LAB N° 5

Application

Lab Objectives

- Use the knowledge previously seen on freertos services
- Using GPIO
- Using Interrupt

5.1 Specification of the application

We want to implement the functional structure depicted in figure 5.1.

5.1.1 Scan task

The task is activated on *ScanH* event. It acquires the value of the *Value* variable, compares it to high and low thresholds. *Value* being a theoretical output of a Digital/Analog converter not available for this Lab. It is possible to simulate its evolution by a random generation made by *ScanH* when activated.
The *rand()* function allows you to generate a number between 0 and RAND_MAX. So you just need, by rule of three, to bring the result between 0 and 100 and to set the low threshold at 15 and the threshold at 85.
If one of the thresholds is exceeded, a message of *High alarm* or *Low alarm* type is sent to the *Mess* queue, accompanied by the threshold value of *Value* variable. Otherwise, the value of *Value* variable is assigned to *SampleValue*.

```
static const uint32_t VALUE_MAX = 100;
static const uint32_t LOW_TRIG = 15;
static const uint32_t HIGH_TRIG = 85;
```

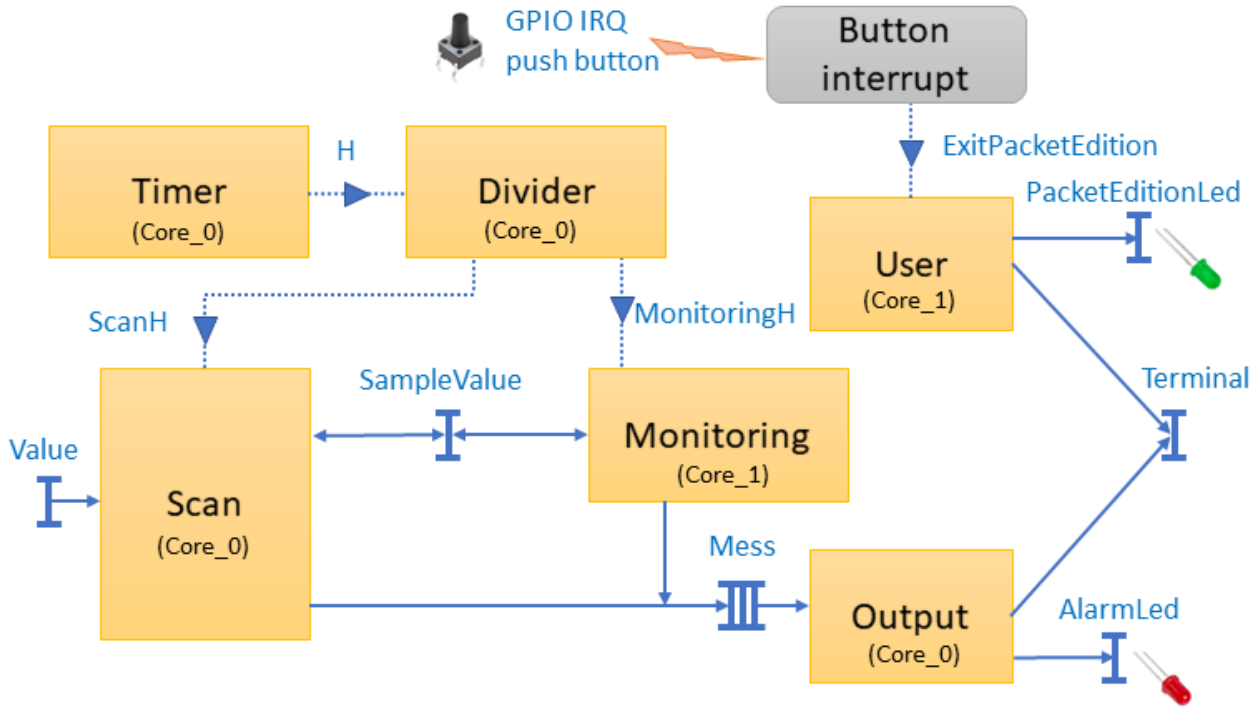

Figure 5.1 – *Functional description.*

```
int iValue = rand() / (RAND_MAX / VALUE_MAX);
if (iValue < LOW_TRIG) {
    // Low Alarm
}
else if (iValue > HIGH_TRIG) {
    // High Alarm
}
else {
    // No Alarm
}
```

5.1.2 Monitoring task

The task is activated on *MonitoringH* event. It sends messages of *Monitoring* type in the *Mess* queue. These messages contain the value of the variable *SampleValue*. Thus, the structure of *Mess* queue can be described as follows :

```
typedef enum typeInfo { Alarm, Monitoring };
typedef struct
{
  enum typeInfo xInfo;
  int xValue;
} typeMessage;
```

5.1.3 Timer task

The *Timer* task generates a periodic *H* event of 100ms and the compute execution time is 10ms Use *COMPUTE_IN_TIME_MS()* macro to simulate the compute execution time.

5.1.4 Divider task

The *Divider* task performs a division of *H* event by 5 to generate *ScanH* event and a division by 18 to generate *MonitoringH* event.

5.1.5 User task

The task is activated by the keyboard. It allows the user to print a packet of characters on the terminal (called *Packet Edition*). These characters are edited by packet, the end of a packet corresponding to the character @ (for example, a packet is « `it is an example@` » as depicted Listing 5.1). When entering a character packet (*Packet Edition* mode), the *Output* task should not display its messages on the terminal and the *PacketEditionLed* is set to 1 (the green led is ON). After entering the character @ (end of *Packet Edition* mode), it can display as many messages as necessary, until the user enters the next character. The characters can be entered using the *getch()* function. An example of use is depicted below :

```
/* Scan keyboard every 50 ms */
while (car != '@') {
  car = getch();
  if (car != 0xff) {
    printf("%c", car);
    }
  vTaskDelay(pdMS_TO_TICKS(50));
}
```

Another solution for exiting *Packet Edition* mode is to press the push button that triggers an interrupt (named *Button interrupt*). This interrupt sends an event (semaphore) to the *User* task so that it can exit *Packet Edition* mode.

5.1.6 Output task

The task displays a formatted message of the *Mess* queue as depicted in the terminal in Listing 5.1. When the message is an alarm, the *AlarmLed* must be set to 1 (the red led is ON).

Listing 5.1 – *Console example of the application*

```
13:0> Monitoring: Value = 69
203:0>  Monitoring: Value = 55
393:0>  Monitoring: Value = 20
583:0>  Monitoring: Value = 84
613:0>  Alarm: Value = 6
733:0>  Alarm: Value = 87
```

```
773:0>  Monitoring: Value = 85
853:0>  Alarm: Value = 98
913:0>  Alarm: Value = 8
963:0>  Monitoring: Value = 15
973:0>  Alarm: Value = 90
1033:0> Alarm: Value = 10
1093:0> Alarm: Value = 92
1153:0> Monitoring: Value = 50
1213:0> Alarm: Value = 87
1343:0> Monitoring: Value = 69

User mode
hello !@
End User mode
2393:0> Alarm: Value = 3
2393:0> Monitoring: Value = 15
2393:0> Alarm: Value = 9
```

5.2 Implementation of the application (Lab5)

1. Create the « lab5_application » lab from « esp32-vscode-project-template » GitHub repository.
2. Copy the provided « my_helper_fct.h » file to the « main » folder.
3. Overwrite the « main.c » file by the provided code of the « lab5_main.c » file.
4. Create a « sdkconfig.defaults » file with right parameters.
5. Write the program for *Timer*, *Divider*, *Scan* and *Monitoring* tasks. The capacity of *Mess* queue is 5. The tasks are mapped on different cores referenced in the figure 5.1.
6. Build to check that is no compilation error.

In order to validate the behavior of the application and to clearly highlight the management of *Mess* queue, we can proceed as below :

1. Validate the functional structure with only the *Timer*, *Divider*, *Scan* and *Monitoring* tasks. Check that after a number of message submissions corresponding to the maximum capacity of the *Mess* queue, the *Monitoring* and *Scan* tasks are blocked.
2. Wire the 2 leds on the board.
3. Add the *Output* task. We will check that the *Monitoring* and *Scan* tasks are no longer blocked. Only the *Output* task can be blocked on an empty *Mess* queue. Check the behavior of the red led.
4. Add the *User* task without using interrupt. Note that the *getch()* function also uses the terminal as the *printf()* function. You must protect the simultaneous write on the terminal. Check the application and green led behaviors.
5. Wire the push button on the board.

6. Add the *button interrupt* function and check the behavior. The name of the *give()* function is different when called from an interrupt (Web help).