r\nPlease provide the Password of the Wi-Fi ");
13     pcSerialComStringWrite("Access Point and press the Enter key\r\n");
14     pcSerialComStringWrite("> ");
15     pcSerialComMode = PC_SERIAL_GET_WIFI_AP_CREDENTIALS_PASSWORD;
16     numberOfCharsInApCredentials = 0;
17 }
18
19 static void commandGetWifiComAssignedIp()
20 {
21     pcSerialComStringWrite( "The assigned IP is: " );
22     pcSerialComStringWrite( wifiComGetIpAddress() );
23     pcSerialComStringWrite( "\r\n" );
24 }
25
26 static void pcSerialComGetWiFiComApSsid( char receivedChar )
27 {
28     if ( (receivedChar == '\r') &&
29          (numberOfCharsInApCredentials < AP_SSID_MAX_LENGTH) ) {
30         pcSerialComMode = PC_SERIAL_COMMANDS;
31         ApSsid[numberOfCharsInApCredentials] = '\0';
32         wifiComSetWiFiComApSsid(ApSsid);
33         pcSerialComStringWrite( "\r\nWi-Fi Access Point SSID configured\r\n\r\n" );
34     } else {
35         ApSsid[numberOfCharsInApCredentials] = receivedChar;
36         pcSerialComCharWrite( receivedChar );
37         numberOfCharsInApCredentials++;
38     }
39 }
40
41 static void pcSerialComGetWiFiComApPassword( char receivedChar )
42 {
43     if ( (receivedChar == '\r') &&
44          (numberOfCharsInApCredentials < AP_PASSWORD_MAX_LENGTH) ) {
45         pcSerialComMode = PC_SERIAL_COMMANDS;
46         ApPassword[numberOfCharsInApCredentials] = '\0';
47         wifiComSetWiFiComApPassword(ApPassword);
48        pcSerialComStringWrite( "\r\nWi-Fi Access Point password configured\r\n\r\n" );
49     } else {
50         ApPassword[numberOfCharsInApCredentials] = receivedChar;
51         pcSerialComStringWrite( "*" );
52         numberOfCharsInApCredentials++;
53     }
54 }
```

***Code 11.10 Implementation of new private functions in pc_serial_com.cpp.***

In Code 11.11 the new implementation of *wifi_com.h* is shown. The definitions that were used in Table 11.7 are shown on lines 8 and 9. It can be seen that the three public functions that were introduced in Code 11.10 are declared in lines 13 to 15. The implementation of these three functions

is in the file *wifi_com.cpp* and is shown in Code 11.12. It can be seen that *wifiComSetWiFiComApSsid()* uses the function *strcnpy()* to copy the content of the string *ApSsid* into the string *wifiComApSsid* in order to avoid the user input causing buffer overflow issues, as discussed in Chapter 4. The function *wifiComSetWiFiComApPassword()* makes a copy of *ApPassword* into *wifiComApPassword*. Finally, *wifiComGetIpAddress()* returns a pointer to the string *wifiComIpAddress*, which is introduced below.

```
1    //=====[#include guards - begin]===============================================
2
3    #ifndef _WIFI_COM_
4    #define _WIFI_COM_
5
6    //=====[Declaration of public defines]=========================================
7
8    #define AP_SSID_MAX_LENGTH     (32 + 1)
9    #define AP_PASSWORD_MAX_LENGTH (63 + 1)
10
11   //=====[Declarations (prototypes) of public functions]=========================
12
13   void wifiComSetWiFiComApSsid( char * ApSsid );
14   void wifiComSetWiFiComApPassword( char * ApPassword );
15   char * wifiComGetIpAddress();
16
17   void wifiComRestart();
18   void wifiComInit();
19   void wifiComUpdate();
20
21   //=====[#include guards - end]=================================================
22
23   #endif /* _WIFI_COM_ */
```

***Code 11.11 New implementation of wifi_com.h.***

```
 1  void wifiComSetWiFiComApSsid( char * ApSsid )
 2  {
 3      strncpy(wifiComApSsid, ApSsid, AP_SSID_MAX_LENGTH);
 4  }
 5
 6  void wifiComSetWiFiComApPassword( char * ApPassword )
 7  {
 8      strncpy(wifiComApPassword, ApPassword, AP_PASSWORD_MAX_LENGTH );
 9  }
10
11  char * wifiComGetIpAddress()
12  {
13     return wifiComIpAddress;
14  }
```

***Code 11.12 Implementation of the public functions of the wifi_com module.***

The lines that were added to *wifi_com.cpp* are shown in Table 11.9. A definition of DELAY_10_SECONDS as 10000 is added, as well as a definition of IP_MAX_LENGTH. Also seven private global strings are declared. Four of them are used to implement specific steps of the FSM, as discussed below. The other three are used to store information regarding the AP, also discussed below.

*Table 11.9 Sections in which lines were added to wifi_com.cpp.*

| Section | Lines that were added |
|---|---|
| Declaration of private defines | `#define DELAY_10_SECONDS 10000`<br>`#define IP_MAX_LENGTH (15 + 1)` |
| Declaration and initialization of private global variables | `static const char responseCwjapOk[] = "+CWJAP:";`<br>`static const char responseCwjap1[] = "WIFI CONNECTED";`<br>`static const char responseCwjap2[] = "WIFI GOT IP";`<br>`static const char responseCifsr[] = "+CIFSR:STAIP,\"";`<br>`static char wifiComApSsid[AP_SSID_MAX_LENGTH] = "";`<br>`static char wifiComApPassword[AP_PASSWORD_MAX_LENGTH] = "";`<br>`static char wifiComIpAddress[IP_MAX_LENGTH];` |

As was mentioned in the section "Summary of the Expected Behavior," in this example the NUCLEO board sends the commands "*AT*", "*AT+CWMODE=1*", "*AT+CWJAP*", and "*AT+CIFSR*" to the ESP-01 module. To handle this functionality, new states are incorporated into the FSM, as shown in Code 11.13 (lines 5 to 14).

```
 1  typedef enum {
 2     WIFI_STATE_INIT,
 3     WIFI_STATE_SEND_AT,
 4     WIFI_STATE_WAIT_AT,
 5     WIFI_STATE_SEND_CWMODE,
 6     WIFI_STATE_WAIT_CWMODE,
 7     WIFI_STATE_SEND_CWJAP_IS_SET,
 8     WIFI_STATE_WAIT_CWJAP_IS_SET,
 9     WIFI_STATE_SEND_CWJAP_SET,
10     WIFI_STATE_WAIT_CWJAP_SET_1,
11     WIFI_STATE_WAIT_CWJAP_SET_2,
12     WIFI_STATE_SEND_CIFSR,
13     WIFI_STATE_WAIT_CIFSR,
14     WIFI_STATE_LOAD_IP,
15     WIFI_STATE_IDLE,
16     WIFI_STATE_ERROR
17  } wifiComState_t;
```

*Code 11.13 New declaration of the user-defined type wifiComState_t.*

The new implementation of the FSM is shown in Code 11.14 to Code 11.16. On lines 3 and 4 of Code 11.14, two new static variables are declared: *receivedCharWifiCom* and *IpStringPositionIndex*. Lines 6 to 32 of Code 11.14 are the same as in Example 11.1 (Code 11.6). The "*AT+CWMODE=1*" command is implemented from lines 34 to 41. The corresponding response is expected using the FSM that is implemented from lines 43 to 53. Note that the program code that is used to implement the "*AT+CWMODE=1*" command is very similar to the program code used to implement the "*AT*" command that was discussed in Example 11.1. For this reason, lines 34 to 53 are not further discussed here.

The implementation of the "*AT+CWJAP*" command is shown between lines 55 and 62 of Code 11.14 and in lines 1 to 51 of Code 11.15. Firstly, the state WIFI_STATE_SEND_CWJAP_IS_SET sends "*AT+CWJAP?*" to the ESP-01 module in order to determine if the AP credentials are configured. Next, the WIFI_STATE_WAIT_CWJAP_IS_SET state assesses if the response is "OK". If so, the next state is WIFI_STATE_SEND_CIFSR. Otherwise, the next state is WIFI_STATE_SEND_CWJAP_SET, where

the AP credentials are sent to the ESP-01, the expected response is set in line 20 of Code 11.15 to *responseCwjap1*, which is "WIFI CONNECTED", and the next state is set to WIFI_STATE_WAIT_CWJAP_SET_1. The WIFI_STATE_WAIT_CWJAP_SET_1 state assesses if "WIFI CONNECTED" is received. If so, the expected response is set to *responseCwjap2*, which is "WIFI GOT IP", and the next state is set to WIFI_STATE_WAIT_CWJAP_SET_2. Finally, the WIFI_STATE_WAIT_CWJAP_SET_2 state assesses if "WIFI GOT IP" is received. If so, the next state is set to WIFI_STATE_SEND_CIFSR.

The implementation of the "*AT+CIFSR*" command is shown between lines 53 and 60 of Code 11.15 and in lines 1 to 24 of Code 11.16. First, the state WIFI_STATE_SEND_CIFSR sends "*AT+CIFSR*" to the ESP-01 module in order to receive the assigned IP address. The expected response is set in line 56 to *responseCifsr*, which is "+CIFSR:STAIP,\", and the next state is set to WIFI_STATE_WAIT_CIFSR. Next, the WIFI_STATE_WAIT_CIFSR state assesses if "+CIFSR:STAIP,\" is received. If so, the next state is set to WIFI_STATE_LOAD_IP, and *IpStringPositionIndex* is set to zero. In the state WIFI_STATE_LOAD_IP, the assigned IP address is read and loaded into *wifiComIpAddress*. The assessment "*IpStringPositionIndex < IP_MAX_LENGTH*" is used to avoid buffer overflow issues, as discussed in Chapter 4, if for any reason the ESP-01 module sends more IP characters than expected. Then, the message "IP address assigned correctly" is sent to the serial terminal (line 20).

Lastly, lines 26 to 28 of Code 11.16 are the same as in the previous implementation of *wifiComUpdate()* in Example 11.1. In this way, the FSM will remain in WIFI_STATE_IDLE or case WIFI_STATE_ERROR if those states are reached. Note that no specific message is displayed in these situations, in order to simplify the implementation of this example.

```
 1  void wifiComUpdate()
 2  {
 3     static char receivedCharWifiCom;
 4     static int IpStringPositionIndex;
 5
 6     switch (wifiComState) {
 7
 8        case WIFI_STATE_INIT:
 9           nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
10           wifiComState = WIFI_STATE_SEND_AT;
11           break;
12
13        case WIFI_STATE_SEND_AT:
14           if (nonBlockingDelayRead(&wifiComDelay)) {
15              wifiComStringWrite( "AT\r\n" );
16              wifiComExpectedResponse = responseOk;
17              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
18              wifiComState = WIFI_STATE_WAIT_AT;
19           }
20           break;
21
22        case WIFI_STATE_WAIT_AT:
23           if (isExpectedResponse()) {
24              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
25              wifiComState = WIFI_STATE_SEND_CWMODE;
26           }
27           if (nonBlockingDelayRead(&wifiComDelay)) {
28              pcSerialComStringWrite("AT command not responded ");
```

```
29              pcSerialComStringWrite("correctly\r\n");
30              wifiComState = WIFI_STATE_ERROR;
31           }
32           break;
33
34        case WIFI_STATE_SEND_CWMODE:
35           if (nonBlockingDelayRead(&wifiComDelay)) {
36              wifiComStringWrite( "AT+CWMODE=1\r\n" );
37              wifiComExpectedResponse = responseOk;
38              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
39              wifiComState = WIFI_STATE_WAIT_CWMODE;
40           }
41           break;
42
43        case WIFI_STATE_WAIT_CWMODE:
44           if (isExpectedResponse()) {
45              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
46              wifiComState = WIFI_STATE_SEND_CWJAP_IS_SET;
47           }
48           if (nonBlockingDelayRead(&wifiComDelay)) {
49              pcSerialComStringWrite("AT+CWMODE=1 command not ");
50              pcSerialComStringWrite("responded correctly\r\n");
51              wifiComState = WIFI_STATE_ERROR;
52           }
53           break;
54
55        case WIFI_STATE_SEND_CWJAP_IS_SET:
56           if (nonBlockingDelayRead(&wifiComDelay)) {
57              wifiComStringWrite( "AT+CWJAP?\r\n" );
58              wifiComExpectedResponse = responseCwjapOk;
59              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
60              wifiComState = WIFI_STATE_WAIT_CWJAP_IS_SET;
61           }
62           break;
```

*Code 11.14 New implementation of wifiComUpdate() (Part 1/3).*

```
 1        case WIFI_STATE_WAIT_CWJAP_IS_SET:
 2           if (isExpectedResponse()) {
 3              wifiComExpectedResponse = responseOk;
 4              wifiComState = WIFI_STATE_SEND_CIFSR;
 5           }
 6           if (nonBlockingDelayRead(&wifiComDelay)) {
 7              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
 8              wifiComState = WIFI_STATE_SEND_CWJAP_SET;
 9           }
10           break;
11
12        case WIFI_STATE_SEND_CWJAP_SET:
13           if (nonBlockingDelayRead(&wifiComDelay)) {
14              wifiComStringWrite( "AT+CWJAP=\"" );
15              wifiComStringWrite( wifiComApSsid );
16              wifiComStringWrite( "\",\"" );
17              wifiComStringWrite( wifiComApPassword );
18              wifiComStringWrite( "\"" );
19              wifiComStringWrite( "\r\n" );
20              wifiComExpectedResponse = responseCwjap1;
21              nonBlockingDelayWrite(&wifiComDelay, DELAY_10_SECONDS);
22              wifiComState = WIFI_STATE_WAIT_CWJAP_SET_1;
23           }
```

```
            break;

         case WIFI_STATE_WAIT_CWJAP_SET_1:
            if (isExpectedResponse()) {
               wifiComExpectedResponse = responseCwjap2;
               wifiComState = WIFI_STATE_WAIT_CWJAP_SET_2;
            }
            if (nonBlockingDelayRead(&wifiComDelay)) {
               pcSerialComStringWrite("Error in state: ");
               pcSerialComStringWrite("WIFI_STATE_WAIT_CWJAP_SET_1\r\n");
               pcSerialComStringWrite("Check Wi-Fi AP credentials ");
               pcSerialComStringWrite("and restart\r\n");
                  wifiComState = WIFI_STATE_ERROR;
            }
            break;

         case WIFI_STATE_WAIT_CWJAP_SET_2:
            if (isExpectedResponse()) {
               wifiComState = WIFI_STATE_SEND_CIFSR;
            }
            if (nonBlockingDelayRead(&wifiComDelay)) {
               pcSerialComStringWrite("Error in state: ");
               pcSerialComStringWrite("WIFI_STATE_WAIT_CWJAP_SET_2\r\n");
               pcSerialComStringWrite("Check Wi-Fi AP credentials ");
               pcSerialComStringWrite("and restart\r\n");
               wifiComState = WIFI_STATE_ERROR;
            }
         break;

         case WIFI_STATE_SEND_CIFSR:
            if (nonBlockingDelayRead(&wifiComDelay)) {
               wifiComStringWrite( "AT+CIFSR\r\n" );
               wifiComExpectedResponse = responseCifsr;
               nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
               wifiComState = WIFI_STATE_WAIT_CIFSR;
            }
            break;
```

*Code 11.15 New implementation of wifiComUpdate() (Part 2/3).*

```
        case WIFI_STATE_WAIT_CIFSR:
            if (isExpectedResponse()) {
               wifiComState = WIFI_STATE_LOAD_IP;
               IpStringPositionIndex = 0;
            }
            if (nonBlockingDelayRead(&wifiComDelay)) {
               pcSerialComStringWrite("AT+CIFSR command not responded ");
               pcSerialComStringWrite("correctly\r\n");
               wifiComState = WIFI_STATE_ERROR;
            }
         break;

         case WIFI_STATE_LOAD_IP:
            if (wifiComCharRead(&receivedCharWifiCom)) {
               if ( (receivedCharWifiCom != '"') &&
                  (IpStringPositionIndex < IP_MAX_LENGTH) ) {
                  wifiComIpAddress[IpStringPositionIndex] = receivedCharWifiCom;
                  IpStringPositionIndex++;
               } else {
                  wifiComIpAddress[IpStringPositionIndex] = '\0';
                  pcSerialComStringWrite("IP address assigned correctly\r\n\r\n");
```

```
22                  wifiComState = WIFI_STATE_IDLE;
23              }
24           }
25         break;
26
27         case WIFI_STATE_IDLE:
28         case WIFI_STATE_ERROR:
29         break;
30      }
31   }
```

***Code 11.16 New implementation of wifiComUpdate() (Part 3/3).***

### Proposed Exercise

1. Once the ESP-01 module is connected to the AP, what should be done in order to serve a web page?

### Answer to the Exercise

1. The AT commands shown from Figure 11.11 to Figure 11.19 must be implemented. This is shown in Example 11.3.

**Example 11.3: Serve a Simple Web Page using the Wi-Fi Connection**

### Objective

Include AT commands in the FSM in order to serve a web page using the Wi-Fi access point.

### Summary of the Expected Behavior

The NUCLEO board will send the commands "*AT*", "*AT+CWMODE=1*", "*AT+CWJAP*", "*AT+CIFSR*", "*AT+CIPMUX=1*", "*AT+CIPSERVER=1,80*", "*AT+CIPSTATUS*", "*AT+CIPSEND*", the HTML document, and the command "*AT+CIPCLOSE*" to the ESP-01 module (recall Section 11.2.2). It will be indicated on the serial terminal if the expected responses are received correctly by the NUCLEO board or not. If everything works as expected, the web page that was shown in Figure 11.20 should be displayed on the web browser.

### Test the Proposed Solution on the Board

Import the project "Example 11.3" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Repeat the same steps as in Example 11.2. Press "p" to get the IP address assigned to the ESP-01 module. Enter this IP in a web browser. The web page that was shown in Figure 11.20 should be displayed in the web browser. If this does not happen, check the connections and press "a" to retry.

## Discussion of the Proposed Solution

The proposed solution is based on new states that are incorporated into the FSM of the *wifi_com* module. These new states implement the steps shown in Figure 11.11 to Figure 11.19.

## Implementation of the Proposed Solution

In this example, the *wifi_com* module incorporates many AT commands. To handle this functionality, new states are incorporated into the FSM as shown in Code 11.17 (lines 15 to 29).

```
 1  typedef enum {
 2     WIFI_STATE_INIT,
 3     WIFI_STATE_SEND_AT,
 4     WIFI_STATE_WAIT_AT,
 5     WIFI_STATE_SEND_CWMODE,
 6     WIFI_STATE_WAIT_CWMODE,
 7     WIFI_STATE_SEND_CWJAP_IS_SET,
 8     WIFI_STATE_WAIT_CWJAP_IS_SET,
 9     WIFI_STATE_SEND_CWJAP_SET,
10     WIFI_STATE_WAIT_CWJAP_SET_1,
11     WIFI_STATE_WAIT_CWJAP_SET_2,
12     WIFI_STATE_SEND_CIFSR,
13     WIFI_STATE_WAIT_CIFSR,
14     WIFI_STATE_LOAD_IP,
15     WIFI_STATE_SEND_CIPMUX,
16     WIFI_STATE_WAIT_CIPMUX,
17     WIFI_STATE_SEND_CIPSERVER,
18     WIFI_STATE_WAIT_CIPSERVER,
19     WIFI_STATE_SEND_CIPSTATUS,
20     WIFI_STATE_WAIT_CIPSTATUS_STATUS_3,
21     WIFI_STATE_WAIT_CIPSTATUS,
22     WIFI_STATE_WAIT_GET_ID,
23     WIFI_STATE_WAIT_CIPSTATUS_OK,
24     WIFI_STATE_SEND_CIPSEND,
25     WIFI_STATE_WAIT_CIPSEND,
26     WIFI_STATE_SEND_HTML,
27     WIFI_STATE_WAIT_HTML,
28     WIFI_STATE_SEND_CIPCLOSE,
29     WIFI_STATE_WAIT_CIPCLOSE
30     WIFI_STATE_IDLE,
31     WIFI_STATE_ERROR
32  } wifiComState_t;
```

***Code 11.17 New declaration of the user-defined type wifiComState_t.***

Two new variables are declared in *wifiComUpdate()*, as shown in Table 11.10. The variable *lengthOfHtmlCode* is used together with the "*AT+CIPSEND*" command in order to indicate the length in bytes of the HTTP that is being sent to the web browser (recall Figure 11.17). The array *strToSend* is used as a string where the message to be sent is stored in the cases of the "*AT+CIPSEND*" and "*AT+CIPCLOSE*" commands, because these commands require some parameters (recall Figure 11.17 and Figure 11.19).

***Table 11.10 Functions in which lines were added in wifi_com.cpp.***

| Function | Lines that were added |
|---|---|
| void wifiComUpdate() | `int lengthOfHtmlCode;`<br>`char strToSend[50] = "";` |

In Table 11.11, the new private global variables that are declared in *wifi_com.cpp* are shown. The strings stored in *responseStatus3* and *responseCipstatus* are used to implement the command "*AT+CIPSTATUS*". The string *responseSendOk* is used to implement the command "*AT+CIPSEND*", while the string *responseCipclose* is used to implement the "*AT+CIPCLOSE=0*" command. The integer variable *currentConnectionId* is used to get the identifier (ID) of the connection that is established. Lastly, the HTML code that is used to implement the web page that is shown in the web browser is stored in the string *htmlCode* (recall Code 11.1, where this HTML is introduced).

***Table 11.11 Sections in which lines were added to wifi_com.cpp.***

| Section | Lines that were added |
|---|---|
| Declaration and initialization of private global variables | `static const char responseStatus3[] = "STATUS:3";`<br>`static const char responseCipstatus[] = "+CIPSTATUS:";`<br>`static const char responseSendOk[] = "SEND OK";`<br>`static const char responseCipclose[] = "CLOSED";`<br>`static int currentConnectionId;`<br>`static const char htmlCode [] =`<br>`"<!doctype html> <html> <body> Hello! </body> </html>"` |

The implementation of the new states of the FSM in *wifiComUpdate()* is shown in Code 11.18 to Code 11.20. In Code 11.18, the implementation of "*AT+CIPMUX=1*" and "*AT+CIPSERVER=1,80*" is shown. The implementation of these commands is not further discussed because it is very similar to the implementation of "*AT+CWMODE=1*", which was explained in Example 11.2.

Code 11.19 shows the implementation of the "*AT+CIPSTATUS*" command. The sequence is as shown in Figure 11.16: first, "*AT+CIPSTATUS*" is sent to the ESP-01 module (line 3), then it is assessed whether the expected response "STATUS:3" is obtained (line 11). After this, it is assessed if the ESP-01 module sends the message "+CIPSTATUS:" (line 23), then the current connection ID is read (line 34), and, finally, it is checked if "OK" has been sent by the ESP-01 module.

Code 11.20 shows the implementation of the "*AT+CIPSEND*" and "*AT+CIPCLOSE*" commands. On line 3, the "*AT+CIPSEND*" command is prepared with the corresponding parameters (recall Figure 11.17). For this purpose, the current connection ID and the length in bytes of the HTML web page code, which is obtained in line 2 using *strlen()*, are used. The "*AT+CIPSEND*" command is sent on line 5. Line 11 checks if the ESP-01 module response is "OK". If so, the FSM moves to the WIFI_STATE_SEND_HTML state (line 13). Otherwise, it returns to the WIFI_STATE_SEND_CIPSTATUS state (line 17).

The HTML document is sent on line 22, and the "SEND OK" response is checked on line 28. If this response is obtained, it moves to the state WIFI_STATE_SEND_CIPCLOSE (line 30). If it is not obtained, the FSM moves back to the WIFI_STATE_SEND_CIPSEND state (line 34). The "*AT+CIPCLOSE*" command is implemented between lines 38 and 57. It is important to note that if everything works as expected (i.e., "OK" is received), the next state is set to WIFI_STATE_SEND_CIPSTATUS (line 51); if not, the next state is also set to WIFI_STATE_SEND_CIPSTATUS (line 55). This is done this way because in either scenario the next step is to wait for a new web page request, and this is done in WIFI_STATE_SEND_CIPSTATUS by means of asking the ESP-01 module about the CIPSTATUS.

```
1       case WIFI_STATE_SEND_CIPMUX:
2           if (nonBlockingDelayRead(&wifiComDelay)) {
3              wifiComStringWrite( "AT+CIPMUX=1\r\n" );
4              wifiComExpectedResponse = responseOk;
5              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
6              wifiComState = WIFI_STATE_WAIT_CIPMUX;
7           }
8           break;
9
10       case WIFI_STATE_WAIT_CIPMUX:
11          if (isExpectedResponse()) {
12             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
13             wifiComState = WIFI_STATE_SEND_CIPSERVER;
14          }
15          if (nonBlockingDelayRead(&wifiComDelay)) {
16             pcSerialComStringWrite("AT+CIPMUX=1 command not responded ");
17             pcSerialComStringWrite("correctly\r\n\r\n");
18             wifiComState = WIFI_STATE_ERROR;
19          }
20          break;
21
22       case WIFI_STATE_SEND_CIPSERVER:
23          if (nonBlockingDelayRead(&wifiComDelay)) {
24             wifiComStringWrite( "AT+CIPSERVER=1,80\r\n" );
25             wifiComExpectedResponse = responseOk;
26             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
27             wifiComState = WIFI_STATE_WAIT_CIPSERVER;
28          }
29          break;
30
31       case WIFI_STATE_WAIT_CIPSERVER:
32          if (isExpectedResponse()) {
33             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
34             wifiComState = WIFI_STATE_SEND_CIPSTATUS;
35          }
36          if (nonBlockingDelayRead(&wifiComDelay)) {
37             pcSerialComStringWrite("AT+CIPSERVER=1,80 command not responded ");
38             pcSerialComStringWrite("correctly\r\n\r\n");
39             wifiComState = WIFI_STATE_ERROR;
40          }
41          break;
```

***Code 11.18 Implementation of the new states in wifiComUpdate() (Part 1/3).***

```
      case WIFI_STATE_SEND_CIPSTATUS:
          if (nonBlockingDelayRead(&wifiComDelay)) {
             wifiComStringWrite( "AT+CIPSTATUS\r\n" );
             wifiComExpectedResponse = responseStatus3;
             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
             wifiComState = WIFI_STATE_WAIT_CIPSTATUS_STATUS_3;
          }
          break;

       case WIFI_STATE_WAIT_CIPSTATUS_STATUS_3:
          if (isExpectedResponse()) {
             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
             wifiComExpectedResponse = responseCipstatus;
             wifiComState = WIFI_STATE_WAIT_CIPSTATUS;
          }
          if (nonBlockingDelayRead(&wifiComDelay)) {
             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
             wifiComState = WIFI_STATE_SEND_CIPSTATUS;
          }
          break;

       case WIFI_STATE_WAIT_CIPSTATUS:
          if (isExpectedResponse()) {
             wifiComState = WIFI_STATE_WAIT_GET_ID;
          }
          if (nonBlockingDelayRead(&wifiComDelay)) {
             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
             wifiComState = WIFI_STATE_SEND_CIPSTATUS;
          }
          break;

       case WIFI_STATE_WAIT_GET_ID:
          if( wifiComCharRead(&receivedCharWifiCom) ){
             currentConnectionId = receivedCharWifiCom;
             wifiComExpectedResponse = responseOk;
             wifiComState = WIFI_STATE_WAIT_CIPSTATUS_OK;
          }
          break;

       case WIFI_STATE_WAIT_CIPSTATUS_OK:
          if (isExpectedResponse()) {
             wifiComState = WIFI_STATE_SEND_CIPSEND;
          }
          if (nonBlockingDelayRead(&wifiComDelay)) {
             nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
             wifiComState = WIFI_STATE_SEND_CIPSTATUS;
          }
          break;
```

***Code 11.19 Implementation of the new states in wifiComUpdate() (Part 2/3).***

```
        case WIFI_STATE_SEND_CIPSEND:
           lengthOfHtmlCode = (strlen(htmlCode));
           sprintf( strToSend, "AT+CIPSEND=%c,%d\r\n", currentConnectionId,
                                                          lengthOfHtmlCode );
           wifiComStringWrite( strToSend );
           wifiComState = WIFI_STATE_WAIT_CIPSEND;
           wifiComExpectedResponse = responseOk;
           break;

        case WIFI_STATE_WAIT_CIPSEND:
           if (isExpectedResponse()) {
              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
              wifiComState = WIFI_STATE_SEND_HTML;
           }
           if (nonBlockingDelayRead(&wifiComDelay)) {
              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
              wifiComState = WIFI_STATE_SEND_CIPSTATUS;
           }
           break;

        case WIFI_STATE_SEND_HTML:
           wifiComStringWrite( htmlCode );
           wifiComState = WIFI_STATE_WAIT_HTML;
           wifiComExpectedResponse = responseSendOk;
           break;

        case WIFI_STATE_WAIT_HTML:
           if (isExpectedResponse()) {
              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
              wifiComState = WIFI_STATE_SEND_CIPCLOSE;
           }
           if (nonBlockingDelayRead(&wifiComDelay)) {
              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
              wifiComState = WIFI_STATE_SEND_CIPSEND;
           }
           break;

        case WIFI_STATE_SEND_CIPCLOSE:
           if (nonBlockingDelayRead(&wifiComDelay)) {
              sprintf( strToSend, "AT+CIPCLOSE=%c\r\n", currentConnectionId );
              wifiComStringWrite( strToSend );
              wifiComExpectedResponse  = responseCipclose;
              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
              wifiComState = WIFI_STATE_WAIT_CIPCLOSE;
           }
           break;

        case WIFI_STATE_WAIT_CIPCLOSE:
           if (isExpectedResponse()) {
              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
              wifiComState = WIFI_STATE_SEND_CIPSTATUS;
           }
           if (nonBlockingDelayRead(&wifiComDelay)) {
              nonBlockingDelayWrite(&wifiComDelay, DELAY_5_SECONDS);
              wifiComState = WIFI_STATE_SEND_CIPSTATUS;
           }
           break;

        case WIFI_STATE_IDLE:
        case WIFI_STATE_ERROR:
           break;
     }
  }
```

*Code 11.20 Implementation of the new states in wifiComUpdate() (Part 3/3).*

### Proposed Exercise

1. How can the web page be configured in order to show some relevant data about the smart home system?

### Answer to the Exercise

1. The HTML document that is served must be modified in order to include some information about the smart home system. This is shown in Example 11.4.

**Example 11.4: Serve a Web Page that Shows the Smart Home System Information**

### Objective

Include in the web page the status of different elements of the smart home system.

### Summary of the Expected Behavior

The NUCLEO board will serve a web page, as in Example 11.3, but in this case the web page will contain relevant information about the smart home system: the temperature expressed in degrees Celsius; the status of the over temperature, gas, and motion detectors; and the status of the alarm, Incorrect code LED, and System blocked LED, as shown in Figure 11.6.

### Test the Proposed Solution on the Board

Import the project "Example 11.4" using the URL available in [3], build the project, and drag the *.bin* file onto the NUCLEO board. Follow the same steps as in Example 11.3. If everything worked correctly, the web page that was introduced in Figure 11.6 should be displayed in the web browser. Otherwise, check the connections and the access point credentials, and press "a" to retry.

### Discussion of the Proposed Solution

The proposed solution is based on the same states of the FSM that were introduced in previous examples. The difference is that in this example more information is shown in the web page served by the smart home system.

### Implementation of the Proposed Solution

In order to show more information in the web page served by the smart home system, some lines were added in different sections of *wifi_com.cpp*, as shown in Table 11.12. It can be seen that the libraries regarding the temperature sensor, siren, fire alarm, motion sensor, and user interface were included. Two new definitions are also made: BEGIN_USER_LINE and END_USER_LINE. The former is used to indicate the beginning of a paragraph by means of . The latter is used to indicate the ending of the paragraph by means of .

***Table 11.12 Sections in which lines were added to wifi_com.cpp.***

| Section | Lines that were added |
|---|---|
| Libraries | `#include "temperature_sensor.h"`<br>`#include "siren.h"`<br>`#include "fire_alarm.h"`<br>`#include "motion_sensor.h"`<br>`#include "user_interface.h"` |
| Declaration of private defines | `#define BEGIN_USER_LINE "<p>"`<br>`#define END_USER_LINE "</p>"` |
| Declaration and initialization of private global variables | `static char stateString[4] = "";`<br>`static const char htmlCodeHeader [] =`<br>`"<!doctype html>"`<br>`"<html> <head> <title>Smart Home System</title> </head>"`<br>`"<body style=\"text-align: center;\">"`<br>`"<h1 style=\"color: #0000ff;\">Smart Home System</h1>"`<br>`"<div style=\"font-weight: bold\">";`<br>`static const char htmlCodeFooter [] = "</div> </body> </html>";`<br>`static char htmlCodeBody[450] = "";` |
| Declarations (prototypes) of private functions | `void wifiComWebPageDataUpdate();`<br>`char * stateToString( bool state );` |

Table 11.12 shows that some new private variables were declared. *stateString* will be used to store a string that will indicate the status of the different elements of the smart home system (i.e., "ON" or "OFF"). *htmlCodeHeader* is used to store the header of the HTML code. Its first lines ("*<!doctype html> <html> <head>*") are the same as in Example 11.3. Next, "*<title>Smart Home System</title>*" is used to assign a title to the web page. Then, center-aligned text is configured for the body of the document. After this, "Smart Home System" is printed using heading size 1 (i.e., h1) and blue color (#0000ff). A division or section in the HTML code is opened where bold font is set. The *htmlCodeFooter* string is used to close the * *, *<body>*, and *<html>* tags that were opened in *htmlCodeHeader*. Finally, the string *htmlCodeBody* is used to store the body of the HTML code. The prototypes of the new private functions *wifiComWebPageDataUpdate()* and *stateToString()* are declared in *wifi_com.cpp*, and are discussed below.

Table 11.13 shows that the string *htmlCode* was removed from *wifi_com.cpp*. In Table 11.14, the implementation of the FSM states that were modified are shown. In WIFI_STATE_WAIT_CIPSTATUS_OK, the function *wifiComWebPageDataUpdate()* is now used to prepare the web page, as discussed below.

***Table 11.13 Sections in which lines were removed from wifi_com.cpp.***

| Section | Lines that were removed |
|---|---|
| Declaration and initialization of private global variables | `static const char htmlCode [] =`<br>`"<!doctype html> <html> <body> Hello! </body> </html>"` |

***Table 11.14 States of the FSM that were modified in wifi_com.cpp.***

| Previous implementation of the state | New implementation of the state |
|---|---|
| `case WIFI_STATE_WAIT_CIPSTATUS_OK:`<br>`  if (isExpectedResponse()) {`<br>`   wifiComState = WIFI_STATE_SEND_CIPSEND;`<br>`   }`<br><br>`  if (nonBlockingDelayRead(&wifiComDelay)) {`<br>`   nonBlockingDelayWrite(&wifiComDelay,`<br>`              DELAY_5_SECONDS);`<br>`   wifiComState = WIFI_STATE_SEND_CIPSTATUS;`<br>`  }`<br>`  break;` | `case WIFI_STATE_WAIT_CIPSTATUS_OK:`<br>`  if (isExpectedResponse()) {`<br>`   wifiComState = WIFI_STATE_SEND_CIPSEND;`<br>`   wifiComWebPageDataUpdate();`<br>`   }`<br>`  if (nonBlockingDelayRead(&wifiComDelay)) {`<br>`   nonBlockingDelayWrite(&wifiComDelay,`<br>`                         DELAY_5_SECONDS);`<br>`   wifiComState = WIFI_STATE_SEND_`<br>`   CIPSTATUS;`<br>`  }`<br>`  break;` |
| `case WIFI_STATE_SEND_CIPSEND:`<br>`  lengthOfHtmlCode =`<br>`                  (strlen(htmlCode));`<br><br><br>`  sprintf( strToSend,`<br>`           "AT+CIPSEND=%c,%d\r\n",`<br>`           currentConnectionId,`<br>`           lengthOfHtmlCode );`<br>`  wifiComStringWrite( strToSend );`<br>`  wifiComState = WIFI_STATE_WAIT_CIPSEND;`<br>`  wifiComExpectedResponse = responseOk;`<br>`  break;` | `case WIFI_STATE_SEND_CIPSEND:`<br>`  lengthOfHtmlCode =`<br>`           (strlen(htmlCodeHeader) +`<br>`            strlen(htmlCodeBody) +`<br>`            strlen(htmlCodeFooter) );`<br>`   sprintf( strToSend,`<br>`           "AT+CIPSEND=%c,%d\r\n",`<br>`           currentConnectionId,`<br>`           lengthOfHtmlCode );`<br>`  wifiComStringWrite( strToSend );`<br>`  wifiComState = WIFI_STATE_WAIT_CIPSEND;`<br>`  wifiComExpectedResponse = responseOk;`<br>`  break;` |
| `case WIFI_STATE_SEND_HTML:`<br>`  wifiComStringWrite( htmlCode );`<br><br><br>`  wifiComState = WIFI_STATE_WAIT_HTML;`<br>`  wifiComExpectedResponse = responseSendOk;`<br>`  break;` | `case WIFI_STATE_SEND_HTML:`<br>`  wifiComStringWrite( htmlCodeHeader );`<br>`  wifiComStringWrite( htmlCodeBody );`<br>`  wifiComStringWrite( htmlCodeFooter );`<br>`  wifiComState = WIFI_STATE_WAIT_HTML;`<br>`  wifiComExpectedResponse = responseSendOk;`<br>`  break;` |

In WIFI_STATE_SEND_CIPSEND, the length of the HTML code to send is obtained by summing the length in bytes of *htmlCodeHeader*, *htmlCodeBody*, and *htmlCodeFooter*.

In WIFI_STATE_SEND_HTML, it can be seen that the HTML code that is sent is composed of *htmlCodeHeader*, followed by *htmlCodeBody*, and finally *htmlCodeFooter*.

In Code 11.21, it can be seen how *htmlCodeBody* is structured. One after the other, the different values to be shown in the web page are appended onto *htmlCodeBody*. Note that *sprintf* is used to append the values, by means of an offset given by *+ strlen(htmlCodeBody)*. In order to separate the information into different lines, BEGIN_USER_LINE and END_USER_LINE are used when appending the string corresponding to each element. Note that in order to print the degrees symbol "º", the HTML predefined character entity "º" is used in line 3. Lastly, note that to append the information corresponding to different elements as an "ON" or "OFF" string, the function *stateToString()* is used.

```
void wifiComWebPageDataUpdate()
{
    sprintf( htmlCodeBody, "%s Temperature: %.2f &ordm;C %s",
             BEGIN_USER_LINE, temperatureSensorReadCelsius(), END_USER_LINE );

    sprintf( htmlCodeBody + strlen(htmlCodeBody),
             "%s Over temperature detected: %s %s", BEGIN_USER_LINE,
             stateToString( overTemperatureDetectorStateRead() ), END_USER_LINE );

    sprintf( htmlCodeBody + strlen(htmlCodeBody), "%s Gas detected: %s %s",
             BEGIN_USER_LINE, stateToString( gasDetectorStateRead() ),
             END_USER_LINE );

    sprintf( htmlCodeBody + strlen(htmlCodeBody),
             "%s Motion detected: %s %s", BEGIN_USER_LINE,
             stateToString( motionSensorRead() ), END_USER_LINE );

    sprintf( htmlCodeBody + strlen(htmlCodeBody), "%s Alarm: %s %s",
             BEGIN_USER_LINE, stateToString( sirenStateRead() ), END_USER_LINE );

    sprintf( htmlCodeBody + strlen(htmlCodeBody),
             "%s Incorrect code LED: %s %s", BEGIN_USER_LINE,
             stateToString( incorrectCodeStateRead() ), END_USER_LINE );

    sprintf( htmlCodeBody + strlen(htmlCodeBody),
             "%s System blocked LED: %s %s", BEGIN_USER_LINE,
             stateToString( systemBlockedStateRead() ), END_USER_LINE );
}
```

***Code 11.21 Implementation of wifiComWebPageDataUpdate().***

**NOTE:** Recall Chapter 3, where it was explained that the file *mbed_app.json* was introduced in order to enable the *%.2f* format that is used in Code 11.21. For more information, please refer to [6].

Code 11.22 shows the implementation of the function *stateToString()*. It returns "ON" or "OFF" depending on the value of its only parameter (*state*).

```
char * stateToString( bool state )
{
    if ( state ) {
        strcpy( stateString, "ON");
    } else {
        strcpy( stateString, "OFF");
    }
    return stateString;
}
```

***Code 11.22 Implementation of stateToString().***

In this way, the HTML code served by the smart home system looks like the example shown in Code 11.23.

```
<!doctype html>
<html>
  <head>
    <title>Smart Home System</title>
  </head>
  <body style="text-align: center;">
    <h1 style="color: #0000ff;">Smart Home System</h1>
    <div style="font-weight: bold">
      <p>Temperature: 10 &ordm;C</p>
      <p>Over temperature detected: OFF</p>
      <p>Gas detected: OFF</p>
      <p>Motion detected: OFF</p>
      <p>Alarm: OFF</p>
      <p>Incorrect code LED: OFF</p>
      <p>System blocked LED: OFF</p>
    </div>
  </body>
</html>
```

***Code 11.23 Example of the HTML code served by the smart home system.***

## Proposed Exercises

1. What should be modified in order to include more information in the web page served by the smart home system?

2. How can the HTML code be modified in order to auto refresh the data every ten seconds?

## Answers to the Exercises

1. The function *wifiComWebPageDataUpdate()* should be modified in order to include more information in *htmlCodeBody*.

2. Table 11.15 shows how to modify *htmlCodeHeader* in *wifi_com.cpp*. The meta tag should be included with the parameters \"refresh\" and content=\"10\". In this way, the web browser will automatically ask the smart home system for the web page every ten seconds.

***Table 11.15 Sections in which lines were modified in wifi_com.cpp.***

| Section | Lines that were added |
|---|---|
| Declaration and initialization of private global variables | `static const char htmlCodeHeader [] =`<br>`"<!doctype html>"`<br>`"<html> <head> <title>Smart Home System</title>`<br>`"<meta http-equiv=\"refresh\" content=\"10\" /> </head>"`<br>`"<body> <h1 style=\"text-align: center;\">"`<br>`"<font color=\"#0000ff\">Smart Home System</font></h1>"` |

# 11.3 Under the Hood

## 11.3.1 Basic Principles of Wi-Fi and TCP Connections

In this chapter, a Wi-Fi connection was used to serve a web page. In fact, Wi-Fi is a family of wireless network protocols based on the IEEE 802.11 family of standards, which are commonly used for local area networking of devices and sharing internet access. Different versions of Wi-Fi are specified that use different radio bands and technologies, which determine their maximum ranges and achievable speeds.

The ESP-01 module used in this chapter uses the same 2.4 GHz band as the HM-10 module that was introduced in Chapter 3. The 2.4 GHz band is currently the most popular band for Wi-Fi connections, together with the 5 GHz band. Each Wi-Fi band is divided into multiple channels in the same way as in Bluetooth communication, as was explained in Chapter 10. Channels can be shared between networks, but only one transmitter can transmit on a channel at any given moment in time.

The set of channels and techniques used to avoid narrowband interference problems varies depending on the Wi-Fi version, as does the maximum bit rate that is reachable under optimal conditions. Wi-Fi equipment frequently supports multiple versions of Wi-Fi. For example, the ESP-01 module supports Wi-Fi 1 (802.11b), Wi-Fi 3 (802.11g), and Wi-Fi 4 (802.11n), as described in [1]. The main characteristics of each Wi-Fi version are listed in Table 11.16.

***Table 11.16 Summary of the main characteristics of the Wi-Fi versions.***

| Generation (Standard) | Maximum Link Rate | Adopted | Frequency |
|---|---|---|---|
| Wi-Fi 6E (802.11ax) | 600 to 9608 Mbit/s | 2019 | 6 GHz |
| Wi-Fi 6 (802.11ax) | 600 to 9608 Mbit/s | 2019 | 2.4/5 GHz |
| Wi-Fi 5 (802.11ac) | 433 to 6933 Mbit/s | 2014 | 5 GHz |
| Wi-Fi 4 (802.11n) | 72 to 600 Mbit/s | 2009 | 2.4/5 GHz |
| Wi-Fi 3 (802.11g) | 3 to 54 Mbit/s | 2003 | 2.4 GHz |
| Wi-Fi 2 (802.11a) | 1.5 to 54 Mbit/s | 1999 | 5 GHz |
| Wi-Fi 1 (802.11b) | 1 to 11 Mbit/s | 1999 | 2.4 GHz |

**NOTE:** Wi-Fi is a trademark of the non-profit Wi-Fi Alliance, integrated by hundreds of companies around the world. For more information about Wi-Fi technology and the Wi-Fi Alliance, please refer to [7].

In the examples in this chapter, a TCP server was used to implement the communications between the ESP-01 module and the web browser. As mentioned earlier, TCP stands for Transmission Control Protocol and is one of the main communications protocols used on the internet and similar computer networks.

TCP provides reliable, ordered, and error-checked delivery of data between applications running on devices that communicate using a network, where every device has a unique IP (Internet Protocol) identifier. Thus, the entire suite is commonly referred to as TCP/IP. Major internet applications such as the World Wide Web, email, and file transfer all rely on TCP.

TCP is connection-oriented, and a connection between client and server has to be established before data can be sent, as was shown in subsection 11.2.2. The server must be listening for connection requests from clients before a connection is established.

TCP includes different techniques in order to improve the reliability of the communication, such as error-detection, retransmission, etc. However, it has some vulnerabilities that can be exploited by hackers. For this reason, among others, TCP has been used in the first two versions of the Hypertext Transfer Protocol (HTTP in 1996 and HTTP/2 in 2015) but is not used by the latest standard (HTTP/3 (2020)).

### Proposed Exercise

1. Which technology allows a higher data transfer rate, Bluetooth Low Energy (BLE) or Wi-Fi?

### Answer to the Exercise

1. In Chapter 10, it was shown that depending on the BLE version, the maximum achievable bit rate is between 1 Mbit/s and 2 Mbit/s. Looking at Table 11.16, it can be seen that Wi-Fi allows a higher data transfer rate.

## 11.4 Case Study

### 11.4.1 Indoor Environment Monitoring

In this chapter, a web server was incorporated into the smart home system. In this way, the user is able to access the information of the smart home system by means of a web browser. In [8], an Mbed-based indoor environment monitoring system is shown that allows facilities managers to measure environmental factors such as humidity, light levels, $CO_2$, and occupancy, by means of a web service that offers real-time insights, alerts, and reports relating to building performance. A representation of the system is shown in Figure 11.21.

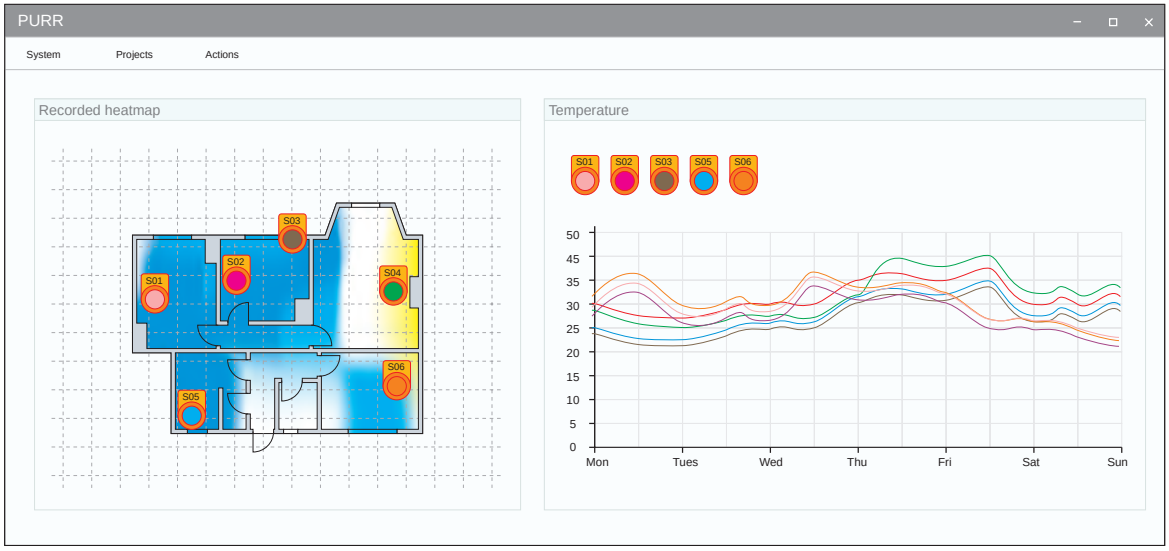



*Figure 11.21 Example of an Mbed-based system having a web service with real-time insights, alerts and reports.*

There are many similarities between the functionality of the indoor environment monitoring system and the functionality of the smart home system, such as:

- Gas detection
- Alert generation
- Occupancy monitoring
- Light level measurement
- Real-time information

## Proposed Exercise

1. How can a plan of the home be added to the web page provided by the smart home system?

## Answer to the Exercise

1. The code of the web page must be modified in order to allow the user to upload a home plan.

## References

[1] "ESP-01/07/12 Series Modules User's Manual". Accessed July 9, 2021. https://docs.ai-thinker.com/_media/esp8266/esp8266_series_modules_user_manual_en.pdf

[2] "esp8266-module-family [ESP8266 Support WIKI]". Accessed July 9, 2021. https://www.esp8266.com/wiki/doku.php?id=esp8266-module-family

[3] "GitHub - armBookCodeExamples/Directory". Accessed July 9, 2021. https://github.com/armBookCodeExamples/Directory

[4] "AT Command Set — ESP-AT User Guide documentation". Accessed July 9, 2021. https://docs.espressif.com/projects/esp-at/en/latest/AT_Command_Set/index.html

[5] "TCP_IP AT Commands — ESP-AT User Guide documentation". Accessed July 9, 2021. https://docs.espressif.com/projects/esp-at/en/latest/AT_Command_Set/TCP-IP_AT_Commands.html#cmd-status

[6] "mbed-os_README.md at master · ARMmbed_mbed-os · GitHub". Accessed July 9, 2021. https://github.com/ARMmbed/mbed-os/blob/master/platform/source/minimal-printf/README.md#usage

[7] "Wi-Fi Alliance". Accessed July 9, 2021. https://www.wi-fi.org/

[8] "Indoor Environment Monitoring | Mbed". Accessed July 9, 2021. https://os.mbed.com/built-with-mbed/indoor-environment-monitoring/

# Chapter 12

## Guide to Designing and Implementing an Embedded System Project

# 12.1 Roadmap

## 12.1.1 What You Will Learn

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

- Describe how an embedded system project can be developed following an ordered process.
- Design and implement a prototype of an embedded system, including its hardware and software.
- Summarize the fundamentals of the concepts of verification and validation.
- Develop the final documentation of an embedded system.

## 12.1.2 Review of Previous Chapters

Throughout this book, a smart home system provided with a broad variety of functionalities has been implemented. The NUCLEO board was used as the system core, and many hardware modules and elements were connected to it. A learn-by-doing approach was used, by means of which several embedded system programming concepts were introduced.

The smart home system project was started from scratch in Chapter 1, and functionality was gradually added through the chapters as different hardware modules and elements were incorporated. This approach led to a single file having hundreds of lines, after which the idea of software modularization was introduced in Chapter 5. From then on, in Chapters 6 to 11, many software modules were included as more hardware was incorporated into the system.

The reader may have noticed that, for pedagogical reasons, the features and functionalities of the smart home system were not established at the beginning. This made it possible to introduce the topics gradually, but also led to many changes during its implementation. As a consequence, it can be concluded that it would be more convenient to adopt a structured process to efficiently design and implement an embedded system. This process should include a step at the beginning where the features and functionality of the system are defined.

## 12.1.3 Contents of This Chapter

In this chapter, a structured process will be introduced to efficiently design and implement an embedded system. The proposed process consists of ten steps, including the selection of the project, its definition, design, implementation, and final documentation. The hardware and software aspects are tackled, following an approach that guarantees consistency between the initial objectives and the obtained results.

In order to illustrate how each of the proposed steps is carried out, a project is implemented within this chapter. For the sake of brevity, a system with a reduced number of sensors and actuators is used in the examples, although this does not limit the introduction of the important concepts.

The proposed process benefits from all the concepts that were introduced throughout this book, while also helping to introduce other important concepts such as *requirements*, *verification*, and *validation*. This chapter constitutes a summary of the book, while introducing important new concepts.

## 12.2 Fundamentals of Embedded System Design and Implementation

### 12.2.1 Proposed Steps to Design and Implement an Embedded System Project

The proposed process to design and implement embedded systems is summarized in Table 12.1. It can be seen that it consists of ten steps, ranging from the selection of the project that will be implemented to its design, implementation, and final documentation, as was mentioned in the previous section.

***Table 12.1 Summary of the proposed steps to design and implement an embedded system project.***

| Step | Outcome |
|---|---|
| 1. Select the project that will be implemented | A rationale that leads to an appropriate project to implement |
| 2. Elicit project requirements and use cases | A concise and structured description of what will be implemented |
| 3. Design the hardware | Diagram of hardware modules, connections, and bill of materials |
| 4. Design the software | Diagram of the software design and description of the modules |
| 5. Implement the user interface | Software implementation |
| 6. Implement the reading of the sensors | Software implementation |
| 7. Implement the driving of the actuators | Software implementation |
| 8. Implement the system behavior | Software implementation |
| 9. Check the system behavior | Assessment of accomplishment of requirements and use cases |
| 10. Develop the final documentation | A reference to the most relevant documentation of the project |

The proposed steps include a gradual implementation of the software. First, the user interface is implemented in step 5. The aim is to have a way to read the system information and to enter commands. In this step, the reading of the sensors is replaced by *stub* code that temporarily substitutes the *yet-to-be-developed* code to read the sensors. In step 6, the reading of the sensors is implemented and the corresponding values shown on the user interface. In step 7, the actuators are driven, but stub code is used to trigger their activation. In step 8, the complete system behavior is implemented, so no more stub code remains in the software.

In the examples below, the ten proposed steps are introduced by means of a given project that is developed from start to end, following a *top-down* approach. This starts by formulating an overview design of the system, and goes on to gradually define and implement each part in detail.

**TIP:** Through the examples, the reader is encouraged to think of their own project and to develop that project as each step is introduced, by adapting each example to their own requirements. It is recommended that the reader choose a simple project to avoid complications that will distract from the main aim of this chapter, which is to understand and adopt the proposed steps.

**NOTE:** There are many other possible approaches to tackling the design and implementation of an embedded system, depending on the characteristics of the project, as well as on the size and skills of the developing team. The proposed steps correspond to the implementation of an embedded system prototype by a single developer or a very reduced team. A more complex project may suggest an iterative approach, where the outcomes of the steps are revised and improved more than once.

**Example 12.1: Select the Project that will be Implemented**

### Objective

Introduce the idea that the project to be implemented should result from a decision process.

### Summary of the Expected Outcome

As a result of this step, a rationale about the most appropriate project to be implemented should be obtained.

**NOTE:** This step is particularly important because the results can lead to either a valuable project or a questionable project (in terms of learning, benefits, etc.).

### Discussion on How to Implement this Step

In order to select the project, it should first be established which aspects will be analyzed in the decision process. Those aspects will vary depending on the developer profile and skills, the organization where the developer is studying or working, and many other aspects. However, a table summarizing the aspects to be analyzed, as well as the score of each proposed project in each aspect, seems to be a reasonable way to decide which project to implement.

### Implementation of this Proposed Step

First, the aspects that will be considered in the decision process should be determined. For the sake of brevity, just a few aspects will be considered in this example. However, an important idea is introduced: different aspects may have different weights in the decision process. To factor all these ideas into the decision process, a quantitative approach will be followed.

Some aspects that could be considered in the decision process are as follows:

- Availability of the hardware
- Utility of the project
- Implementation time

In the particular context of this book, the hardware availability might be considered an important aspect, given that it is convenient for the reader to reuse the hardware from previous chapters. The usefulness of the project could be considered less relevant, because in this context the reader is more concerned about learning than using the project in a real-life application. Finally, the implementation time can be considered an important aspect, given that the aim is to choose a simple project that can be completed in a single chapter.

Some possible projects that could be analyzed include the following:

- A mobile robot that avoids obstacles
- A flying drone fitted with a camera
- A home irrigation system for indoor plants

In this way, Table 12.2 can be obtained, which includes the possible projects and all the proposed aspects with given weights, as per the above discussion. A weighted score for each project is obtained.

The mobile robot that avoids obstacles will require hardware that was not used in this book (wheels, structure, motors, obstacle sensors, etc.), and for that reason was awarded a score of three out of ten points in the hardware availability aspect. The utility of this project is questionable, and for that reason this project again obtains three out of ten points in the corresponding aspect. Finally, this project will take a reasonable implementation time and, therefore, obtains five points in that aspect. Consequently, considering that the aspects are weighted by a factor of ten, five, and eight, respectively, the weighted scores are obtained (30, 15, and 40), and the sum of weighted scores is 85, as can be seen in the last column of Table 12.2.

The flying drone fitted with a camera will require even more specific and complex hardware than the mobile robot, and for that reason it gets two points in the hardware availability aspect. The utility of this project might be considered higher than the utility of the mobile robot, and therefore it gets five points in that aspect. Finally, this project will demand a considerable implementation time and, therefore, this project scores two points in that aspect. As a result, this project gets a sum of weighted scores of 61.

**NOTE:** The implementation time aspect gets a lower score when the time demand is bigger.

A typical home irrigation system allows the user to set how often and for how long the plants are irrigated. It can be implemented by means of a few buttons, an LCD display, a moisture sensor, and an on/off electro-valve, as will be discussed in Example 12.3. Most of this hardware is already available to the reader or is easy to obtain and use. Therefore, this project gets eight points in the hardware availability aspect. This project can be useful if the reader has some indoor plants, and therefore it gets seven points in the utility of the project aspect. Finally, this project can be implemented with limited effort, and, therefore, it gets eight points in the implementation time aspect. As a result, this project gets a sum of weighted scores of 179.

The home irrigation system for indoor plants project is chosen, as it gets the highest value in the sum of weighted scores, as can be seen in the last column of Table 12.2.

***Table 12.2 Selection of the project to be implemented.***

| Project | | Hardware availability (weight: 10) | Utility of the project (weight: 5) | Implementation time (weight: 8) | Sum of weighted scores |
|---|---|---|---|---|---|
| Mobile robot that avoids obstacles | Score on each aspect: | 3 | 3 | 5 | – |
| | Weighted score: | 30 | 15 | 40 | 85 |
| Flying drone provided with a camera | Score on each aspect: | 2 | 5 | 2 | – |
| | Weighted score: | 20 | 25 | 16 | 61 |
| Home irrigation system for indoor plants | Score on each aspect: | 8 | 7 | 8 | – |
| | Weighted score: | 80 | 35 | 64 | 179 |

**NOTE:** The score in each aspect, as well as the weight of each aspect, is an approximate value only. It should be noted that, in general, a small variation in an aspect score for a given project, or in an aspect weighting, does not modify which project obtains the highest sum of weighted scores shown in Table 12.2.

## Proposed Exercise

1. How can the reader use the concepts that were introduced in this chapter to select a project to implement?

## Answer to the Exercise

1. The reader should first establish a list of aspects that they would like to consider in the selection of a project. Then, the weighting of each aspect should be determined, and finally the different projects should be scored.

**TIP:** Consider including aspects such as "How fun the project is," "What will be learned," or "Profitability" as a way to guarantee that the selection reflects the reader's personal motivations.

### Example 12.2: Elicit Project Requirements and Use Cases

#### Objective

Introduce the concepts of *requirements* and *use cases*.

#### Summary of the Expected Outcome

As a result of this step, the project to be implemented should be clearly defined. This will be done by means of a list of requirements and use cases.

#### Discussion of How to Implement this Step

This step is based on the concepts of requirements and use cases.

**DEFINITION:** In product development, a *requirement* is a singular documented physical, functional, or non-functional need that a particular design, product, or process aims to satisfy.

**DEFINITION:** In software and systems engineering, a *use case* is a list of actions or event steps typically defining the interactions between an actor (for example, a user) and a system to achieve a goal.

The requirements are the basis on which a project is to be developed. Therefore, there are many important criteria that a developer should apply when writing the requirements. Some of these criteria are frequently summarized using the "SMART" mnemonic acronym, as shown in Table 12.3.

***Table 12.3 Summary of the SMART mnemonic acronym.***

| Letter | Term adopted in this book | Meaning |
|---|---|---|
| S | Specific | Clearly defined |
| M | Measurable | Able to be measured |
| A | Achievable | Able to be achieved |
| R | Relevant | Targets a significant need |
| T | Time-bound | Has a time limit or deadline |

**NOTE:** SMART is sometimes used to refer to other criteria. For example, the letter "A" is frequently related to the term "agreed," meaning that the requirements must be agreed with the client.

**WARNING:** In professional *project management*, every project should have a due date. Therefore, the time-bound criterion applies to the whole project and can also be applied to each requirement. In this example, it is established that the whole project (all the requirements) should be completed in one week.

A use case can be defined in multiple ways. In the scope of this book, the elements shown in Table 12.4 will be used. The reader should note that alongside each is a simplified approach that is appropriate for many projects.

***Table 12.4 Elements that will be used to define a use case.***

| Use case element | Meaning |
|---|---|
| ID | A unique number that represents a use case |
| Title | The title that is associated with the use case |
| Trigger | What event triggers this use case |
| Precondition | What must be met before this use case can start |
| Basic flow | The events that occur when there are no errors or exceptions |
| Alternate flows | The most significant alternatives and exceptions |

Lastly, in most cases there are already products on the market that implement the same functionality as the embedded system project that will be designed. Many of those products may be very successful. Therefore, it is recommended to analyze and summarize those products before writing the requirements and use cases of a new embedded system project, as per the example below.

## Implementation of this Proposed Step

First, some examples of home irrigation systems are analyzed, as was suggested in the previous paragraph. Table 12.5 summarizes the main characteristics of two products. Based on those characteristics, the requirements indicated in Table 12.6 are established with consideration of what can be achieved.

**NOTE:** The requirements in Table 12.6 are preliminary requirements that may be modified as the project evolves. If requirements are modified, then the modifications should be clearly indicated in order to avoid misunderstandings.

***Table 12.5 Summary of the main characteristics of two home irrigation systems currently available on the market.***

| Characteristics | Product A | Product B |
|---|---|---|
| Water-in port | ½-inch connector | ½-inch connector |
| Irrigation circuits | Controls two independent irrigation circuits | Controls one irrigation circuit |
| Operation modes | Continuous: water flow is controlled by a button<br>Programmed irrigation: irrigation is time-controlled | Continuous: water flow is controlled by a button<br>Programmed irrigation: irrigation is time-controlled |
| Configuration | The parameters "how often" to irrigate and "how long" to irrigate can be configured for each circuit | The parameters "how often" to irrigate and "how long" to irrigate can be configured |
| User interface | A rotary control key, two buttons, and a display | A set of buttons and a set of LEDs |
| Power supply | Two AA batteries | Four AA batteries |
| Sensors | None | None |
| Price | 80 USD | 60 USD |

*Table 12.6 Initial requirements defined for the home irrigation system.*

| Req. Group | Req. ID | Description |
|---|---|---|
| 1. Water | 1.1 | The system will have one water-in port based on a ½-inch connector |
| | 1.2 | The system will control one irrigation circuit by means of a solenoid valve |
| 2. Modes | 2.1 | The system will have a continuous mode in which a button will enable the water flow |
| | 2.2 | The system will have a programmed irrigation mode based on a set of configurations: |
| | 2.2.1 | Irrigation will be enabled only if moisture is below the "Minimum moisture level" value |
| | 2.2.2 | Irrigation will be enabled every H hours with H being the "How often" configuration |
| | 2.2.3 | Irrigation will be enabled for S seconds, with S being the "How long" configuration |
| | 2.2.4 | Irrigation will be skipped if "How long" is configured to 0 (zero) |
| 3. Configuration | 3.1 | The system configuration will be done by means of a set of buttons: |
| | 3.1.1 | The "Mode" button will change between "Programmed irrigation" and "Continuous irrigation" |
| | 3.1.2 | The "How often" button will increase the time between irrigations in programmed mode by one hour |
| | 3.1.3 | The "How long" button will increase the irrigation time in programmed mode by ten seconds |
| | 3.1.4 | The "Moisture" button will increase the "Minimum moisture level" configuration by 5% |
| | 3.1.5 | The maximum values will be: "How often": 24 h; "How long": 90 s; "Moisture": 95% |
| | 3.1.6 | "How long" and "Moisture" will be set to 0 (zero) if the maximum is reached and the button is pressed |
| | 3.1.7 | "How often" will be set to 1 if the maximum is reached and the button is pressed |
| 4. Display | 4.1 | The system will have an LCD display: |
| | 4.1.1 | The LCD display will show the current operation mode: Continuous or Programmed |
| | 4.1.2 | The LCD display will show the values of "How often", "How long", and "Minimum moisture level" |
| 5. Sensor | 5.1 | The system will measure soil moisture at one point with an accuracy better than 5% |
| 6. Power supply | 6.1 | The system will be powered using two AA batteries |
| 7. Due date | 7.1 | The system will be finished one week after starting (this includes buying the parts) |
| 8. Cost | 8.1 | The components for the prototype should cost less than 60 USD |
| 9. Documents | 9.1 | The prototype should be accompanied by a list of parts, a connection diagram, the code repository, and a table indicating the accomplishment of requirements and use cases |

Finally, three use cases are defined, as can be seen in Table 12.7, Table 12.8, and Table 12.9.

*Table 12.7 Use Case #1 – Title: The user wants to irrigate plants immediately for a couple of minutes.*

| Use case element | Definition |
|---|---|
| Trigger | The user realizes that the plants need irrigation immediately |
| Precondition | The system must be powered on, and the water supply should be connected.<br>The system is not irrigating the plants. |
| Basic flow | The user presses the "Mode" button to set "Continuous" irrigation. Water starts to flow to the plants. After a couple of minutes, the user presses the "Mode" button to set "Programmed" irrigation. Water irrigation stops. |
| Alternate flows | 1.a. There is no water supply. The user presses the "Mode" button to set "Continuous" irrigation. The solenoid valve is activated, but the plants are not irrigated.<br>1.b. The user presses the "Mode" button. The user notes that the water is overflowing the plant pot. The user presses the "Mode" button and irrigation stops. |

*Table 12.8 Use Case #2 – Title: The user wants to program irrigation to take place for ten seconds every six hours.*

| Use case element | Definition |
|---|---|
| Trigger | The user wants to establish an irrigation program |
| Precondition | The system must be powered on, and the water supply should be connected.<br>The system is not irrigating the plants. |
| Basic flow | The user presses the "How often" button until the value "6 hours" is shown on the display. The user presses the "How long" button until the value "10 seconds" is shown on the display. Irrigation will start in six hours and will last for ten seconds if the measured moisture is below the "Minimum moisture level" configured. |
| Alternate flows | 2.a. There is no water supply. The solenoid valve will be activated, but the plants will not be irrigated. |

*Table 12.9 Use Case #3 – Title: The user wants the plants not to be irrigated.*

| Use case element | Definition |
|---|---|
| Trigger | The user wants the plants to not be irrigated |
| Precondition | The system must be powered on, and water supplied should be connected.<br>The system is not irrigating the plants. |
| Basic flow | The user presses the "How long" button until "How long" is set to 0 (zero).<br>Irrigation is skipped, and a legend indicating "Programmed-Skip" is shown. |
| Alternate flows | 3.a. Plants will not be irrigated even if the measured moisture is below the "Minimum moisture level" configured. |

### Proposed Exercise

1. Define the requirements and three use cases for the project that was selected in the "Proposed Exercise" of Example 12.1.

### Answer to the Exercise

1. It is strongly recommended to start by analyzing and summarizing some products that are available on the market. Based on the characteristics of those products and the reader's own ideas, the initial requirements may be established following the format shown in Table 12.6. The use cases can be defined following the format shown in Table 12.7.

**TIP:** When defining the requirements, keep in mind the SMART criteria discussed above. In particular, consider the time available to implement the project and the relevancy and achievability of each requirement. Moreover, try to clearly define each requirement, and use measurable quantities if possible.

## Example 12.3: Design the Hardware

### Objective

Design hardware that fulfills the requirements and can be used to implement the software functionality.

### Summary of the Expected Outcome

The outcomes of this step are expected to include:

- a diagram of the hardware modules showing all of their interconnections,
- a rationale that argues the most appropriate components with which to implement the hardware modules,
- a connection diagram of the selected components and tables summarizing their interconnections, and
- a bill of materials that includes all the necessary elements to implement the prototype.

### Discussion of How to Implement this Step

A reasonable approach for this step might be to start by analyzing designs made by other developers to implement similar requirements; this also includes the previous chapters of this book. A diagram following the ideas shown in Chapter 1 will be the starting point. After this diagram is created, the parts can be selected and their interconnections defined. The final step will be to create the bill of materials.

### Implementation of this Proposed Step

Figure 12.1 shows a first proposal for the hardware modules of the irrigation system prototype. It is based on the elements introduced in previous chapters, as well as the assumption that it will be possible to find an appropriate moisture sensor and a suitable solenoid valve, which converts electrical energy into mechanical energy and, in turn, opens or closes the valve mechanically.

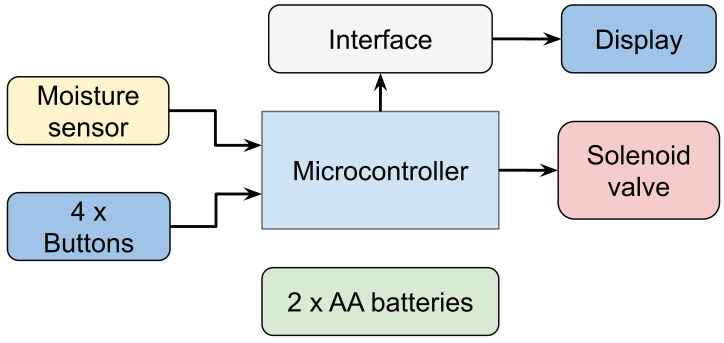



*Figure 12.1 First proposal of the hardware modules of the irrigation system.*

In the prototype implementation, it is reasonable to use the NUCLEO board for the microcontroller module. The reader already owns this board and knows that it is capable of implementing all the software functionality.

For the display, any of the options introduced in Chapter 6 can be used. For the sake of simplicity, and considering the hardware costs, the character-based LCD display is selected using 4-bit mode.

The buttons will be implemented using a breadboard and tactile switches, as in Chapter 1. Therefore, all that is needed is to define how to implement the moisture sensor, the solenoid valve, and the battery power supply.

In Table 12.10, three moisture sensor modules are shown. After this comparison, it seems reasonable to use the HL-69 in this first prototype due to its price. It also seems very difficult to accomplish Requirement 5.1, related to having an accuracy of better than 5%, as accuracy is not specified for any of the sensors.

***Table 12.10 Comparison of moisture sensors.***

| Sensor name | Technology | Accuracy | Interface | Unit price [USD] |
|---|---|---|---|---|
| HL-69 | Resistive | Not specified | VCC, GND, Digital Output, Analog Output | 2 |
| SEN-13322 | Resistive | Not specified | VCC, GND, Analog Output | 5 |
| Moisture v1.2 | Capacitive | Not specified | VCC, GND, Analog Output | 2 |

In Table 12.11, three solenoid valves are shown. After this comparison, it seems reasonable to use an FPD-270A in the first prototype. Given that the solenoid valve must be powered using 12 V, a relay module will need to be used to control its activation. In order to keep this first design simple, a 12 V power supply can be included in the system. Consequently, in this prototype, the NUCLEO board can be supplied using the USB connection, as it has throughout this book.

**NOTE:** In a future version of the system, it might be considered to use a switching step-up power supply connected to two AA batteries to provide 12 V for the solenoid and 5 V for the NUCLEO board.

***Table 12.11 Comparison of solenoid valves.***

| Solenoid name | Water-in connector | Activation method | Unit price [USD] |
|---|---|---|---|
| FPD-270A | ½-inch | 12 V / 0.25 A | 9 |
| USS-NSV00003 | ½-inch | 12 V / 1 A | 35 |
| VA-8H | ½-inch | 12 V / 0.5 A | 45 |

The resulting final version of the hardware modules of the irrigation system is shown in Figure 12.2.

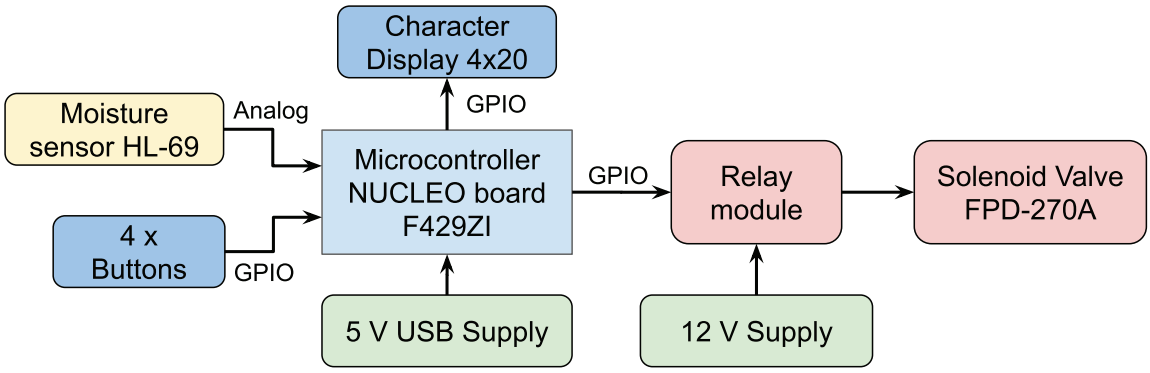



*Figure 12.2 Final version of the hardware modules of the irrigation system.*

**NOTE:** In the hardware design shown in Figure 12.2, an MB102 module is not used because the 300 mA maximum current consumption from the 5 V USB supply discussed in Chapter 4 is not reached.

The proposed connection diagram of all the hardware elements is shown in Figure 12.3. From Table 12.12 to Table 12.17, the corresponding connections are summarized. For the pin assignments, the diagram available in [1] has been used, while the connections used in previous chapters were kept whenever possible.

*Table 12.12 Summary of the connections between the NUCLEO board and the character-based LCD display.*

| NUCLEO board | Character LCD display |
|---|---|
| D4 | D4 |
| D5 | D5 |
| D6 | D6 |
| D7 | D7 |
| D8 | RS |
| D9 | E |

*Table 12.13 Summary of other connections that should be made to the character-based LCD display.*

| Character LCD display | Voltage/Element |
|---|---|
| VSS | GND |
| VDD | 5 V |
| VO | 10 kΩ potentiometer |
| R/W | GND |
| A | 1 kΩ resistor to 5 V |
| K | GND |

*Table 12.14 Summary of the connections between the NUCLEO board and the buttons.*

| NUCLEO board | Button |
|---|---|
| PG_1 | "Mode" |
| PF_9 | "How Often" |
| PF_7 | "How Long" |
| PF_8 | "Moisture" |

*Table 12.15 Summary of connections that should be made to the HL-69 moisture sensor.*

| HL-69 moisture sensor | Voltage/Element |
|---|---|
| GND | GND |
| VCC | 3.3 V |
| DO | Unconnected |
| AO | A3 pin - NUCLEO board |

*Table 12.16 Summary of connections that should be made to the relay module.*

| Relay module | Voltage/Element |
|---|---|
| VCC | 5 V |
| GND | GND |
| IN1 | PF_2 |
| NO1 | FPD-270A (Terminal 1) |
| COM1 | 12 V |
| NC1 | Unconnected |
| IN2 | Unconnected |
| NO2 | Unconnected |
| COM2 | Unconnected |
| NC2 | Unconnected |

*Table 12.17 Summary of connections that should be made to the FPD-270A.*

| FPD-270A | Voltage/Element |
|---|---|
| Terminal 1 | Relay module (NO1) |
| Terminal 2 | GND12 |

**NOTE:** The GND terminal of the 12 V power supply (named *GND12*) was intentionally kept isolated from the GND of the 5 V USB power supply. This is done to prevent any potential damage to the microcontroller caused by the 12 V voltage, and also to diminish the electrical noise interference over the microcontroller that could be generated when the load is activated, as was explained in Chapter 7.

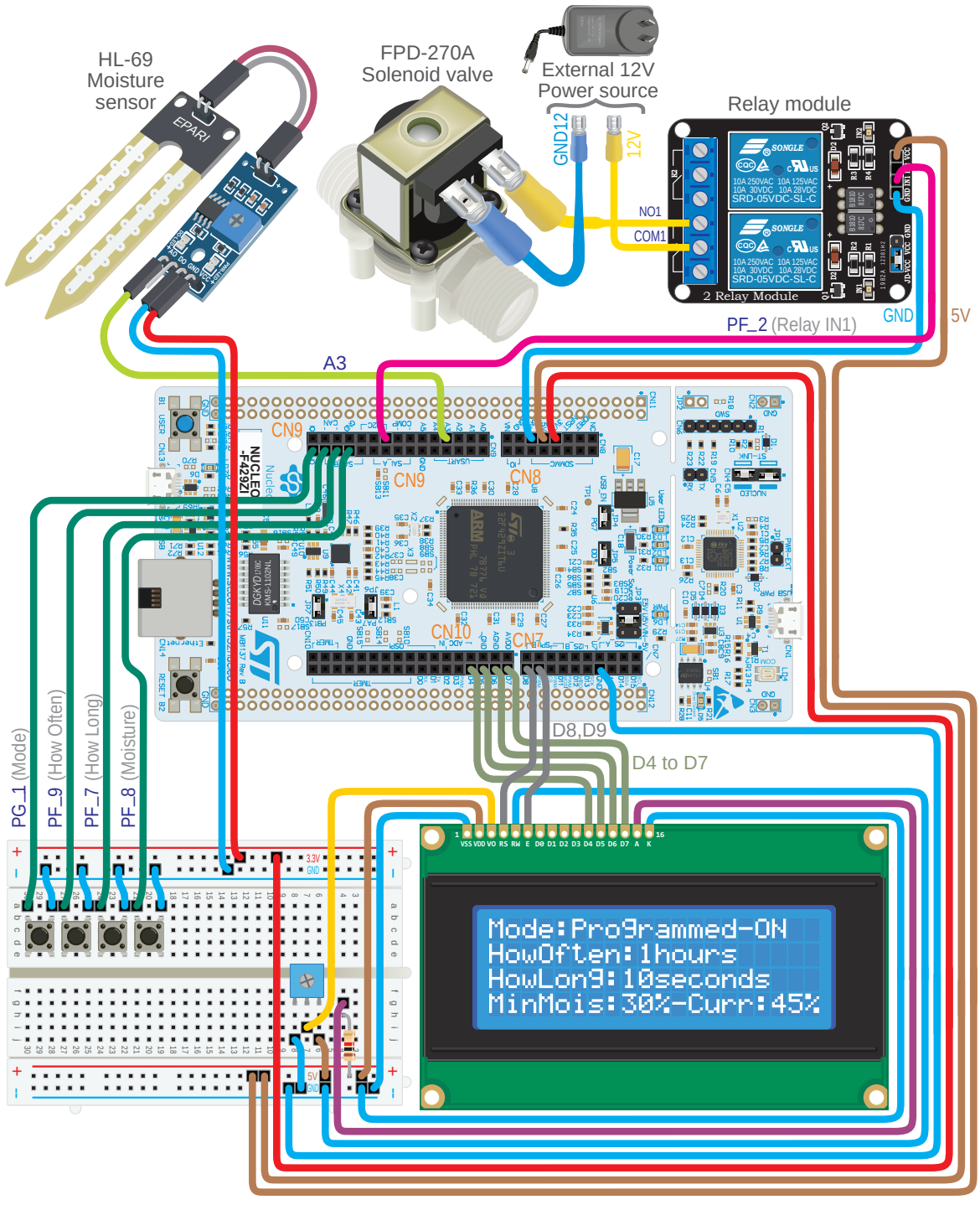



*Figure 12.3 Connection diagram of all the hardware elements of the irrigation system.*

**NOTE:** The connections used in previous chapters have been kept whenever possible. In this way, the setup shown in Figure 12.3 can be promptly implemented from the setup used in previous chapters.

**TIP:** In order to be able to use the program codes of Chapters 6 to 11 again without delay, the elements connected in those chapters can be left connected to the NUCLEO board and to the breadboard (they will not cause any interference in this chapter). In that case, disconnect the wire that connects the 3.3 V output of the NUCLEO board with the breadboard, and use the MB102 module to supply 3.3 V to the system.

Finally, in Table 12.18, the bill of materials is shown. It can be seen that the estimated cost is below 60 USD, and therefore requirement 8.1, which was established in Table 12.6, is fulfilled. Note that if this prototype were to be produced in quantity, a redesign would have to be done. This would lower some costs (e.g., a bespoke design of the microcontroller board will save cost), while it would increase other costs (e.g., container, printed circuit board, time required).

***Table 12.18 Bill of materials.***

| Item | Quantity | Price [USD] |
|---|---|---|
| NUCLEO Board F429ZI | 1 | 25 |
| Moisture sensor HL-69 | 1 | 2 |
| FPD-270A solenoid valve | 1 | 9 |
| Character display 4 × 20 | 1 | 15 |
| Relay module | 1 | 5 |
| Tactile switches | 4 | 0.1 |
| | Total: | 56.4 |

## Proposed Exercise

1. Design the hardware of the project that was selected in the "Proposed Exercise" of Example 12.1. Create a diagram of the connections and tables indicating all the details. A bill of materials will also be very useful.

## Answer to the Exercise

1. It is strongly recommended to start by reusing as much as possible from the previous chapters of this book. A search of the internet may help to find appropriate components and circuits for the remaining parts of the project.

### Example 12.4: Design the Software

#### Objective

Design software that fulfills the functionality described in the requirements and the use cases.

#### Summary of the Expected Outcome

The results of this step are expected to include:

- a diagram of the software modules indicating all their interconnections,
- a table indicating the variables and objects of each of the software modules,
- a table indicating the functions of each of the software modules that will be used, and
- a diagram of the finite-state machine (FSM) that will be used to implement the functionality.

#### Discussion of How to Implement this Step

Given the set of requirements, the software design is not necessarily unique. The design will depend on the developer's experience and preferences. A reasonable approach to this step might be to start by analyzing designs made by other developers to implement similar requirements. This also includes the previous chapters of this book. A diagram using the ideas shown in Chapter 5 can be the starting point. After this, a set of tables showing the variables, objects, and functions of each of the software modules can be prepared. Finally, a diagram of the FSM that will be used to implement the functionality can be drawn, as well as a sketch of the proposed layout for the display.

#### Implementation of this Proposed Step

Since the functionality of the system does not require any high-speed reactions, a very simple approach such as the one shown in Figure 12.4 can be followed. It can be seen that there is an initialization of all the modules and then an update every 100 milliseconds. The reader might notice that it is very similar to the approach followed in Chapter 5 of this book. However, a non-blocking delay will be used to obtain more accurate timing behavior.

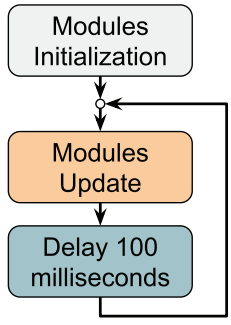



*Figure 12.4 Software design of the irrigation system program.*

The next step will be to determine the software modules. Figure 12.5 shows a proposal based on

software modules introduced in previous chapters (display, relay, etc.), as well as the assumption that it will be possible to implement a *Moisture sensor* module. An *Irrigation timer* module has been included to account for the *waiting time* between irrigations, and to account for the *irrigation time* during the irrigation. Finally, an *Irrigation control* module has been included to implement an FSM to control the irrigation.

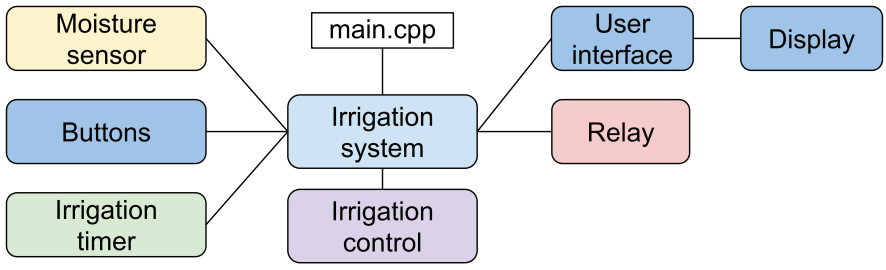



***Figure 12.5 Software modules of the irrigation system program.***

The proposed organization of folders and files to implement the software is shown in Figure 12.6. It can be appreciated that it is very similar to the organization introduced in Chapter 5.

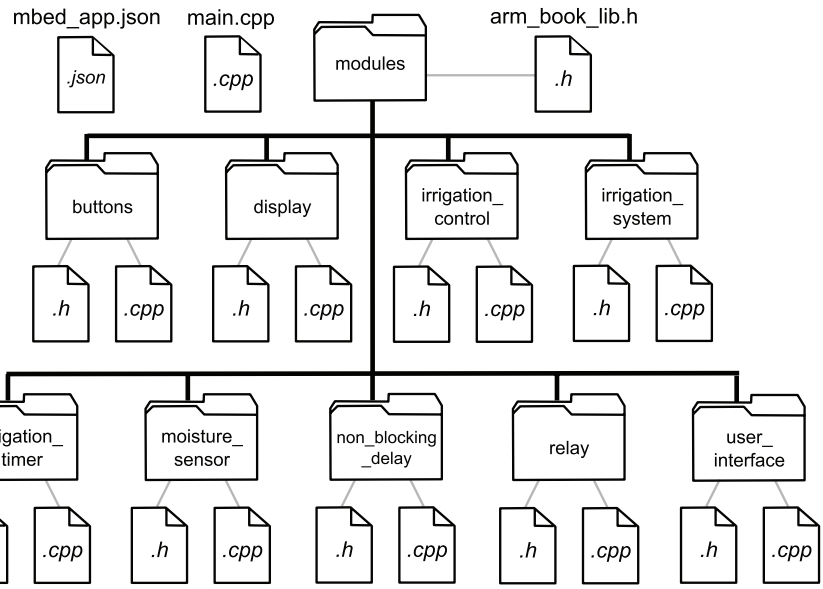



***Figure 12.6 Diagram of the .cpp and .h files of the irrigation system software.***

In Table 12.19, there is a brief description of each of the modules shown in Figure 12.5. Note that the proposed implementation is very simple, with the aim of keeping the attention on the proposed steps to implement a project. For this reason, there is only one driver that is used to manage the display, which uses the same files introduced in Chapter 6 and will be used without any modification.

*Table 12.19 Functionalities and roles of the home irrigation system modules.*

| Module | Description of its functionality | Role |
|---|---|---|
| irrigation_system | Calls initialization and update functions of other modules | System |
| moisture_sensor | Reads the HL-69 moisture sensor and processes the readings | Subsystem |
| buttons | Detects buttons pressed and accounts for the configurations | Subsystem |
| irrigation_timer | Accounts for waiting and irrigation time in programmed mode | Subsystem |
| irrigation_control | Controls irrigation based on the mode and the timer | Subsystem |
| relay | Controls relay activation | Subsystem |
| user_interface | Sends information to be printed to the display driver | Subsystem |
| display | Receives commands from the user_interface module | Driver |

**NOTE:** In the proposed implementation, the *buttons* module detects the buttons that are pressed by the user. It also processes the information to determine the current value of the configurations that were detailed in Table 12.6: "How often", "How long", and "Minimum moisture level". In addition, it detects when the Mode button is pressed and informs the *irrigation_control* module, which determines the current operation mode.

The proposed next step is to define the private variables and objects of each of the subsystem modules. These are shown in Table 12.20 to Table 12.25.

*Table 12.20 Private objects and variables of the moisture_sensor module.*

| Name of the element | Type | Description of its functionality |
|---|---|---|
| hl69 | AnalogIn object | Is used to read the A3 analog input of the NUCLEO board where the HL-69 is connected. |
| hl69AveragedValue | Float variable | Is used to process the reading to avoid noise problems. It is the average of the last ten readings. |
| hl69ReadingsArray | Float variable | Is used to store the last ten readings of the HL-69 moisture sensor. |

*Table 12.21 Private objects and variables of the buttons module.*

| Name of the element | Type | Description of its functionality |
|---|---|---|
| changeModeButton | DigitalIn object | Is used to read the PG_1 digital input of the NUCLEO board, which indicates when the current mode has to be changed. |
| howOftenButton | DigitalIn object | Is used to read the PF_9 digital input of the NUCLEO board, by means of which the "how often" configuration is changed. |
| howLongButton | DigitalIn object | Is used to read the PF_7 digital input of the NUCLEO board, by means of which the "how long" configuration is changed. |
| moistureButton | DigitalIn object | Is used to read the PF_8 digital input of the NUCLEO board, by means of which the "minimum moisture level" is changed. |
| buttonsStatus | Typedef | Is used to store the configuration status. Its members are changeMode, howOften, howLong, and moisture. |

***Table 12.22 Private objects and variables of the irrigation_timer module.***

| Name of the element | Type | Description of its functionality |
|---|---|---|
| irrigationTimer | Typedef | Used to track the status of the timers. Its members are *waitedTime* and *irrigatedTime*, both integer variables. |

***Table 12.23 Private objects and variables of the irrigation_control module.***

| Name of the element | Type | Description of its functionality |
|---|---|---|
| irrigationControlStatus | Typedef | Used to inform the state of the FSM that is implemented in this module and also used to indicate when to reset the timers of the irrigation_timer module. Its members are:<br>• *irrigationState:* Enum type defined, with valid states: INITIAL_MODE_ASSESSMENT, CONTINUOUS_MODE_IRRIGATING, PROGRAMMED_MODE_WAITING_TO_IRRIGATE, PROGRAMMED_MODE_IRRIGATION_SKIPPED, and PROGRAMMED_MODE_IRRIGATING.<br>• *waitedTimeMustBeReset:* a Boolean variable.<br>• *irrigatedTimeMustBeReset:* a Boolean variable. |

***Table 12.24 Private objects and variables of the user_interface module.***

| Name of the element | Type | Description of its functionality |
|---|---|---|
| – | – | This module has no private objects or variables. |

***Table 12.25 Private objects and variables of the relay module.***

| Name of the element | Type | Description of its functionality |
|---|---|---|
| relayControlPin | DigitalInOut object | Used to write the PF_2 pin of the NUCLEO board. When it is 0, the relay is activated. |

From Table 12.26 to Table 12.32, the proposed public functions for each of the modules are detailed.

***Table 12.26 Public functions of the irrigation_system module.***

| Name of the function | Description of its functionality | File that uses it |
|---|---|---|
| irrigationSystemInit() | Initializes the subsystems of the irrigation system and the non-blocking delay. | main.cpp |
| irrigationSystemUpdate() | Calls the functions that update the modules when the corresponding time of non-blocking delay has elapsed. | main.cpp |

***Table 12.27 Public functions of the moisture_sensor module.***

| Name of the function | Description of its functionality | Modules that use it |
|---|---|---|
| moistureSensorInit() | Has no functionality. | – |
| moistureSensorUpdate() | Updates the value of *hl69ProcessedValue*. | irrigation_system |
| moistureSensorRead() | Returns *hl69ProcessedValue*. | irrigation_control<br>user_interface |

***Table 12.28 Public functions of the buttons module.***

| Name of the function | Description of its functionality | Modules that use it |
|---|---|---|
| buttonsInit() | Configures all buttons in pull-up mode and sets initial configuration of the system after power on. | irrigation_system |
| buttonsUpdate() | Updates the values of *buttonsStatus*. | irrigation_system |
| buttonsRead() | Returns the values of *buttonsStatus*. | irrigation_control<br>user_interface |

***Table 12.29 Public functions of the irrigation_timer module.***

| Name of the function | Description of its functionality | Modules that use it |
|---|---|---|
| irrigationTimerInit() | Sets initial values of *irrigationTimer*. | irrigation_system |
| irrigationTimerUpdate() | Updates the values of *irrigationTimer*. | irrigation_system |
| irrigationTimerRead() | Returns the values of *irrigationTimer*. | irrigation_control |

***Table 12.30 Public functions of the irrigation_control module.***

| Name of the function | Description of its functionality | Modules that use it |
|---|---|---|
| irrigationControlInit() | *IrrigationState* is set to "INITIAL_MODE_ASSESSMENT"; *waitedTimeMustBeReset* and *waitedTimeMustBeReset* are set to true. | irrigation_system |
| irrigationControlUpdate() | Updates the FSM of the *irrigation_control* module. | irrigation_system |
| irrigationControlRead() | Returns the values of *irrigationControlStatus*. | user_interface<br>irrigation_timer<br>relay |

***Table 12.31 Public functions of the display module.***

| Name of the function | Description of its functionality | Modules that use it |
|---|---|---|
| userInterfaceInit() | Calls *displayInit()* and prints the text that does not change over time on the display. | irrigation_system |
| userInterfaceUpdate() | Updates the current values of mode and configurations in the display by means of the display driver. | irrigation_system |
| userInterfaceRead() | Has no functionality. | - |

***Table 12.32 Public functions of the relay module.***

| Name of the function | Description of its functionality | Modules that use it |
|---|---|---|
| relayInit() | Initializes the value of *relayControlPin*. | irrigation_system |
| relayUpdate() | Updates the value of *relayControlPin*. | irrigation_system |
| relayRead() | Has no functionality. | - |

In Figure 12.7, the proposed FSM is shown. After the start, the first state is INITIAL_MODE_ASSESSMENT. In this mode, the state of the Mode button is read (it is indicated as the *changeMode* variable in Figure 12.7). If it is pressed (*changeMode* is true), *irrigationControlStatus.irrigationState* (introduced in Table 12.23) is set to CONTINUOUS_MODE_IRRIGATING. If the Mode button is not being pressed (*changeMode* is false), then *irrigationControlStatus.irrigationState* is set to PROGRAMMED_MODE_WAITING_TO_IRRIGATE, and *irrigationControlStatus.waitedTimeMustBeReset* (Table 12.23) is set to true.

In the CONTINUOUS_MODE_IRRIGATING state, the Mode button is checked. If it is not pressed (*changeMode* is false), then *irrigationControlStatus.irrigationState* is not modified, and the FSM remains in the CONTINUOUS_MODE_IRRIGATING state. If the Mode button is pressed (*changeMode* is true), *irrigationControlStatus.irrigationState* is set to PROGRAMMED_MODE_WAITING_TO_IRRIGATE, and *irrigationControlStatus.waitedTimeMustBeReset* is set to true.

**NOTE:** The relay is controlled by the *relay* module based on the return value of *irrigationControlRead()*. If the read state is CONTINUOUS_MODE_IRRIGATING or PROGRAMMED_MODE_IRRIGATING, the relay will be activated, and it will be deactivated if the read state is INITIAL_MODE_ASSESSMENT, PROGRAMMED_MODE_WAITING_TO_IRRIGATE, or PROGRAMMED_MODE_IRRIGATION_SKIPPED.

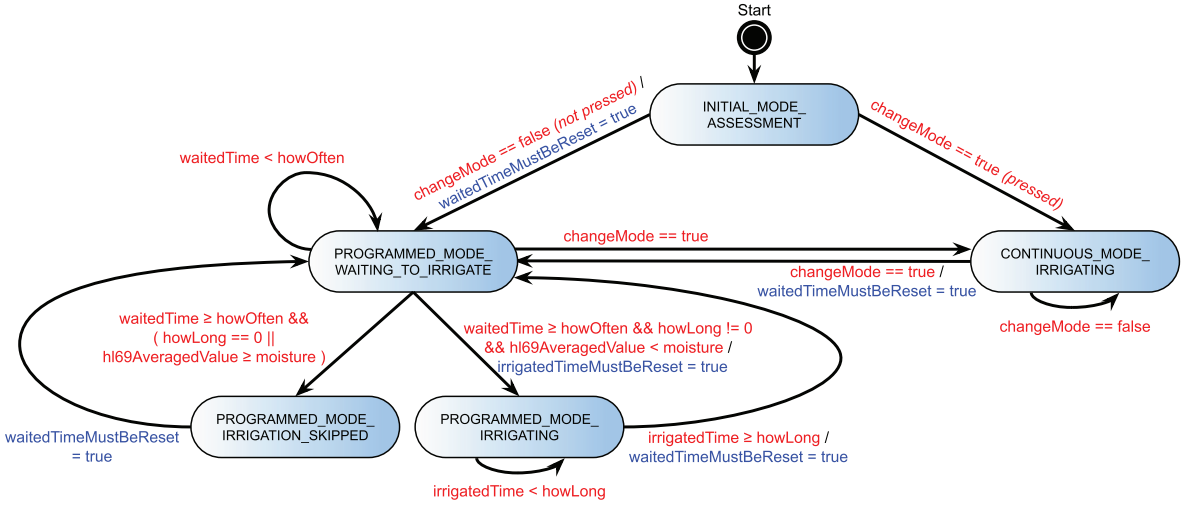



***Figure 12.7 Diagram of the proposed FSM.***

In the state PROGRAMMED_MODE_WAITING_TO_IRRIGATE, the first step is to set *irrigationControlStatus.waitedTimeMustBeReset* to false, because once in this state *irrigationTimer.waitedTime* (introduced in Table 12.22) has already been reset by the *irrigation_timer* module (see the Note below). Next, it is assessed whether the Mode button is pressed.

If it is (*changeMode* is true), *irrigationControlStatus.irrigationState* is set to CONTINUOUS_MODE_IRRIGATING. If the Mode button is not pressed, it is checked if *irrigationTimer.waitedTime* is smaller than *howOften*. If so, the FSM remains in PROGRAMMED_MODE_WAITING_TO_IRRIGATE. If *irrigationTimer.waitedTime* is equal to or greater than *howOften*, then the other conditions are checked.

**NOTE:** In the code to be implemented, if the FSM is in PROGRAMMED_MODE_WAITING _TO_IRRIGATE, the value of *irrigationTimer.waitedTime* will be increased by the *irrigation_timer* module each time *irrigationTimerUpdate()* is executed, and will be reset by the *irrigation_timer* module when it detects, by means of the return value of *irrigationControlRead()*, that *irrigationControlStatus.waitedTimeMustBeReset* is true. If the current state is PROGRAMMED_MODE_IRRIGATING, the *irrigation_timer* module will increase *irrigationTimer.irrigatedTime* (introduced in Table 12.22) each time *irrigationTimerUpdate()* is executed and will reset *irrigationTimer.irrigatedTime* if it detects that *irrigationControlStatus.irrigatedTimeMustBeReset* is true, by means of the return value of *irrigationControlRead()*. In this way, the modularization principle will not be violated, because only the *irrigation_timer* module will modify the irrigation and waiting timers.

If *hl69AveragedValue* is greater than or equal to the minimum moisture level configuration (*moisture*) or *irrigationtimer.howLong* is zero, then *irrigationControlStatus.irrigationState* is set to PROGRAMMED_MODE_IRRIGATION_SKIPPED. If *howLong* is not zero and *hl69AveragedValue* is smaller than the minimum moisture level configuration (*moisture*), then *irrigationControlStatus.irrigationState* is set to PROGRAMMED_MODE_IRRIGATING, and *irrigationControlStatus.irrigatedTimeMustBeReset* is set to true.

The state PROGRAMMED_MODE_IRRIGATION_SKIPPED is only used to show on the display that the irrigation was skipped. This is done by the *user_interface* module, which reads the value of *irrigationControlStatus.irrigationState* using *irrigationControlRead()*. In this state, *irrigationControlStatus.waitedTimeMustBeReset* is set to true, and *irrigationControlStatus.irrigationState* is set to PROGRAMMED_MODE_WAITING_TO_IRRIGATE.

In the state PROGRAMMED_MODE_IRRIGATING, *irrigationControlStatus.irrigatedTimeMustBeReset* is first set to false. Then, it assesses if *irrigationTimer.irrigatedTime* is smaller than *howLong*. After the irrigation is completed, it sets *irrigationControlStatus.waitedTimeMustBeReset* to true and *irrigationControlStatus.IrrigationState to* PROGRAMMED_MODE_WAITING_TO_IRRIGATE.

To conclude this step, in Figures 12.8 to 12.11 some sketches of the proposed layout of the LCD display are shown. On the first line of the display, the current irrigation mode is shown. It is also indicated whether the system is irrigating or not in the case of the Programmed irrigation mode. On the second and third lines of the display, the values of the “How Often” and “How Long” configurations are shown. On the fourth line, the minimum moisture level below which irrigation will be activated in the Programmed irrigation mode is shown on the left, and the current moisture measurement is shown on the right.

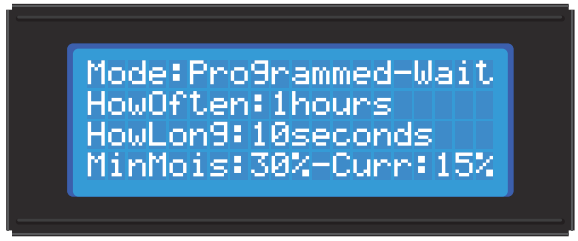



*Figure 12.8 Layout of the LCD for the Programmed irrigation mode when the system is waiting to irrigate.*


*Figure 12.9 Layout of the LCD for the Programmed irrigation mode when the system is irrigating.*

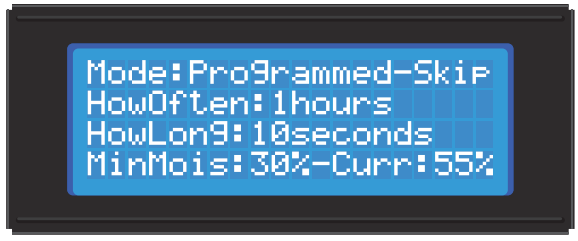



*Figure 12.10 Layout of the LCD for the Programmed irrigation mode when irrigation is skipped.*

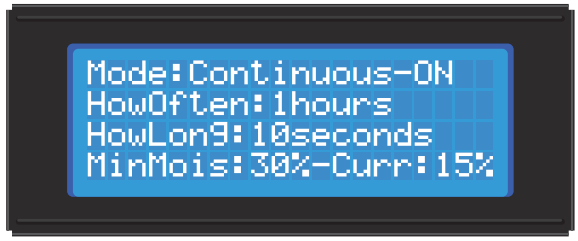



*Figure 12.11 Layout of the LCD for the Continuous irrigation mode.*

## Proposed Exercise

1. Design the software for the project that was selected in the "Proposed Exercise" of Example 12.1. Produce a diagram of the module interconnections and tables indicating all of the details. Include a diagram of the FSM designed to implement the functionality.

### Answer to the Exercise

1. It is strongly recommended to start by reusing as much as possible from the previous chapters of this book. Using the internet may help to find more ideas for the remaining parts of the software.

**Example 12.5: Implement the User Interface**

### Objective

Implement the system's user interfaces, as designed in step 4.

### Summary of the Expected Outcome

The result of this step is expected to be a set of folders and *.h* and *.cpp* files that implement the user interface.

### Discussion on how to Implement this Step

In this step, the software should be implemented as a set of *.h* and *.cpp* files. The layout of the software should follow the design that was created previously unless there are strong rationales to introduce changes. In the irrigation system, the user interface consists of four buttons and a character-based LCD display. Therefore, only the corresponding functionality is implemented in this step.

### Implementation of this Proposed Step

In Code 12.1, *main.cpp* is shown. On line 3, the library *irrigation_system.h* is included. The *main()* function is presented on lines 7 to 14. On line 9, the function *irrigationSystemInit()* is called, and on lines 10 to 13 the *superloop* is implemented. The function *irrigationSystemUpdate()* is called inside the superloop.

```
1  //=====[Libraries]=============================================================
2
3  #include "irrigation_system.h"
4
5  //=====[Main function, the program entry point after power on or reset]========
6
7  int main()
8  {
9      irrigationSystemInit();
10     while (true) {
11         irrigationSystemUpdate();
12     }
13 }
14
```

***Code 12.1 Implementation of main.cpp.***

In Code 12.2, the implementation of *irrigation_system.h* is shown. On line 8, SYSTEM_TIME_INCREMENT_MS is defined, and the public functions *irrigationSystemInit()* and *irrigationSystemUpdate()* are declared.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _IRRIGATION_SYSTEM_H_
4  #define _IRRIGATION_SYSTEM_H_
5
6  //=====[Declaration of public defines]=========================================
7
8  #define SYSTEM_TIME_INCREMENT_MS   100
9
10 //=====[Declarations (prototypes) of public functions]=========================
11
12 void irrigationSystemInit();
13 void irrigationSystemUpdate();
14
15 //=====[#include guards - end]=================================================
16
17 #endif // _IRRIGATION_SYSTEM_H_
```

***Code 12.2 Implementation of irrigation_system.h.***

In Code 12.3, the implementation of *irrigation_system.cpp* is shown. From lines 3 to 7, libraries are included. On line 11, the private variable *irrigationSystemDelay* of type *nonBlockingDelay_t* is declared. This variable will be used to implement the non-blocking delay. On line 15, the function *irrigationSystemInit()* is implemented. First, *tickInit()* is called. Then, *buttonsInit()* and *userInterfaceInit()* are called. Finally, on line 20, the non-blocking delay is initialized to SYSTEM_TIME_INCREMENT_MS.

On line 23, *irrigationSystemUpdate()* is implemented. Line 25 assesses whether *irrigationSystemDelay* has reached the value set by *nonBlockingDelayInit()*. In that case, *buttonsUpdate()* and *userInterfaceUpdate()* are called.

**NOTE:** For the sake of brevity, only the file sections that have some content are shown in the Codes. The full versions of the files are available in [2].

```
1  //=====[Libraries]=============================================================
2
3  #include "irrigation_system.h"
4
5  #include "buttons.h"
6  #include "user_interface.h"
7  #include "non_blocking_delay.h"
8
9  //=====[Declaration and initialization of public global variables]=============
10
11 static nonBlockingDelay_t irrigationSystemDelay;
12
13 //=====[Implementations of public functions]===================================
```

```
14
15  void irrigationSystemInit()
16  {
17      tickInit();
18      buttonsInit();
19      userInterfaceInit();
20      nonBlockingDelayInit( &irrigationSystemDelay, SYSTEM_TIME_INCREMENT_MS );
21  }
22
23  void irrigationSystemUpdate()
24  {
25      if( nonBlockingDelayRead(&irrigationSystemDelay) ) {
26          buttonsUpdate();
27          userInterfaceUpdate();
28      }
29  }
```

***Code 12.3 Implementation of irrigation_system.cpp.***

In Code 12.4, the implementation of *buttons.h* is shown. From line 8 to line 16, nine definitions are introduced. They are used to implement requirements 3.1.2 to 3.1.7, as shown in Table 12.33. On line 20, the type definition of the struct named *buttonsStatus_t* is shown. This struct has four members: *changeMode*, which is used to keep track of the mode; *howOften*, which is used to keep track of the value of the "How often" configuration; *howLong*, which is used to keep track of the value of the "How long" configuration; and *moisture*, which used to keep track of the value of the "Minimum moisture level" configuration. On lines 29 to 31, the three public functions of this module are declared: *buttonsInit()*, *buttonsUpdate()*, and *buttonsRead()*. The latter is the only one that has a return value, which is of type *buttonsStatus_t*.

In Code 12.5, the implementation of *buttons.cpp* is shown. On lines 10 to 13, the private DigitalIn objects *changeModeButton*, *howOftenButton*, *howLongButton*, and *moistureButton* are declared and assigned to PG_1, PF_9, PF_7, and PF_8, respectively. On line 17, the private variable *buttonsStatus* of the type *buttonsStatus_t* is declared. On line 21, *buttonsInit()* is implemented. First, all of the buttons are configured with pull-up resistors (lines 23 to 26) and then the members of *buttonsStatus* are set to an initial value (lines 28 to 31).

**NOTE:** In previous chapters, objects were declared public in order to simplify the software implementation. In this chapter, objects are declared private to improve the software modularity.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _BUTTONS_H_
4  #define _BUTTONS_H_
5
6  //=====[Declaration of public defines]=========================================
7
8  #define HOW_OFTEN_INCREMENT   1
9  #define HOW_OFTEN_MIN         1
10 #define HOW_OFTEN_MAX         24
11 #define HOW_LONG_INCREMENT    10
```

```
12  #define HOW_LONG_MIN          0
13  #define HOW_LONG_MAX          90
14  #define MOISTURE_INCREMENT    5
15  #define MOISTURE_MIN          0
16  #define MOISTURE_MAX          95
17
18  //=====[Declaration of public data types]======================================
19
20  typedef struct buttonsStatus {
21      bool changeMode;
22      int howOften;
23      int howLong;
24      int moisture;
25  } buttonsStatus_t;
26
27  //=====[Declarations (prototypes) of public functions]=========================
28
29  void buttonsInit();
30  void buttonsUpdate();
31  buttonsStatus_t buttonsRead();
32
33  //=====[#include guards - end]=================================================
34
35  #endif // _BUTTONS_H_
```

***Code 12.4 Implementation of buttons.h.***

***Table 12.33 Some of the initial requirements defined for the home irrigation system.***

| Req. Group | Req. ID | Description |
|---|---|---|
| 3. Configuration | 3.1 | The system configuration will be done by means of a set of buttons: |
| | 3.1.1 | The "Mode" button will change between "Programmed irrigation" and "Continuous irrigation". |
| | 3.1.2 | The "How often" button will increase the time between irrigations in programmed mode by one hour. |
| | 3.1.3 | The "How long" button will increase the irrigation time in programmed mode by ten seconds. |
| | 3.1.4 | The "Moisture" button will increase the "Minimum moisture level" configuration by 5%. |
| | 3.1.5 | The maximum values are: "How often": 24 h; "How long": 90 s; "Moisture": 95%. |
| | 3.1.6 | "How long" and "Moisture" will be set to 0 (zero) if they reach the maximum and the button is pressed. |
| | 3.1.7 | "How often" will be set to 1 if it reaches its maximum value and the button is pressed. |

On line 34, the function *buttonsUpdate()* is implemented. First, *buttonsStatus.changeMode* is assigned the value of *!changeModeButton* because *changeModeButton* is configured with a pull-up resistor. On line 38, *howOftenButton* is assessed. If it is pressed (*howOftenButton* is false), then *buttonsStatus.howOften* is incremented by HOW_OFTEN_INCREMENT in line 39. On line 40, it is assessed if *buttonsStatus.howOften* is greater than or equal to HOW_OFTEN_MAX + HOW_OFTEN_INCREMENT. If so, it is assigned HOW_OFTEN_MIN on line 41.

The code on lines 45 to 50, and the code on lines 52 to 57, are very similar to the code from lines 38 to 43 and, therefore, are not discussed line by line.

Finally, on lines 60 to 63, the function *buttonsRead()* is implemented. This function returns the value of the private variable *buttonsStatus*. In this way, other modules can get the values of the

*buttonsStatus* members *changeMode*, *howOften*, *howLong*, and *moisture* without violating the principle of modularization.

```
//=====[Libraries]=============================================================

#include "mbed.h"
#include "arm_book_lib.h"

#include "buttons.h"

//=====[Declaration and initialization of private global objects]===============

static DigitalIn changeModeButton(PG_1);
static DigitalIn howOftenButton(PF_9);
static DigitalIn howLongButton(PF_7);
static DigitalIn moistureButton(PF_8);

//=====[Declaration and initialization of private global variables]============

static buttonsStatus_t buttonsStatus;

//=====[Implementations of public functions]===================================

void buttonsInit()
{
    changeModeButton.mode(PullUp);
    howOftenButton.mode(PullUp);
    howLongButton.mode(PullUp);
    moistureButton.mode(PullUp);

    buttonsStatus.changeMode = OFF;
    buttonsStatus.howOften = HOW_OFTEN_MIN;
    buttonsStatus.howLong = HOW_LONG_MIN;
    buttonsStatus.moisture = MOISTURE_MIN;
}

void buttonsUpdate()
{
    buttonsStatus.changeMode = !changeModeButton;

   if ( !howOftenButton ) {
        buttonsStatus.howOften = buttonsStatus.howOften + HOW_OFTEN_INCREMENT;
        if ( buttonsStatus.howOften >= HOW_OFTEN_MAX + HOW_OFTEN_INCREMENT) {
            buttonsStatus.howOften = HOW_OFTEN_MIN;
        }
   }

   if ( !howLongButton ) {
        buttonsStatus.howLong = buttonsStatus.howLong + HOW_LONG_INCREMENT;
        if ( buttonsStatus.howLong >= HOW_LONG_MAX + HOW_LONG_INCREMENT) {
            buttonsStatus.howLong = HOW_LONG_MIN;
        }
   }

   if ( !moistureButton ) {
        buttonsStatus.moisture = buttonsStatus.moisture + MOISTURE_INCREMENT;
        if ( buttonsStatus.moisture >= MOISTURE_MAX + MOISTURE_INCREMENT) {
           buttonsStatus.moisture = MOISTURE_MIN;
        }
   }
}

buttonsStatus_t buttonsRead()
{
    return buttonsStatus;
}
```

***Code 12.5 Implementation of buttons.cpp.***

In Code 12.6, the implementation of *user_interface.h* is shown. The private functions *userInterfaceInit()*, *userInterfaceUpdate()*, and *userInterfaceRead()* are declared on lines 8 to 10.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _USER_INTERFACE_H_
4  #define _USER_INTERFACE_H_
5
6  //=====[Declarations (prototypes) of public functions]=========================
7
8  void userInterfaceInit();
9  void userInterfaceUpdate();
10 void userInterfaceRead();
11
12 //=====[#include guards - end]=================================================
13
14 #endif // _USER_INTERFACE_H_
```

***Code 12.6 Implementation of user_interface.h.***

In Code 12.7, the first part of the implementation of *user_interface.cpp* is shown. On lines 3 to 9, the libraries are included. From lines 13 to 27, the function *userInterfaceInit()* is implemented. On line 15, *displayInit()* is used to establish that a character-based display in 4-bit mode is used. On line 17, *displayClear()* is used to clear the display. Lines 19 to 25 are used to write some text on the display following the design introduced in Figure 12.8. This text does not change in this example, even if the buttons are pressed.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "user_interface.h"
7
8  #include "display.h"
9  #include "buttons.h"
10
11 //=====[Implementations of public functions]===================================
12
13 void userInterfaceInit()
14 {
15     displayInit( DISPLAY_TYPE_LCD_HD44780,DISPLAY_CONNECTION_GPIO_4BITS );
16
17     displayClear();
18     displayCharPositionWrite( 0, 0 );
19     displayStringWrite( "Mode:Programmed-Wait" );
20     displayCharPositionWrite( 0, 1 );
21     displayStringWrite( "HowOften:  hours" );
22     displayCharPositionWrite( 0, 2 );
23     displayStringWrite( "HowLong:  seconds" );
24     displayCharPositionWrite( 0, 3 );
25     displayStringWrite( "MinMois:  %-Curr:15%" );
26 }
```

***Code 12.7 Implementation of user_interface.cpp (Part 1/2).***

From lines 1 to 20 of Code 12.8, the function *userInterfaceUpdate()* is implemented. On line 3, an array of char named *number* is declared. On line 5, a variable named *buttonsStatusLocalCopy* of type *buttonsStatus_t* is declared. On line 7, *buttonsStatusLocalCopy* is assigned the return value of *buttonsRead()*. Lines 9 to 11 are used to write the value of *buttonsStatusLocalCopy.howOften* to location (9,1) of the display. The format tag prototype "%02d" is used to indicate that two characters should be used for the value. If the value to be written has less than two characters, the result is padded with leading zeros. A very similar approach is used to write *buttonsStatusLocalCopy.howLong* (lines 13 to 15) and *buttonsStatusLocalCopy.moisture* (lines 17 to 19). Lastly, on line 22, it can be seen that *userInterfaceRead()* has no functionality.

```
 1  void userInterfaceUpdate()
 2  {
 3      char number[3];
 4
 5      buttonsStatus_t buttonsStatusLocalCopy;
 6
 7      buttonsStatusLocalCopy = buttonsRead();
 8
 9      displayCharPositionWrite( 9, 1 );
10      sprintf( number, "%02d", buttonsStatusLocalCopy.howOften );
11      displayStringWrite( number );
12
13      displayCharPositionWrite( 8, 2 );
14      sprintf( number, "%02d", buttonsStatusLocalCopy.howLong );
15      displayStringWrite( number );
16
17      displayCharPositionWrite( 8, 3 );
18      sprintf( number, "%02d", buttonsStatusLocalCopy.moisture );
19      displayStringWrite( number );
20  }
21
22  void userInterfaceRead()
23  {
24  }
```

***Code 12.8 Implementation of user_interface.cpp (Part 2/2).***

Finally, given that the pins used to connect the display (Table 12.12) are the same pins used in Chapter 6, it is not necessary to modify the pins assignment used in *display.cpp*. If the display were to be connected to other pins, then *displayEN*, *displayRS*, *displayD4*, *displayD5*, *displayD6*, and *displayD7* would have to be assigned with the corresponding pins.

## Proposed Exercise

1. Implement the system's user interface for the project that was selected in the "Proposed Exercise" of Example 12.1. Write all the corresponding *.h* and *.cpp* files.

## Answer to the Exercise

1. It is strongly recommended to start by reusing as much as possible from the previous chapters of this book, or even from this example.

### Example 12.6: Implement the Reading of the Sensors

#### Objective

Implement the reading of the sensors, as designed in step 4.

#### Summary of the Expected Outcome

As a result of this step, it is expected to have a set of *.h* and *.cpp* files that implement the user interface and the reading of the sensors.

#### Discussion on How to Implement this Step

As was previously mentioned, the layout of the software should follow the design that was done in step 4 unless there are strong rationales to introduce changes. The irrigation system has only one sensor, the moisture sensor. Therefore, only the corresponding functionality is implemented in this step.

#### Implementation of this Proposed Step

For the sake of brevity, in this example only new files or files that have changes are shown. The full set of files for this example are available in [2]. In particular, new lines are introduced in *irrigation_system.cpp*, as can be seen in Code 12.9. The library *moisture_sensor.h* is included on line 8, and the functions *moistureSensorInit()* and *moistureSensorUpdate()* are called on lines 21 and 30, respectively.

There are also a few new lines in *user_interface.cpp*, as can be seen in Code 12.10. On line 10, the library *moisture_sensor.h* is included. On line 26, "MinMois: %-Curr: %" is written. On line 34, the float variable *hl69AveragedValueLocalCopy* is declared, and on line 37 it is assigned the return value of *moistureSensorRead()*. Lines 51 to 53 are included in order to write the reading of the moisture sensor.

```
1  //=====[Libraries]=============================================================
2
3  #include "irrigation_system.h"
4
5  #include "buttons.h"
6  #include "user_interface.h"
7  #include "non_blocking_delay.h"
8  #include "moisture_sensor.h"
9
10 //=====[Declaration and initialization of public global variables]=============
11
12 static nonBlockingDelay_t irrigationSystemDelay;
13
14 //=====[Implementations of public functions]===================================
15
16 void irrigationSystemInit()
17 {
18     tickInit();
19     buttonsInit();
20     userInterfaceInit();
```

```
21      moistureSensorInit();
22      nonBlockingDelayInit( &irrigationSystemDelay, SYSTEM_TIME_INCREMENT_MS );
23  }
24
25  void irrigationSystemUpdate()
26  {
27      if( nonBlockingDelayRead(&irrigationSystemDelay) ) {
28          buttonsUpdate();
29          userInterfaceUpdate();
30          moistureSensorUpdate();
31      }
32  }
```

***Code 12.9 New implementation of irrigation_system.cpp.***

```
1   //=====[Libraries]=============================================================
2
3   #include "mbed.h"
4   #include "arm_book_lib.h"
5
6   #include "user_interface.h"
7
8   #include "display.h"
9   #include "buttons.h"
10  #include "moisture_sensor.h"
11
12  //=====[Implementations of public functions]===================================
13
14  void userInterfaceInit()
15  {
16      displayInit( DISPLAY_TYPE_LCD_HD44780,DISPLAY_CONNECTION_GPIO_4BITS );
17
18      displayClear();
19      displayCharPositionWrite( 0, 0 );
20      displayStringWrite( "Mode:Programmed-Wait" );
21      displayCharPositionWrite( 0, 1 );
22      displayStringWrite( "HowOften:  hours" );
23      displayCharPositionWrite( 0, 2 );
24      displayStringWrite( "HowLong:  seconds" );
25      displayCharPositionWrite( 0, 3 );
26      displayStringWrite( "MinMois:  %-Curr:  %" );
27  }
28
29  void userInterfaceUpdate()
30  {
31      char number[3];
32
33      buttonsStatus_t buttonsStatusLocalCopy;
34      float hl69AveragedValueLocalCopy;
35
36      buttonsStatusLocalCopy = buttonsRead();
37      hl69AveragedValueLocalCopy = moistureSensorRead();
38
39      displayCharPositionWrite( 9, 1 );
40      sprintf( number, "%02d", buttonsStatusLocalCopy.howOften );
41      displayStringWrite( number );
42
43      displayCharPositionWrite( 8, 2 );
44      sprintf( number, "%02d", buttonsStatusLocalCopy.howLong );
45      displayStringWrite( number );
46
```

```
47      displayCharPositionWrite( 8, 3 );
48      sprintf( number, "%02d", buttonsStatusLocalCopy.moisture );
49      displayStringWrite( number );
50
51      displayCharPositionWrite( 17, 3 );
52      sprintf( number, "%2.0f", 100*hl69AveragedValueLocalCopy );
53      displayStringWrite( number );
54  }
55
56  void userInterfaceRead()
57  {
58  }
```

***Code 12.10 New implementation of user_interface.cpp.***

In Code 12.11, the implementation of *moisture_sensor.h* is shown. It can be seen that the public functions *moistureSensorInit()*, *moistureSensorUpdate()*, and *moistureSensorRead()* are declared.

```
1   //=====[#include guards - begin]===============================================
2
3   #ifndef _MOISTURE_SENSOR_H_
4   #define _MOISTURE_SENSOR_H_
5
6   //=====[Declarations (prototypes) of public functions]=========================
7
8   void moistureSensorInit();
9   void moistureSensorUpdate();
10  float moistureSensorRead();
11
12  //=====[#include guards - end]=================================================
13
14  #endif // _MOISTURE_SENSOR_H_
```

***Code 12.11 Implementation of moisture_sensor.h.***

In Code 12.12, the first part of the implementation of *moisture_sensor.cpp* is shown. From lines 3 to 6, the libraries are included. On line 10, NUMBER_OF_AVERAGED_SAMPLES is defined as 10. On line 14, the AnalogIn object *hl69* is declared and assigned to the A3 pin. On line 18, the private float variable *hl69AveragedValue*, which will be used to store the average of the last NUMBER_OF_AVERAGED_ SAMPLES, is declared and initialized. The private array of float variable *hl69ReadingsArray* is declared on line 19. On line 23, the implementation of the function *moistureSensorInit()* is shown. As can be seen, it has no functionality.

In Code 12.13, the implementation of the function *moistureSensorUpdate()* is shown on lines 1 to 18. Note that it is very similar to the way in which the LM35 sensor was read earlier in this book. On line 3, a static integer variable named *hl69SampleIndex* is declared. On line 4, an integer variable *i* is declared. On line 6, the result of 1 minus the reading of the HL-69 sensor is assigned to the corresponding position of *hl69ReadingsArray*. The assignment is done this way because the sensor retrieves 1 when the moisture is 0%. On lines 8 to 12, *hl69AveragedValue* is computed. On lines 14 to 17, the value of *hl69SampleIndex* is updated. Finally, the function *moistureSensorRead()* is implemented on lines 20 to 23.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "moisture_sensor.h"
7
8  //=====[Declaration of private defines]========================================
9
10 #define NUMBER_OF_AVERAGED_SAMPLES    10
11
12 //=====[Declaration and initialization of private global objects]===============
13
14 static AnalogIn hl69(A3);
15
16 //=====[Declaration and initialization of private global variables]============
17
18 static float hl69AveragedValue = 0.0;
19 static float hl69ReadingsArray[NUMBER_OF_AVERAGED_SAMPLES];
20
21 //=====[Implementations of public functions]===================================
22
23 void moistureSensorInit()
24 {
25 }
```

*Code 12.12 Implementation of moisture_sensor.cpp (Part 1/2).*

```
1  void moistureSensorUpdate()
2  {
3      static int hl69SampleIndex = 0;
4      int i;
5
6      hl69ReadingsArray[hl69SampleIndex] = 1 - hl69.read();
7
8      hl69AveragedValue = 0.0;
9      for (i = 0; i < NUMBER_OF_AVERAGED_SAMPLES; i++) {
10         hl69AveragedValue = hl69AveragedValue + hl69ReadingsArray[i];
11     }
12     hl69AveragedValue = hl69AveragedValue / NUMBER_OF_AVERAGED_SAMPLES;
13
14     hl69SampleIndex++;
15     if ( hl69SampleIndex >= NUMBER_OF_AVERAGED_SAMPLES) {
16         hl69SampleIndex = 0;
17     }
18 }
19
20 float moistureSensorRead()
21 {
22     return hl69AveragedValue;
23 }
```

*Code 12.13 Implementation of moisture_sensor.cpp (Part 2/2).*

**NOTE:** The fact that the positions of the *hl69ReadingsArray* are not initialized and *hl69AveragedValue* is calculated using those values does not lead to incorrect behavior, as this situation lasts for only the first second after power on, during which the Programmed mode is still waiting to irrigate.

### Proposed Exercise

1. Implement the reading of the sensors for the project that was selected in the "Proposed Exercise" of Example 12.1. Write all the corresponding *.h* and *.cpp* files.

### Answer to the Exercise

1. It is strongly recommended to start by reusing as much as possible from the previous chapters of this book, or even from this example.

## Example 12.7: Implement the Driving of the Actuators

### Objective

Implement the drivers for the actuators, as designed in step 4.

### Summary of the Expected Outcome

As a result of this step, it is expected to have a set of *.h* and *.cpp* files that implement the user interface, the reading of the sensors, and the drivers for the actuators.

### Discussion of How to Implement this Step

The irrigation system has only one actuator, the solenoid valve, which is activated by means of a relay module. In this example, the activation of this relay module is implemented.

### Implementation of this Proposed Step

New lines are introduced in *irrigation_system.cpp*, as can be seen on lines 9, 23, and 33 of Code 12.14. The other files that were introduced in previous examples are not changed.

In Code 12.15, the implementation of *relay.h* is shown. It can be seen that the public functions *relayInit()*, *relayUpdate()*, and *relayRead()* are declared.

```
1  //=====[Libraries]=============================================================
2
3  #include "irrigation_system.h"
4
5  #include "buttons.h"
6  #include "user_interface.h"
7  #include "non_blocking_delay.h"
8  #include "moisture_sensor.h"
9  #include "relay.h"
10
11 //=====[Declaration and initialization of public global variables]=============
12
13 static nonBlockingDelay_t irrigationSystemDelay;
14
```

```
15  //=====[Implementations of public functions]===================================
16
17  void irrigationSystemInit()
18  {
19      tickInit();
20      buttonsInit();
21      userInterfaceInit();
22      moistureSensorInit();
23      relayInit();
24      nonBlockingDelayInit( &irrigationSystemDelay, SYSTEM_TIME_INCREMENT_MS );
25  }
26
27  void irrigationSystemUpdate()
28  {
29      if( nonBlockingDelayRead(&irrigationSystemDelay) ) {
30          buttonsUpdate();
31          userInterfaceUpdate();
32          moistureSensorUpdate();
33          relayUpdate();
34      }
35  }
```

***Code 12.14 New implementation of irrigation_system.cpp.***

```
1   //=====[#include guards - begin]===============================================
2
3   #ifndef _RELAY_H_
4   #define _RELAY_H_
5
6   //=====[Declarations (prototypes) of public functions]=========================
7
8   void relayInit();
9   void relayUpdate();
10  float relayRead();
11
12  //=====[#include guards - end]=================================================
13
14  #endif // _RELAY_H_
```

***Code 12.15 Implementation of relay.h.***

In Code 12.16, the Implementation of *relay.cpp* is shown. On lines 3 to 8, the libraries are included. On line 12, a private DigitalInOut object named *relayControlPin* is declared and assigned PF_2. The function r*elayInit()* initializes *relayControlPin* as an input, which turns off the relay.

**NOTE:** This same type of initialization and use was implemented in previous chapters to control the buzzer using the relay, given that buzzers are 5 V devices, and it is not advised to turn them on directly using a digitalOut object.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "relay.h"
7
8  #include "buttons.h"
9
10 //=====[Declaration and initialization of private global objects]===============
11
12 static DigitalInOut relayControlPin(PF_2);
13
14 //=====[Implementations of public functions]===================================
15
16 void relayInit()
17 {
18     relayControlPin.mode(OpenDrain);
19     relayControlPin.input();
20 }
21
22 void relayUpdate()
23 {
24     buttonsStatus_t buttonsStatusLocalCopy;
25
26     buttonsStatusLocalCopy = buttonsRead();
27
28    if( buttonsStatusLocalCopy.changeMode ) {
29         relayControlPin.output();
30         relayControlPin = LOW;
31     } else {
32              relayControlPin.input();
33     }
34 }
35
36 void relayRead()
37 {
38 }
```

***Code 12.16 Implementation of relay.cpp.***

The implementation of the function *relayUpdate()* is shown from lines 22 to 34. On line 24, *buttonsStatusLocalCopy*, a variable of type *buttonsStatus_t*, is declared. On line 26, *buttonsRead()* is used to load the status of the buttons into this variable. On line 28, *buttonsStatusLocalCopy.changeMode* is assessed. If it is true, then the relay is turned on by means of the statements on lines 29 and 30. Otherwise, the relay is turned off on line 32. In this way, the relay should become active each time the Mode button is pressed.

The function *relayRead()*, which has no functionality, is shown on lines 36 to 38.

**NOTE:** The behavior implemented in this example only has the purpose of testing the activation of the relay. The behavior will be changed in the next example as the remaining modules of the software are incorporated in the system implementation.

### Proposed Exercise

1. Implement the drivers of the actuators for the project that was selected in the "Proposed Exercise" of Example 12.1. Write all the corresponding *.h* and *.cpp* files.

### Answer to the Exercise

1. It is strongly recommended to start by reusing as much as possible from the previous chapters of this book, or even from this example. Remember that a quick look on the internet may help to find more ideas for the remaining parts of the software.

## Example 12.8: Implement the Behavior of the System

### Objective

Implement the behavior of the system as established in previous steps (Table 12.6 and Figure 12.7).

### Summary of the Expected Outcome

As a result of this step, it is expected to have a set of *.h* and *.cpp* files that implement the complete behavior of the system.

### Discussion on How to Implement this Step

In this step, the last two remaining modules, *irrigation_timer* and *irrigation_control*, are included in the system, and all the functionality described in the requirements is implemented.

### Implementation of this Proposed Step

New lines are introduced in *irrigation_system.cpp*, as can be seen on lines 10, 11, 25, 26, 37, and 38 of Code 12.17. In this new implementation, *userInterfaceInit()* and *userInterfaceUpdate()* are called after all the other initialization and update function calls to properly implement the logic introduced in Example 12.4.

```
1  //=====[Libraries]=============================================================
2
3  #include "irrigation_system.h"
4
5  #include "buttons.h"
6  #include "user_interface.h"
7  #include "non_blocking_delay.h"
8  #include "moisture_sensor.h"
9  #include "relay.h"
10 #include "irrigation_control.h"
11 #include "irrigation_timer.h"
12
13 //=====[Declaration of private defines]========================================
14
```

```
15  static nonBlockingDelay_t irrigationSystemDelay;
16
17  //=====[Implementations of public functions]===================================
18
19  void irrigationSystemInit()
20  {
21      tickInit();
22      buttonsInit();
23      moistureSensorInit();
24      relayInit();
25      irrigationControlInit();
26      irrigationTimerInit();
27      userInterfaceInit();
28      nonBlockingDelayInit( &irrigationSystemDelay, SYSTEM_TIME_INCREMENT_MS );
29  }
30
31  void irrigationSystemUpdate()
32  {
33      if( nonBlockingDelayRead(&irrigationSystemDelay) ) {
34          buttonsUpdate();
35          moistureSensorUpdate();
36          relayUpdate();
37          irrigationControlUpdate();
38          irrigationTimerUpdate();
39          userInterfaceUpdate();
40      }
41  }
```

***Code 12.17 New implementation of irrigation_system.cpp.***

In Code 12.18, the implementation of *irrigation_control.h* is shown. On line 8, TO_SECONDS is defined as 10. This #define will be used to convert from a number of counts of 100 milliseconds to seconds. On line 9, TO_HOURS is defined as 36000. This #define will be used to convert from a number of counts of 100 milliseconds to hours.

On line 13, the public data type *irrigationState_t* is declared. As can be seen on lines 14 to 18, it can have five possible values. On line 21, the public data type *irrigationControlStatus_t* is declared. Its first member is *irrigationState*, of type *irrigationState_t*. The other two members are the Boolean variables, *waitedTimeMustBeReset* and *irrigatedTimeMustBeReset*. On lines 29 to 31, the prototypes of the public functions *irrigationControlInit()*, *irrigationControlUpdate()*, and *irrigationControlRead()* are declared.

```
1   //=====[#include guards - begin]===============================================
2
3   #ifndef _IRRIGATION_CONTROL_H_
4   #define _IRRIGATION_CONTROL_H_
5
6   //=====[Declaration of public defines]=========================================
7
8   #define TO_SECONDS   10
9   #define TO_HOURS     36000
10
11  //=====[Declaration of public data types]======================================
12
13  typedef enum {
14      INITIAL_MODE_ASSESSMENT,
```

```
15      CONTINUOUS_MODE_IRRIGATING,
16      PROGRAMMED_MODE_WAITING_TO_IRRIGATE,
17      PROGRAMMED_MODE_IRRIGATION_SKIPPED,
18      PROGRAMMED_MODE_IRRIGATING
19  } irrigationState_t;
20
21  typedef struct irrigationControlStatus {
22      irrigationState_t irrigationState;
23      bool waitedTimeMustBeReset;
24      bool irrigatedTimeMustBeReset;
25  } irrigationControlStatus_t;
26
27  //=====[Declarations (prototypes) of public functions]=======================
28
29  void irrigationControlInit();
30  void irrigationControlUpdate();
31  irrigationControlStatus_t irrigationControlRead();
32
33  //=====[#include guards - end]===============================================
34
35  #endif // _IRRIGATION_CONTROL_H_
```

***Code 12.18 Implementation of irrigation_control.h.***

In Code 12.19 and Code 12.20, the implementation of *irrigation_control.cpp* is shown. The libraries are included on lines 3 to 10 of Code 12.19. A private global variable named *irrigationControlStatus* is declared on line 14. The function *irrigationControlInit()*, declared on line 18, is used to initialize the members of *irrigationControlStatus*. The function *irrigationControlUpdate()*, shown from lines 25 to 65 of Code 12.19 and 1 to 40 of Code 12.20, implements the FSM discussed in Example 12.4. It is, therefore, not discussed here. Finally, in Code 12.20, the function *irrigationControlRead()* is implemented on lines 42 to 45 of Code 12.20.

**NOTE:** FSM transitions corresponding to *waitedTime* < *howOften*, *irrigatedTime* < *howLong*, *changeMode* == false (see Figure 12.7) are to the same state, so there is no need to include code to implement them. Lines 8 to 10 of Code 12.20 are included only to improve code clarity but can be replaced by *else irrigationControlStatus*. *irrigationState* = PROGRAMMED_MODE_IRRIGATION_SKIPPED.

```
1   //=====[Libraries]=============================================================
2
3   #include "mbed.h"
4   #include "arm_book_lib.h"
5
6   #include "irrigation_control.h"
7
8   #include "buttons.h"
9   #include "irrigation_timer.h"
10  #include "moisture_sensor.h"
11
12  //=====[Declaration and initialization of private global variables]============
13
14  static irrigationControlStatus_t irrigationControlStatus;
15
```

```
16  //=====[Implementations of public functions]=================================
17
18  void irrigationControlInit()
19  {
20      irrigationControlStatus.irrigationState = INITIAL_MODE_ASSESSMENT;
21      irrigationControlStatus.waitedTimeMustBeReset = true;
22      irrigationControlStatus.irrigatedTimeMustBeReset = true;
23  }
24
25  void irrigationControlUpdate()
26  {
27      buttonsStatus_t buttonsStatusLocalCopy;
28      irrigationTimer_t irrigationTimerLocalCopy;
29      float hl69AveragedValueLocalCopy;
30
31      buttonsStatusLocalCopy = buttonsRead();
32      irrigationTimerLocalCopy = irrigationTimerRead();
33      hl69AveragedValueLocalCopy = moistureSensorRead();
34
35      switch( irrigationControlStatus.irrigationState ) {
36
37          case INITIAL_MODE_ASSESSMENT:
38
39              if( buttonsStatusLocalCopy.changeMode ) {
40                 irrigationControlStatus.irrigationState = CONTINUOUS_MODE_IRRIGATING;
41              } else {
42                     irrigationControlStatus.irrigationState =
43                                                 PROGRAMMED_MODE_WAITING_TO_IRRIGATE;
44                  irrigationControlStatus.waitedTimeMustBeReset = true;
45              }
46              break;
47
48          case CONTINUOUS_MODE_IRRIGATING:
49
50              if( buttonsStatusLocalCopy.changeMode ) {
51                  irrigationControlStatus.irrigationState =
52                                                 PROGRAMMED_MODE_WAITING_TO_IRRIGATE;
53                  irrigationControlStatus.waitedTimeMustBeReset = true;
54              }
55              break;
56
57          case PROGRAMMED_MODE_WAITING_TO_IRRIGATE:
58
59              irrigationControlStatus.waitedTimeMustBeReset = false;
60              if( buttonsStatusLocalCopy.changeMode ) {
61                  irrigationControlStatus.irrigationState =
62                                                 CONTINUOUS_MODE_IRRIGATING;
63              }
64              else if( irrigationTimerLocalCopy.waitedTime >= (
65                       buttonsStatusLocalCopy.howOften * TO_HOURS) ) {
```

***Code 12.19 Implementation of irrigation_control.cpp (Part 1/2).***

```
                if( ( buttonsStatusLocalCopy.howLong != 0 ) &&
                    ( (int) (100*hl69AveragedValueLocalCopy) <
                       buttonsStatusLocalCopy.moisture ) ) {
                irrigationControlStatus.irrigationState =
                                            PROGRAMMED_MODE_IRRIGATING;
                irrigationControlStatus.irrigatedTimeMustBeReset = true;
                }
                else if ( ( buttonsStatusLocalCopy.howLong == 0 ) ||
                          ( (int) (100*hl69AveragedValueLocalCopy) >=
                            buttonsStatusLocalCopy.moisture ) ) {
                irrigationControlStatus.irrigationState =
                                            PROGRAMMED_MODE_IRRIGATION_SKIPPED;
                }
            }

        case PROGRAMMED_MODE_IRRIGATION_SKIPPED:

            irrigationControlStatus.waitedTimeMustBeReset = true;
           irrigationControlStatus.irrigationState =
                                            PROGRAMMED_MODE_WAITING_TO_IRRIGATE;

            break;

        case PROGRAMMED_MODE_IRRIGATING:

            irrigationControlStatus.waitedTimeMustBeReset = false;

            if( irrigationTimerLocalCopy.irrigatedTime >= (
                buttonsStatusLocalCopy.howLong * TO_SECONDS) ) {
                irrigationControlStatus.irrigationState =
                                            PROGRAMMED_MODE_WAITING_TO_IRRIGATE;
                irrigationControlStatus.waitedTimeMustBeReset = true;
            }
            break;

        default:
            irrigationControlInit();
            break;
    }
}

irrigationControlStatus_t irrigationControlRead()
{
    return irrigationControlStatus;
}
```

***Code 12.20 Implementation of irrigation_control.cpp (Part 2/2).***

In Code 12.21, the implementation of *irrigation_timer.h* is shown. On line 8, the defined type *irrigationTimer_t* is declared. It has two members, *waitedTime* and *irrigatedTime*, used to account for the time elapsed while waiting to irrigate and the time elapsed while irrigating. On lines 15 to 17, the public functions are declared.

In Code 12.22, the implementation of *irrigation_timer.cpp* is shown. Libraries are included on lines 3 to 8. On line 12, the defined type variable *irrigationTimer* is declared. On lines 16 to 20, *irrigationTimerInit()* is implemented. It can be seen that *irrigationTimer.waitedTime* and *irrigationTimer.irrigatedTime* are both set to 0. On lines 22 to 45, *irrigationTimerUpdate()* is implemented. The variable *irrigationControlStatusLocalCopy* is declared on line 24, and it is assigned the return value of

*irrigationControlRead()* on line 26. Line 28 assesses whether *irrigationTimer.waitedTime* must be reset by evaluating *irrigationControlStatusLocalCopy.waitedTimeMustBeReset*. Line 32 assesses whether *irrigationTimer.irrigatedTime* must be reset.

Line 36 assesses whether *irrigationTimer.waitedTime* must be incremented, which is true if *irrigationControlStatusLocalCopy.irrigationState* is equal to PROGRAMMED_MODE_WAITING_TO_IRRIGATE. Line 41 assesses whether *irrigationTimer.irrigatedTime* must be incremented, which is true if *irrigationControlStatusLocalCopy.irrigationState* is equal to PROGRAMMED_MODE_IRRIGATING. Finally, on lines 47 to 50, *irrigationTimerRead()* is implemented.

```
1  //=====[#include guards - begin]===============================================
2
3  #ifndef _IRRIGATION_TIMER_H_
4  #define _IRRIGATION_TIMER_H_
5
6  //=====[Declaration of public data types]======================================
7
8  typedef struct irrigationTimer {
9      int waitedTime;
10     int irrigatedTime;
11 } irrigationTimer_t;
12
13 //=====[Declarations (prototypes) of public functions]=========================
14
15 void irrigationTimerInit();
16 void irrigationTimerUpdate();
17 irrigationTimer_t irrigationTimerRead();
18
19 //=====[#include guards - end]=================================================
20
21 #endif // _IRRIGATION_TIMER_H_
```

***Code 12.21 Implementation of irrigation_timer.h.***

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "irrigation_timer.h"
7
8  #include "buttons.h"
9
10 //=====[Declaration and initialization of private global variables]============
11
12 static irrigationTimer_t irrigationTimer;
13
14 //=====[Implementations of public functions]===================================
15
16 void irrigationTimerInit()
17 {
18     irrigationTimer.waitedTime = 0;
19     irrigationTimer.irrigatedTime = 0;
20 }
21
22 void irrigationTimerUpdate()
23 {
24     irrigationControlStatus_t irrigationControlStatusLocalCopy;
```

```
25
26      irrigationControlStatusLocalCopy = irrigationControlRead();
27
28      if ( irrigationControlStatusLocalCopy.waitedTimeMustBeReset ) {
29          irrigationTimer.waitedTime = 0;
30      }
31
32      if ( irrigationControlStatusLocalCopy.irrigatedTimeMustBeReset ) {
33          irrigationTimer.irrigatedTime = 0;
34      }
35
36      if ( irrigationControlStatusLocalCopy.irrigationState ==
37          PROGRAMMED_MODE_WAITING_TO_IRRIGATE ) {
38          irrigationTimer.waitedTime++;
39      }
40
41      if ( irrigationControlStatusLocalCopy.irrigationState ==
42          PROGRAMMED_MODE_IRRIGATING ) {
43          irrigationTimer.irrigatedTime++;
44      }
45  }
46
47  irrigationTimer_t irrigationTimerRead()
48  {
49      return irrigationTimer;
50  }
```

***Code 12.22 Implementation of irrigation_timer.cpp.***

In Code 12.23 and Code 12.24, the new implementation of *user_interface.cpp* is shown. In Code 12.23, libraries are included on lines 3 to 11, and line 21 is modified to only write "Mode:". On line 9 of Code 12.24, the display position after "Mode:" is written (i.e., "5,0"). Then, a switch statement is used on *irrigationControlStatusLocalCopy.irrigationState* to write the corresponding text on the display.

```
1   //=====[Libraries]=============================================================
2
3   #include "mbed.h"
4   #include "arm_book_lib.h"
5
6   #include "user_interface.h"
7
8   #include "display.h"
9   #include "buttons.h"
10  #include "moisture_sensor.h"
11  #include "irrigation_control.h"
12
13  //=====[Implementations of public functions]===================================
14
15  void userInterfaceInit()
16  {
17      displayInit( DISPLAY_TYPE_LCD_HD44780,DISPLAY_CONNECTION_GPIO_4BITS );
18
19      displayClear();
20      displayCharPositionWrite( 0, 0 );
21      displayStringWrite( "Mode:" );
22      displayCharPositionWrite( 0, 1 );
23      displayStringWrite( "HowOften:  hours" );
24      displayCharPositionWrite( 0, 2 );
```

```
25      displayStringWrite( "HowLong:  seconds" );
26      displayCharPositionWrite( 0, 3 );
27      displayStringWrite( "MinMois:  %-Curr:  %" );
28  }
29
30  void userInterfaceUpdate()
31  {
32      char number[3];
33
34      buttonsStatus_t buttonsStatusLocalCopy;
35      float hl69AveragedValueLocalCopy;
36      irrigationControlStatus_t irrigationControlStatusLocalCopy;
37
38      buttonsStatusLocalCopy = buttonsRead();
39      hl69AveragedValueLocalCopy = moistureSensorRead();
40      irrigationControlStatusLocalCopy = irrigationControlRead();
41
42      displayCharPositionWrite( 9, 1 );
43      sprintf( number, "%02d", buttonsStatusLocalCopy.howOften );
44      displayStringWrite( number );
45
46      displayCharPositionWrite( 8, 2 );
47      sprintf( number, "%02d", buttonsStatusLocalCopy.howLong );
48      displayStringWrite( number );
49
```

*Code 12.23 New implementation of user_interface.cpp (Part 1/2).*

```
 1      displayCharPositionWrite( 8, 3 );
 2      sprintf( number, "%02d", buttonsStatusLocalCopy.moisture );
 3      displayStringWrite( number );
 4
 5      displayCharPositionWrite( 17, 3 );
 6      sprintf( number, "%2.0f", 100*hl69AveragedValueLocalCopy );
 7      displayStringWrite( number );
 8
 9      displayCharPositionWrite( 5, 0 );
10
11      switch( irrigationControlStatusLocalCopy.irrigationState ) {
12
13          case INITIAL_MODE_ASSESSMENT:
14              displayStringWrite( "Initializing..." );
15              break;
16
17          case CONTINUOUS_MODE_IRRIGATING:
18              displayStringWrite( "Continuous-ON  " );
19              break;
20
21          case PROGRAMMED_MODE_WAITING_TO_IRRIGATE:
22              displayStringWrite( "Programmed-Wait" );
23              break;
24
25          case PROGRAMMED_MODE_IRRIGATION_SKIPPED:
26              displayStringWrite( "Programmed-Skip" );
27              break;
28
29          case PROGRAMMED_MODE_IRRIGATING:
30              displayStringWrite( "Programmed-ON  " );
31              break;
32
33          default:
34              displayStringWrite( "Non-supported  " );
35              break;
36      }
37
38  void userInterfaceRead()
39  {
40  }
```

*Code 12.24 New implementation of user_interface.cpp (Part 2/2).*

In Code 12.25, the new implementation of *relay.cpp* is shown. Libraries are included on lines 3 to 8. Note that *buttons.h* is not included anymore. The private DigitalInOut object *relayControlPin* is declared on line 12 and assigned to pin PF_2. The function *relayInit()* is declared on line 16. This function turns off the relay by configuring *relayControlPin* as an input, so no energy is supplied to the relay.

On lines 22 to 56, the implementation of *relayUpdate()* is shown. On line 24, *irrigationControlStatusLocalCopy* is declared and assigned the return value of *irrigationControlRead()* on line 26. On line 28, there is a switch over *irrigationControlStatusLocalCopy.irrigationState*. Depending on the value of *irrigationControlStatusLocalCopy.irrigationState*, the relay is turned on or off. For example, in the irrigation state CONTINUOUS_MODE_IRRIGATING, the relay should be turned on, and, therefore, *relayControlPin* is configured as output and LOW is assigned to the pin (remember that the relay is turned on by assigning 0 V to the pin PF_2).

Finally, the function *relayRead()* is shown on lines 57 to 61.

```
1  //=====[Libraries]=============================================================
2
3  #include "mbed.h"
4  #include "arm_book_lib.h"
5
6  #include "relay.h"
7
8  #include "irrigation_control.h"
9
10 //=====[Declaration and initialization of public global objects]===============
11
12 static DigitalInOut relayControlPin(PF_2);
13
14 //=====[Implementations of public functions]===================================
15
16 void relayInit()
17 {
18     relayControlPin.mode(OpenDrain);
19     relayControlPin.input();
20 }
21
22 void relayUpdate()
23 {
24     irrigationControlStatus_t irrigationControlStatusLocalCopy;
25
26     irrigationControlStatusLocalCopy = irrigationControlRead();
27
28     switch( irrigationControlStatusLocalCopy.irrigationState ) {
29
30         case INITIAL_MODE_ASSESSMENT:
31             relayControlPin.input();
32             break;
33
34         case CONTINUOUS_MODE_IRRIGATING:
35             relayControlPin.output();
36             relayControlPin = LOW;
37             break;
38
39         case PROGRAMMED_MODE_WAITING_TO_IRRIGATE:
40             relayControlPin.input();
```

```
41              break;
42
43          case PROGRAMMED_MODE_IRRIGATION_SKIPPED:
44              relayControlPin.input();
45              break;
46
47          case PROGRAMMED_MODE_IRRIGATING:
48              relayControlPin.output();
49              relayControlPin = LOW;
50              break;
51
52          default:
53              relayControlPin.input();
54              break;
55      }
56  }
57
58  void relayRead()
59  {
60  }
```

***Code 12.25 New implementation of relay.cpp.***

## Proposed Exercise

1. Implement the behavior for the project that was selected in the "Proposed Exercise" of Example 12.1. Write all the corresponding *.h* and *.cpp* files.

## Answer to the Exercise

1. The implementation of the behavior might be a difficult task. For that reason, some tips are presented below.

**TIP:** It is strongly recommended to start by implementing a reduced set of the system functionality. In this way, the code is easier to revise and test. For example, in Code 12.26, it is shown how a first version of *irrigationControlUpdate()* could be written to include only three of the states.

```
 1  void irrigationControlUpdate()
 2  {
 3      buttonsStatus_t buttonsStatusLocalCopy;
 4
 5      buttonsStatusLocalCopy = buttonsRead();
 6
 7      switch( irrigationControlStatus.irrigationState ) {
 8
 9          case INITIAL_MODE_ASSESSMENT:
10
11              if( buttonsStatusLocalCopy.changeMode ) {
12                 irrigationControlStatus.irrigationState = CONTINUOUS_MODE_IRRIGATING;
13              } else {
14                     irrigationControlStatus.irrigationState =
15                                             PROGRAMMED_MODE_WAITING_TO_IRRIGATE;
16                  }
```

```
17              break;
18
19          case CONTINUOUS_MODE_IRRIGATING:
20
21              if( buttonsStatusLocalCopy.changeMode ) {
22                  irrigationControlStatus.irrigationState =
23                                              PROGRAMMED_MODE_WAITING_TO_IRRIGATE;
24              }
25              break;
26
27          case PROGRAMMED_MODE_WAITING_TO_IRRIGATE:
28              if( buttonsStatusLocalCopy.changeMode ) {
29                  irrigationControlStatus.irrigationState =
30                                              CONTINUOUS_MODE_IRRIGATING;
31                }
32              break;
33
34          default:
35              irrigationControlInit();
36              break;
37      }
38  }
```

*Code 12.26 Simplified implementation of irrigation_control.cpp.*

**TIP:** It is also recommended to consider showing some additional information on the user interface to have a better understanding of what is going on. For example, lines 59 to 65 of Code 12.27 are used to print on the display the values of *waitedTime* and *irrigatedTime*. By means of these lines, as well as the values of TO_SECONDS and TO_HOUR shown in Table 12.34, the behavior of the system can be tested in a shorter amount of time by having more information on hand.

```
1   void userInterfaceUpdate()
2   {
3       char number[3];
4
5       buttonsStatus_t buttonsStatusLocalCopy;
6       float hl69AveragedValueLocalCopy;
7       irrigationControlStatus_t irrigationControlStatusLocalCopy;
8       irrigationTimer_t irrigationTimerLocalCopy;
9
10      buttonsStatusLocalCopy = buttonsRead();
11      hl69AveragedValueLocalCopy = moistureSensorRead();
12      irrigationControlStatusLocalCopy = irrigationControlRead();
13      irrigationTimerLocalCopy = irrigationTimerRead();
14
15      displayCharPositionWrite( 9, 1 );
16      sprintf( number, "%02d", buttonsStatusLocalCopy.howOften );
17      displayStringWrite( number );
18
19      displayCharPositionWrite( 8, 2 );
20      sprintf( number, "%02d", buttonsStatusLocalCopy.howLong );
21      displayStringWrite( number );
22
23      displayCharPositionWrite( 8, 3 );
24      sprintf( number, "%02d", buttonsStatusLocalCopy.moisture );
25      displayStringWrite( number );
```

```
26
27      displayCharPositionWrite( 17, 3 );
28      sprintf( number, "%2.0f", 100*hl69AveragedValueLocalCopy );
29      displayStringWrite( number );
30
31      displayCharPositionWrite( 5, 0 );
32      switch( irrigationControlStatusLocalCopy.irrigationState ) {
33
34          case INITIAL_MODE_ASSESSMENT:
35              displayStringWrite( "Initializing..." );
36              break;
37
38          case CONTINUOUS_MODE_IRRIGATING:
39              displayStringWrite( "Continuous-ON  " );
40              break;
41
42          case PROGRAMMED_MODE_WAITING_TO_IRRIGATE:
43              displayStringWrite( "Programmed-Wait" );
44              break;
45
46          case PROGRAMMED_MODE_IRRIGATION_SKIPPED:
47              displayStringWrite( "Programmed-Skip" );
48              break;
49
50          case PROGRAMMED_MODE_IRRIGATING:
51              displayStringWrite( "Programmed-ON  " );
52              break;
53
54          default:
55              displayStringWrite( "Non-supported  " );
56              break;
57      }
58
59      displayCharPositionWrite( 18, 1 );
60      sprintf( number, "%02d", irrigationTimerLocalCopy.waitedTime );
61      displayStringWrite( number );
62
63      displayCharPositionWrite( 18, 2 );
64      sprintf( number, "%02d", irrigationTimerLocalCopy.irrigatedTime );
65      displayStringWrite( number );
66  }
```

***Code 12.27 Alternative version of the implementation of user_interface.cpp, where more information is shown.***

***Table 12.34 Sections in which lines were modified in irrigation_control.h.***

| Section | Lines that were added |
|---|---|
| Declaration of public defines | `#define TO_SECONDS  1`<br>`#define TO_HOURS  1` |

**WARNING:** The UART connection with the PC over USB can be used to print the values of *waitedTime* and *irrigatedTime* on the serial terminal. In this case, it should be remembered that there is a delay for each character that is sent to the PC (recall the Under the Hood section of Chapter 2), and this may alter the system behavior.

### Example 12.9: Check the System Behavior

#### Objective

Check the behavior of the system against the requirements and use cases defined in step 2.

#### Summary of the Expected Outcome

The results of this step are expected to be:

- a table listing all the requirements, indicating whether they were accomplished or not, and
- a table listing all the use cases, indicating whether the system behaves as expected or not.

#### Discussion of How to Implement this Step

In step 2 (Example 12.2), the requirements were established, as well as the use cases. In this step, the accomplishment of each requirement and use case will be analyzed. In the case of non-compliance, it will be explained why it was not possible to accomplish the requirement and what the proposed solution is.

#### Implementation of this Proposed Step

In Table 12.35, the accomplishment of the requirements is assessed. It can be seen that out of 24 requirements, only 2 were not accomplished (Req. 5.1 and 6.1). In the case of Req. 5.1, the moisture sensor that was used is not, and cannot be, calibrated. Perhaps in a future version of the system the sensor can be changed or calibrated. Regarding Req. 6.1, it was noticed that the available solenoid valves operate with 12 V, and for the sake of simplicity in this version it was decided not to use an extra circuit (e.g., a step-up DC/DC converter) to obtain 12 V out of two AA batteries.

***Table 12.35 Accomplishment of the requirements defined for the home irrigation system.***

| Req. ID | Description | Accomplished? |
|---|---|---|
| 1.1 | The system will have one water-in port based on a ½-inch connector. | ✓ |
| 1.2 | The system will control one irrigation circuit by means of a solenoid valve. | ✓ |
| 2.1 | The system will have a continuous mode in which a button will enable the water flow. | ✓ |
| 2.2 | The system will have a programmed irrigation mode based on a set of configurations: | ✓ |
| 2.2.1 | Irrigation will be enabled only if moisture is below the “Minimum moisture level” value. | ✓ |
| 2.2.2 | Irrigation will be enabled every H hours, with H being the “How often” configuration. | ✓ |
| 2.2.3 | Irrigation will be enabled for S seconds, with S being the “How long” configuration. | ✓ |
| 2.2.4 | Irrigation will be skipped if “How long” is configured to 0 (zero). | ✓ |
| 3.1 | The system configuration will be done by means of a set of buttons: | ✓ |
| 3.1.1 | The “Mode” button will change between “Programmed irrigation” and “Continuous irrigation”. | ✓ |
| 3.1.2 | The “How often” button will increase the time between irrigations in programmed mode by one hour. | ✓ |
| 3.1.3 | The “How long” button will increase the irrigation time in programmed mode by ten seconds. | ✓ |

| Req. ID | Description | Accomplished? |
|---|---|---|
| 3.1.4 | The "Moisture" button will increase the "Minimum moisture level" configuration by 5%. | ✓ |
| 3.1.5 | The maximum values are: "How often": 24 h; "How long": 90 s; "Moisture": 95%. | ✓ |
| 3.1.6 | "How long" and "Moisture" will be set to 0 (zero) if they reach the maximum and the button is pressed. | ✓ |
| 3.1.7 | "How often" will be set to 1 if it reaches its maximum value and the button is pressed. | ✓ |
| 4.1 | The system will have an LCD display: | ✓ |
| 4.1.1 | The LCD display will show the current operation mode: Continuous or Programmed. | ✓ |
| 4.1.2 | The LCD display will show the value of "How often", "How long", and "Moisture". | ✓ |
| 5.1 | The system will measure soil moisture with an accuracy better than 5%. | × |
| 6.1 | The system will be powered using two AA batteries. | × |
| 7.1 | The system should be finished one week after starting (this includes buying the parts). | ✓ |
| 8.1 | The components for the prototype should cost less than 60 USD. | ✓ |
| 9.1 | The prototype should be accompanied with a list of parts, a connection diagram, the code repository, a table indicating the accomplishment of requirements, and use cases. | ✓ |

**NOTE:** Req. 7.1 is considered accomplished based on an estimation of the time that it will take the reader to complete this project, considering the skills acquired through this book and six hours/day of work.

In Table 12.36, the set of use cases that were established in Example 12.2 are shown, as is an assessment of whether they have been accomplished. It can be seen that the three use cases were fully accomplished.

***Table 12.36 Accomplishment of the use cases defined for the home irrigation system.***

| Use case | Title | Accomplished? |
|---|---|---|
| #1 | The user wants to irrigate plants immediately for a couple of minutes. | ✓ |
| #2 | The user wants to program irrigation to take place for ten seconds every six hours. | ✓ |
| #3 | The user wants the plants not to be irrigated. | ✓ |

Considering that 92% of the requirements and 100% of the use cases were accomplished, it can be considered that the project was successfully developed.

Lastly, this helps to introduce two important concepts, which are frequently confused with each other: *verification* and *validation* [3].

**DEFINITION:** *Verification* asks the question, "Are we building the product right?" That is, does the software conform to its specifications? (As a house conforms to its blueprints.)

**DEFINITION:** *Validation* asks the question, "Are we building the right product?" That is, does the software do what the user really requires? (As a house conforms to what the owner wants.)

After these definitions, and given the accomplishment results shown above, it can be stated that:

- It was verified that the system that was developed accomplished almost all of its specifications.
- Given that users were not asked about their needs, at this point validation cannot be assessed.

It can be concluded that it is of capital importance to involve the users at an early stage of the system development process in order to be able to adjust the system specifications to their actual needs. Otherwise, a product can be built that works as expected but is not useful for the users.

### Proposed Exercise

1. Analyze the accomplishment of the requirements and use cases of the project that was selected in the "Proposed Exercise" of Example 12.1. Document them by means of tables, as has been shown in this example.

### Answer to the Exercise

1. Table 12.35 and Table 12.36 can be used as templates to list all the requirements and use cases and the corresponding assessment of their accomplishment.

**Example 12.10: Develop the Documentation of the System**

### Objective

Provide a set of documents that summarize the final outcome of the project.

### Summary of the Expected Outcome

The results of this step are expected to be:

- a set of tables, drawings, and rationales summarizing the results of the project, and
- a list of proposed steps that could be followed in order to continue and improve the project.

### Discussion of How to Implement this Step

Throughout the examples, many tables, drawings, and rationales about the project were shown. In this final step, all those documents can be presented as a final summary of the project. Additionally, based on the experience obtained during the project implementation, proposals can be made about how to continue and improve the project.

## Implementation of this Proposed Step

In Table 12.37, a set of elements that summarize the most important information about the home irrigation system design and implementation are presented. It can be seen that by analyzing this information, a developer can understand most of the project details. A user manual and complementary documents can be added to this list, depending on the specific characteristics of the project.

***Table 12.37 Elements that summarize the most important information about the home irrigation system.***

| Element | Reference |
|---|---|
| Rationale explaining why the project was selected | Example 12.1 |
| Requirements of the project | Table 12.6 |
| Use cases of the project | Table 12.7 |
| Diagram of the hardware modules of the system | Figure 12.2 |
| Connection diagram of all the hardware elements | Figure 12.3 |
| Bill of materials | Table 12.18 |
| Diagram of the software design | Figure 12.4, Figure 12.5 |
| Definition of the software modules (public functions, variables, etc.) | Table 12.19 to Table 12.32 |
| Diagram of the proposed finite-state machine | Figure 12.7 |
| Software implementation | [2] |
| Assessment of the accomplishment of the requirements | Table 12.35 |
| Assessment of the accomplishment of the use cases | Table 12.36 |
| Proposal for next steps | Example 12.10 |
| Final conclusions | Example 12.10 |

Based on the experience obtained during the project design and implementation, the following next steps can be addressed:

- A step-up module might be included in the design in order to provide the 5 V supply for the NUCLEO board and the 12 V supply for the solenoid using two AA batteries.
- A calibration process can be implemented to determine the correspondence between the output signal of the HL-69 sensor and the moisture level.

Moreover, in order to achieve a commercial version of the design, the following items can be considered:

- The development board might be replaced by a bespoke design on a printed circuit board (PCB), which should reduce the cost and size of the system.

- A case can be designed for the irrigation system in order to make it portable and usable for final users.

- The reliability of the system can be tested in order to determine if more improvements are required.

As a final conclusion for this project, it can be seen that the software design proved to be appropriate for implementing a project with a certain complexity. Therefore, the approach can be reused in future projects, such as the mobile robot that avoids obstacles or the flying drone equipped with a camera, which were mentioned in Example 12.1.

### Proposed Exercise

1. Develop a list of the relevant documentation regarding the project which was selected in the "Proposed Exercise" of Example 12.1. Document the project by means of tables, as has been shown in this example. Recommend a set of future steps for the project, and finally develop a brief conclusion.

### Answer to the Exercise

1. Table 12.37 can be used as a template to list all of the relevant documentation. The next steps and the final conclusion can be similar to the ones presented in this example.

## 12.3 Final Words

### 12.3.1 The Projects to Come

Throughout this book, many concepts have been introduced. At this point, the reader is capable of implementing a broad set of embedded systems. Moreover, the reader now has many keys that will open the doors to "projects to come." Those projects will include some technologies and techniques that were introduced and discussed in this book, and probably some other elements that are beyond the scope of this book. In any case, the reader now has a solid basis into which new knowledge can be incorporated.

For example, the HC-SR04 ultrasonic module shown in Figure 12.12 is very popular. It uses the principle of sonar (sound navigation ranging) to determine the distance to an object in the same way that bats do. It has four pins: VCC, Trig, Echo, and GND. The Trig pin is used to trigger a high-frequency sound. When the signal is reflected, the echo pin changes its state. The time between the transmission and reception of the signal allows us to calculate the distance to an object, which is useful, for example, in parking systems or in autonomous vehicles. Note that the knowledge to implement the appropriate software module to use this sensor was introduced within this book (i.e., look at the Tip on Example 10.4).


*Figure 12.12 Image of the HC-SR04 ultrasonic module.*

A microphone module, as shown in Figure 12.13, can be easily used by applying the concepts introduced in this book. The module has a digital output pin (DO) and analog output pin (AO). The digital output sends a high signal when the sound intensity reaches a certain threshold, which is adjusted using the potentiometer on the sensor. The analog output can be sampled and stored in an array using an appropriate sampling rate, in order to be reproduced later, for example in a public address system. The procedure is similar to that applied to the temperature sensor.

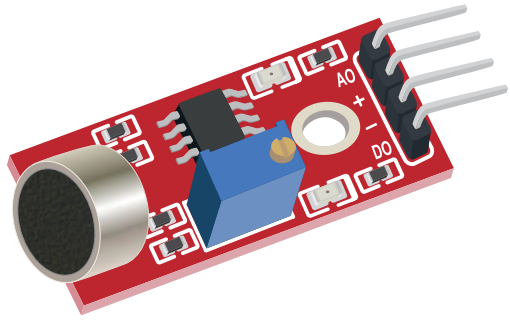



*Figure 12.13 Image of a microphone module.*

In Figure 12.14, a digital barometric pressure module is shown, used to measure the absolute pressure of the environment. By converting the pressure measures into altitude, the height of a drone can be estimated. The sensor also measures temperature and humidity and has an I2C bus interface, such as the one introduced in Chapter 6.

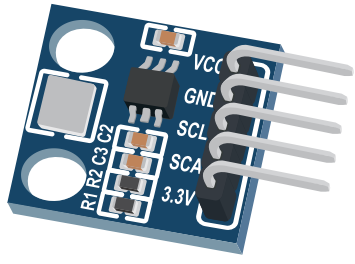



*Figure 12.14 Image of a digital barometric pressure module.*

There are many other sensor modules available, as well as many actuators that the reader is ready to use. For example, the micro servo motor shown in Figure 12.15 has three pins: GND, VCC, and Signal. The angle of the motor axis angle is controlled in the range 0 to 180° by the duty cycle of the signal delivered at the Signal pin. The reader is encouraged to explore actuators that allow tiny movements, such as stepper motors.

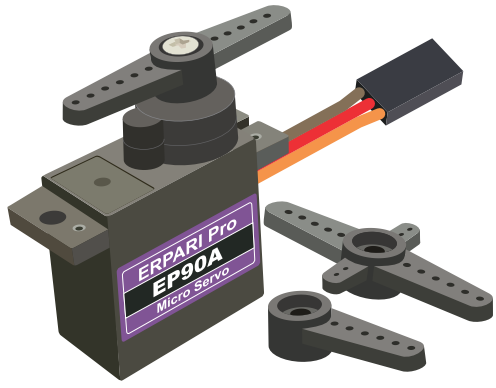



*Figure 12.15 Image of a micro servo motor.*

GPS (global positioning system) modules, such as the one shown in Figure 12.16, are also very popular. They are commanded using AT commands, in a very similar way to the ESP-01 module introduced in Chapter 11.

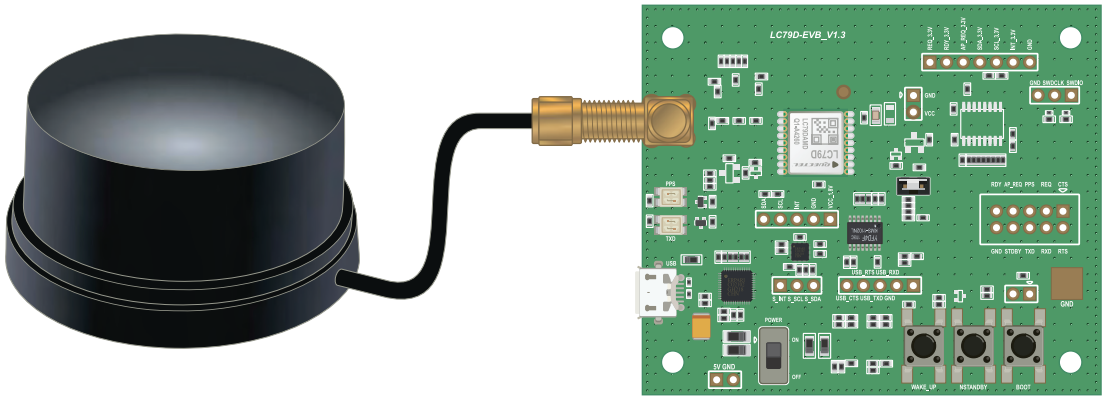



*Figure 12.16 Image of a GPS module. Note the GPS antenna on the left.*

Finally, in Figure 12.17, an NB-IoT (Narrow Band Internet of Things) cellular module is shown. These modules are also controlled using AT commands and can be used to implement communications based on 4G, 5G, LoRa, or SigFox, for example.

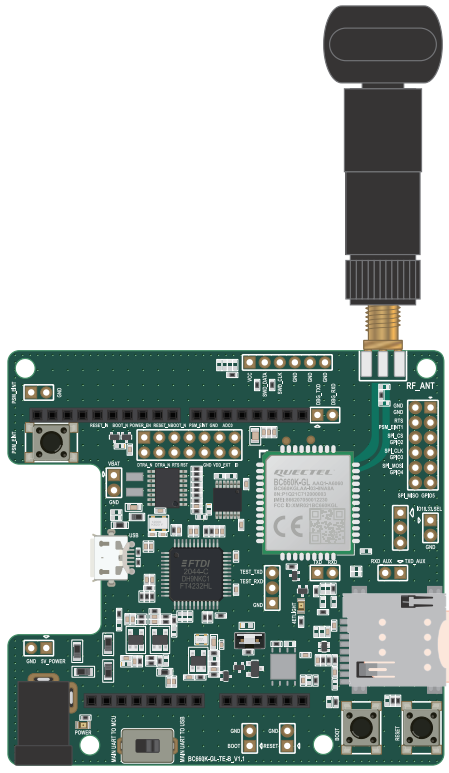


*Figure 12.17 Image of an NB-IoT cellular module. Note the SIM card slot on the right.*

Given the vast range of modules that are available, the aim of this section is just to give an overview of how most of them, maybe even all of them, can be used by applying the techniques and concepts introduced in this book.

Enjoy the projects to come!

**TIP:** Never forget that good engineering is based on following processes, keeping things as simple as possible, dividing problems into small pieces, and solving trade-off situations.

## Proposed Exercise

1. How can a micro servo motor be controlled using the techniques introduced in this book?

## Answer to the Exercise

1. As explained above, the motor axis angle is controlled by the duty cycle of the signal delivered at the Signal pin. Thus, the PWM technique that was introduced in Chapter 8 can be used.

## References

[1] "NUCLEO-F429ZI | Mbed". Accessed July 9, 2021. https://os.mbed.com/platforms/ST-Nucleo-F429ZI/#zio-and-arduino-compatible-headers

[2] "GitHub - armBookCodeExamples/Directory". Accessed July 9, 2021. https://github.com/armBookCodeExamples/Directory

[3] Pham, H. (1999). Software Reliability. John Wiley & Sons, Inc. p. 567. ISBN 9813083840.

# Glossary of Abbreviations

| | |
|---|---|
| 4G | Fourth Generation |
| A | Anode |
| AC | Alternating Current |
| ACK | Acknowledge |
| ACSE | Asociación Civil para la Investigación, Promoción y Desarrollo de Sistemas Electrónicos Embebidos, Civil Association for Research, Promotion and Development of Embedded Electronic Systems |
| ADC | Analog to Digital Converter |
| ALT | Alternative |
| AOUT, AO | Analog Output |
| API | Application Programming Interface |
| ASCII | American Standard Code For Information Interchange |
| AT | Attention |
| BLE | Bluetooth Low Energy |
| BRK | Break |
| CADIEEL | Cámara Argentina de Industrias Electrónicas, Electromecánicas y Luminotécnicas, Argentine Chamber of Electronic, Electromechanical and Lighting Industries |
| CGRAM | Custom Generated Random-Access Memory |
| CIAA | Computadora Industrial Abierta Argentina, Argentine Open Industrial Computer |
| $CO_2$ | Carbon Dioxide |
| COM | Communication |

| | |
|---|---|
| CONICET | Consejo Nacional de Investigaciones Científicas y Técnicas, National Scientific and Technical Research Council of Argentina |
| CPU | Central Processing Unit |
| CR | Carriage Return |
| CS | Chip Select |
| CSS | Cascading Style Sheets |
| DAC | Digital to Analog Converter |
| DB | Data Bus |
| DC | Direct Current |
| DDRAM | Display Data Random Access Memory |
| DL | Display Lines |
| DO-LED | Digital Output Light Emitting Diode |
| DOUT, DO | Digital Output |
| E | Enable |
| ESP | Espressif Systems |
| FAT | File Allocation Table |
| FAT32 | File Allocation Table 32 |
| FPU | Floating-Point Unit |
| FSM | Finite-State Machine |
| GB | GigaByte |
| GDRAM | Graphic Display RAM |
| GLCD | Graphical Liquid Crystal Display |
| GND | Ground |

| | |
|---|---|
| GPIO | General Purpose Input Output |
| GPS | Global Positioning System |
| HAL | Hardware Abstraction Layer |
| HDMI | High-Definition Multimedia Interface |
| HMI | Human–Machine Interface |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| HV | High Voltage |
| I/D | Increment/Decrement |
| I/O or IO | Input/Output |
| I2C | Inter Integrated Circuit |
| IC | Integrated Circuit |
| ICC | Iterative Conversion Controller |
| IDE | Integrated Development Environment |
| IEEE | Institute of Electrical and Electronics Engineers |
| IoT | Internet of Things |
| IP | Internet Protocol |
| IR | Infrared |
| ISA | Instruction Set Architecture |
| ISR | Interrupt Service Routine |
| K | Cathode |
| K&R | Kernighan & Ritchie |

| | |
|---|---|
| kbps | Kilobits per second |
| LCD | Liquid Crystal Display |
| LDR | Light-Dependent Resistor |
| LED | Light-Emitting Diode |
| LF | Line Feed |
| Li-Po | Lithium Polymer |
| LoRa | Long Range |
| LS | Limit Switch |
| LSb | Least Significant bit |
| LV | Low Voltage |
| MAC | Media Access Control |
| Mbps | Megabits per second |
| MCU | Microcontroller |
| MISO | Manager Input Subordinate Output |
| MOSI | Manager Output Subordinate Input |
| MSb | Most Significant bit |
| NB-IoT | Narrow Band Internet of Things |
| NC | Normally Closed |
| NO | Normally Opened |
| OOP | Object-Oriented Programming |
| PC | Personal Computer |
| PCM | Pulse-Code Modulation |

| | |
|---|---|
| PIR | Passive Infrared Sensor |
| PSB | Parallel/Serial Bus |
| PWM | Pulse-Width Modulation |
| RC | Resistor-Capacitor |
| RGB | Red, Green and Blue |
| RS | Register Select |
| RST | Reset |
| RTC | Real-Time Clock |
| RTOS | Real-Time Operating System |
| RW | Read/Write |
| RxD | Received Data |
| SAR | Successive Approximation Register |
| SCL | Serial Clock Line |
| SCLK | Serial Clock |
| SD | Secure Digital |
| SDA | Serial Data Line |
| SID | Serial Input Data |
| SMART | Specific, Measurable, Achievable, Relevant, and Time-bound |
| SMD | Surface-Mount Device |
| SONAR | Sound Navigation Ranging |
| SPI | Serial Peripheral Interface |
| SS | Subordinate Select |

| | |
|---|---|
| SSID | Service Set IDentifier |
| SSL | Secure Sockets Layer |
| ST | STMicroelectronics |
| STAMAC | Station MAC |
| TCP | Transmission Control Protocol |
| Trimpot | Trimmer Potentiometer |
| TxD | Transmitted Data |
| UART | Universal Asynchronous Receiver Transmitter |
| UBA | Universidad de Buenos Aires, University of Buenos Aires |
| UDP | User Datagram Protocol |
| UNLa | Universidad Nacional de Lanús, National University of Lanus |
| UNQ | Universidad Nacional de Quilmes, National University of Quilmes |
| URL | Uniform Resource Locator |
| USB | Universal Serial Bus |
| USD | United States Dollar |
| UTC | Universal Time Coordinated |
| UTN-FRBB | Universidad Tecnológica Nacional – Facultad Regional Bahía Blanca, National Technological University – Bahía Blanca Regional School |
| UTN-FRP | Universidad Tecnológica Nacional – Facultad Regional Paraná, National Technological University Paraná Regional School |
| VCC | Voltage Common Collector |

# Index

## The Arm Education Media Story

Did you know that Arm processor design is at the heart of technology that touches 70% of the world's population - from sensors to smartphones  to super computers.

Given the vast reach of Arm's computer chip and software designs, our aim at Arm Education Media is to play a leading role in addressing the electronics and computing skills gap; i.e., the disconnect between what engineering students are taught and the skills they need in today's job market.

Launched in October 2016, Arm Education Media is the culmination of several years of collaboration with thousands of educational institutions, industrial partners, students, recruiters and managers worldwide. We complement other initiatives and programs at Arm, including the Arm University Program, which provides university academics worldwide with free teaching materials and technologies.

Via our subscription-based digital content hub, we offer interactive online courses and textbooks that enable academics and students to keep up with the latest Arm technologies.

We strive to serve academia and the developer community at large with low-cost, engaging educational materials, tools and platforms.

# We are Arm Education Media: Unleashing Potential

# Arm Education Media Online Courses

Our online courses have been developed to help students learn about state of the art technologies from the Arm partner ecosystem. Each online course contains 10-14 modules, and each module comprises lecture slides with notes, interactive quizzes, hands-on labs and lab solutions.

The courses will give your students an understanding of Arm architecture and the principles of software and hardware system design on Arm-based platforms, skills essential for today's computer engineering workplace.

For more information, visit www.arm.com/education

**Available Now:**


Professional Certificate in Embedded Systems Essentials with Arm (on the edX platform)


Efficient Embedded Systems Design and Programming


Rapid Embedded Systems Design and Programming


Internet of Things


Graphics and Mobile Gaming


Real-Time Operating Systems Design and Programming


Introduction to System-on-Chip Design


Advanced System-on-Chip Design


Embedded Linux


Mechatronics and Robotics

# Arm Education Media Books

The Arm Education books program aims to take learners from foundational knowledge and skills covered by its textbooks to expert-level mastery of Arm-based technologies through its reference books. Textbooks are suitable for classroom adoption in Electrical Engineering, Computer Engineering and related areas. Reference books are suitable for graduate students, researchers, aspiring and practising engineers.

For more information, visit www.arm.com/education

**Available now, in print and ePub formats:**

Embedded Systems Fundamentals with Arm Cortex-M based Microcontrollers:
A Practical Approach, FRDM-KL25Z EDITION
by Dr Alexander G. Dean
ISBN 978-1-911531-03-6

Embedded Systems Fundamentals with Arm Cortex-M based Microcontrollers:
A Practical Approach, NUCLEO-F09IRC EDITION
by Dr Alexander G. Dean
ISBN 978-1-911531-26-5

Digital Signal Processing using Arm Cortex-M based Microcontrollers: Theory and Practice
by Cem Ünsalan, M. Erkin Yücel and H. Deniz Gürhan
ISBN 978-1911531-16-6

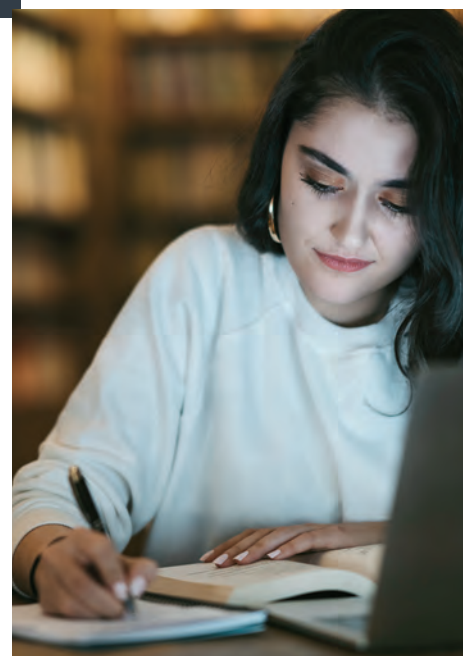
Operating Systems Foundations with Linux on the Raspberry Pi
by Wim Vanderbauwhede and Jeremy Singer
ISBN 978-1-911531-20-3

Fundamentals of System-on-Chip Design on Arm Cortex-M Microcontrollers
by René Beuchat, Florian Depraz, Sahand Kashani and Andrea Guerrieri
ISBN 978-1-911531-33-3

Modern System-on-Chip Design on Arm
by David J. Greaves
ISBN 978-1-911531-36-4

System-on-Chip with Arm Cortex-M Processors
by Joseph Yiu, Distinguished Engineer at Arm
ISBN 978-1-911531-19-7

Arm Helium Technology
M-Profile Vector Extension (MVE) for Arm Cortex-M Processors
by Jon Marsh
ISBN: 978-1-911531-23-4



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

This textbook is the perfect introduction for the beginner looking to enter the exciting world of embedded devices and IoT. Over the course of twelve chapters, readers will gain the practical skills needed to build a fully functional smart home system featuring a fire alarm, motion detector and security sensor. No prior knowledge of programming or electronics is assumed as the authors have adopted a "learn-by-doing" approach. Basic ideas are explained and then demonstrated by means of examples that progressively introduce the fundamental concepts, techniques, and tools of embedded system design. All exercises are based on the **ST Nucleo-F429ZI** board, so readers can gain experience in implementing these key concepts on an industry-relevant Arm-based microcontroller.

For educators looking to adopt this textbook, the authors have conveniently organized the book to align with a typical twelve-week semester, the idea being that one chapter can be addressed each week. This textbook also takes a blended learning approach with a set of pre-lesson activities for the students which are designed to develop the reader's curiosity and enthusiasm for embedded system design.

## Contents




**Ariel Lutenberg** is currently a Professor at the School of Engineering, and Director of the master's degrees on IoT and Embedded Artificial Intelligence at the University of Buenos Aires. He is also Researcher at the National Council of Scientific and Technical Research. He established and led the Proyecto CIAA (Argentine Open Industrial Computer), where Argentinian universities, companies and institutions worked together on developing embedded computers, covering both hardware and software.

**Pablo Gomez** is a full-time Researcher with the School of Engineering at the University of Buenos Aires, having received his doctorate in 2015. As well as directing the master's programs on Embedded Systems, he is Editor and Contributor of the 'Acoustics and Audio' section of *Elektron Journal*, published by the School of Engineering at the University of Buenos Aires.

**Eric Pernia** is currently a Research Professor with the Science and Technology department at the National University of Quilmes and a Field Application Engineer at Quectel Wireless Solutions. He has broad experience in hardware, software and firmware development, and has contributed to many open-source projects on GitHub.



ISBN 978-1-911531-42-5

arm Education Media

**Arm Education Media is a publishing operation within Arm Ltd, providing a range of educational materials for aspiring and practicing engineers. For more information, visit: arm.com/resources/education**