Laboratory 3: Radio Frequency Identification (RFID) and software scheduling

Introduction

This laboratory aims to equip you with knowledge on how to:

- Read the Unique Identifier **UID** of RFID contactless smart card (a blue tag or a card) and utilise the UID for identification and access control.
- Interface to the Grove RGB backlight liquid crystal display (LCD).
- Utilise a basic task scheduler.

Equipment and resources

Hardware:

- ESP8266 Aston IoT hardware platform.
- USB A to micro B cable.
- An RFID PICC/tag.

Software & documentation:

- Arduino Integrated Development Environment (IDE).

Task 1: Interfacing with the VMA-405 RFID reader and reading Mifare card unique identifiers (UIDs)

This tasks aims to give you experience of reading RFID UIDs for the applications of identification and access control. A tags UID can be used to determine if access should be given to a system, with possession of the physical RFID granting the user access, due to the (hopefully) unique UID number.

1. Create a new sketch and copy the code from Figure 1 to use as a starting point.
   - This code will read the unique identifier of a RFID presented to the tag reader (the white printed circuit board located on the bottom left of the Aston IoT application trainer).
   - Two libraries are required. The **RFID** library handles the reading of RFID tags via communication to the VMA-405 module. Communication occurs via the Serial Peripheral Interface (SPI), hence the SPI library must also be included.
   - An instance of the RFID class is instantiated via `RFID myrfid(SS_PIN,RST_PIN);`. myrfid is the name of our class instance. The class constructor (a function which configures the class on creation) requires knowledge of the reset and slave select pin mapping, SS_PIN and RST_PIN respectively.
   - In the **setup()** function, the library and the board are initialised via calling the **RFID.init()** class function/method.
   - In the **loop()** function, we first check to see if a card is present by calling the **RFID.iscard()** class method/function. It returns true (≠0) on a card being present and false (0) on card not being present. However, the library utilised will **continue** reading UIDs until the card is **removed**.

     Hence a simple **rising-edge detector** (similar to the code you created to de-bounce the switch) is used to detect when a card is presented (i.e. the transition of the variable **cardpresent** from a 0 to a 1). To store the previous value of the **cardpresent** variable from the previous loop iteration, we set **lastcardpresent** to **cardpresent** at the end of the loop.
   - The **RFID.readCardSerial()** class method/function can then be used to read the UID (serial number). If the function returns with a UID being successfully read (returning true), a for loop is used to read out all four UID bytes over serial.
   - The four UID bytes are stored in a public array named **serNum** contained within the RFID class. Public variables within a class can be accessed using . operator after the class instances name.
   - The **Serial.printf**() class function is now utilised to make formatting of serial data easier and neater. In a similar fashion to **sprintf** used in the previous laboratory. **%3d** is a **format specifier** and will be replaced with the value of **serNum[i]**. The **d** specifies that the number should be represented as an integer decimal number. The number 3 specifies that three digits will be utilised in the resultant printed string.
   - The final call to **Serial.printf()** method prints a newline character to the serial monitor. A new line is represented by **\n**.
   - The **RFID.halt()** class function/method must be called before attempting to read another RFID tag. Hence, the method is called at the end of the **loop()** function.

```
#include <RFID.h>   // include the RFID library
#include <SPI.h>    // include the SPI library. SPI is used to communicate to RFID module
#define SS_PIN 0    // the SPI slave select pin
#define RST_PIN 2   // the pin used to reset the RFID module
int cardpresent = 0, lastcardpresent = 0; // variables used to detect a card presented
RFID myrfid(SS_PIN,RST_PIN);  // construct a class of the RFID type. The constructor
                              // of the class requires hardware specific info
void setup()                  // the setup function is called once on startup
{
  Serial.begin(115200);       // initialise serial, for communication with the host pc
  while (!Serial)             // while the serial port is not ready
  {
    ;                         // do nothing
  }
  SPI.begin();                // initialise SPI
  myrfid.init();              // initilialise the RFID module
}

void loop()                   // the loop function is called repeatedly
{
  int i;                      // a variable used for a for loop
  cardpresent = myrfid.isCard();
  if( cardpresent && !lastcardpresent )         // if a card is NEWLY present
  {
        if( myrfid.readCardSerial() )   // if we can read the serial/UID
        {
            for ( i = 0 ; i < 4 ; i++ ) // for all four UID bytes
            {
              Serial.printf("%3d ",myrfid.serNum[i]);   // print the UID byte at index i
            }
            Serial.printf("\n");                        // print a new line
        }
  }
  lastcardpresent = cardpresent;  // update the lastcard present
  myrfid.halt();                  // halt any processing of tag data
}
```

Figure 1: Code to read the UID of a MiFare classic RFID tag and display it over serial

2. Compile your sketch, debug any errors and upload the sketch to the Aston IOT hardware platform.
3. Start the Arduino's serial monitor (🔍) and place the supplied RFID tag on the reader. Make a note of the four-byte UID number displayed on the serial monitor.
4. Modify the sketch such that after reading the UID, a comparison is made from the UID from step 3. If the UID matches, print a message to the serial monitor stating access granted, otherwise print a message stating that access is not granted.
   – Code to make this comparison should be placed within the if (myrfid.readCardSerial()) code block.
   – The comparison can be achieved by using an if structure to compare each byte read from the tag with the byte read from step 3. Remember that **&&** can be used to perform a **logical AND**.
5. Compile your sketch, debug any errors and upload the sketch to the Aston IOT hardware platform.
6. Verify that your sketch only allows access for your RFID tag and not that of your neighbours.

Task 2: Writing text to the Grove RGB LCD display

The Grove RGB LCD consists of two major components:

- A LCD controller that receives character data and control information from the ESP8266. The display can be thought of as an array of 16x2 alphanumeric characters.
- A RGB backlight controller. This receives information from the ESP8266 and controls the average current drawn by red, green and blue LEDs that form a backlight for the LCD display such that the backlight colour can be controlled by additive mixing of the colours. For example, if the red, green and blue LEDs were at full brightness, the display would appear white.

This is achieved via an I^{2}C (Inter-Integrated Circuit) communication bus, a two wire serial communication protocol that can support many devices with only two wires! This is achieved by an addressing system to select a device which the master (ESP8266) wishes to communicate with. Subsequent to the address, data can be communicated to that particular device.

1. Make a copy of the sketch used for task 1 and use it as a starting point for task 2.
2. Include the library **rgb_lcd.h**
   - To include a library, use the **#include** directive.
3. Create three **integer** global variables, **red**, **green** and **blue** to store the initial intensities of the red, green and blue LEDs.
   - The intensity of each LED is controlled by a number in the range of 0 to 255. 0 = off and 255 full intensity.
   - Initialise the three variables to a colour of your choice. Refer to Table 1 for further guidance.

Table 1: RGB values for various colours

Colour	Red	Green	Blue
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Yellow	255	255	0
Cyan	0	255	255
Magenta	255	0	255
White	255	255	255
No backlight	0	0	0

4. Create an instance of the RGB_lcd class as of `rgb_lcd LCD;` .
   - This should be placed above your setup() function.
5. To initialise the RGB LCD, two class function/method calls are required in the **setup()** function of your sketch:
   - `LCD.begin(16, 2);`
   - `LCD.setRGB(red, green, blue);`
6. The LCD can now be written to using the following class methods/functions:
   - `LCD.setCursor(0, 0);`

     The first argument controls the **X** position of the cursor and the second argument the **Y** position. Remember that the display only contains 2 lines with 16 characters each. **Subsequent prints will begin from the cursor position and any characters currently displayed at those positions will be overwritten.**
   - `LCD.clear();`

     Will clear the entire contents of the display. The backlight is controlled by a separate function.

- `LCD.setRGB(red, green, blue);`

  Will set the intensity of the red, green and blue backlight LEDs respectively. Backlight control is independent of text currently displayed as the backlight is controlled by an entirely different integrated circuit!
- `LCD.print("A message");`
  `LCD.printf("A message");`

  The print and printf class functions/methods behave in exactly the same manner as those in the Serial class.

7. Modify the sketch such that on access being granted, the display shows **your username** and turns **green for 5 seconds, before returning to your choice of initial colour**.
   – Software delays are acceptable here, so using the delay function is fine!
8. If access is not granted, turn the display **red** and display a message stating **not granted for 5 seconds before returning to your choice of initial colour**.
9. Count and display the number of failed access attempts on the second line of the display.

SHOW A LECTURER/DEMONSTRATOR TO BE SIGNED OFF.

Task 3: Time driven scheduler

From laboratory 2 you should have learned that blocking code (for example, adding the while(1) loop or using a loop to wait until a button is pressed e.g. **while(buttonnotpressed);**) is incompatible with many communication libraries and embedded software techniques in general.

The various functions that handle communications need to be called periodically (this is hidden from our view by the Arduino IDE) and waiting for such an event stops the the periodic execution of such code (refer to Figure 2). In laboratory 2 it actually caused the microcontroller to reset due to the Watch Dog Timer (**WDT**) activating as it was not periodically reset, refer to Figure 2.

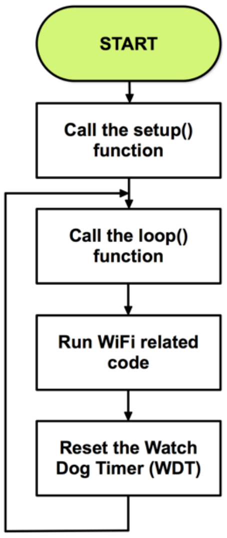

Figure 2: Simplified ESP8266 Arduino program flow

This poses a problem for embedded software engineers, how can we create code that concurrently:

- checks for button presses,
- updates a display,
- flashes LEDs,
- reads sensors,
- communicates over a network,
- controls an output actuator of some type, etc.

On a single core/thread microcontroller, you cannot actually achieve this. What actually happens is that tasks are executed sequentially fast enough such that to the end user, they **appear** to be performed concurrently.

A real time operating system (**RTOS**) and its **task scheduler** is typically used for this task. However, this task will look at a simple task scheduling approach commonly known as the **timer driven scheduler**. It allows for tasks to be scheduled to occur at regular intervals and to check for events occurring in a non-blocking manner. **You should be comfortable with using this approach as it is almost one of the simplest approaches to task scheduling approaches possible!**

The timer driven scheduler is:

- In the loop function we are going to read the current time elapsed since the code starting using the **millis()** function. Refer to this function here: https://www.arduino.cc/reference/en/language/functions/time/millis/
- Each task is given a variable. This is used to store the last time the task/function was executed.
- We can now perform a time out check from the last time the function was called to the current time. If the current time exceeds the **difference** in the current time and the last time we performed the task, we can perform the task/function and update the variable with the current time (the last time the function was called).

1. Create a new sketch.
2. Declare a global integer variable **timespressed** and initialise the variable to 0.
   - Global variables are variables declared outside of any function and should be declared above the **setup()** function.
3. Create a function **void task1 (void)** to increment the global variable **timespressed** every time the button is pressed.
   - The function task1 should call the **checkbutton()** function you created in laboratory 1 task 4.
   - Remember to copy your function code and the prototype for the **checkbutton**() function.
4. Create a function called **void task2 (void)** to toggle the state of a LED on every call.
   - The function should toggle (i.e. high to low and low to high) the state of the LED on every call.
   - This can be achieved by reading the current state of the pin or by using a variable to store the current state.
5. Create a function called **void task3 (void)** to update the LCD display with the current value of the button counting variable **timespressed**.
6. Referring to the **lecture on RFID and scheduling (Lecture 4)**, create a timer driven scheduler that:
   - Calls the **task1()** function every 5ms to check if the button has been pressed and increment the variable timespressed if the button has been pressed.
   - Calls the **task2()** function to flash the LED connected to pin 15 at **1 Hz**. Note that a **frequency** is given and not the time between cyclic executive calls.
   - Calls the **task3()** function to update the display with the value of **timespressed** every second.
7. Compile your sketch, debug any errors and upload the sketch to the Aston IOT hardware platform.
8. Verify that your cyclic executive performs as expected.

Task 4: Doing more with the scheduler

Sensor data is often noisy and the average sensor value over a given time is required.

1. Create a copy of your sketch from task 3 to base this task on.
2. In the **task1()** function, increment another global variable called **total** when the button is pressed.
3. Create another function **task4()** that prints the average button press rate in **Hz** over 10 seconds to the 2nd line of the LCD.
4. Add code to your scheduler such that task4() is called every 10 seconds.