EENG34030 Embedded and Real-time System

Part 1 Coursework

Group 35: Hongbing Qiu (mk20661), Yumu Xie (po21744) .

1 Introduction

An embedded system was created in this assignment to verify 16 – *digit identification numbers* found on payment cards, social security IDs, survey codes, and similar documents. The validation process utilizes *LuhnAlgorithm* to verify numbers. Except for a good implementation of *LuhnAlgorithm* in the embedded system, a good communication process between the PC (terminal) and chip (MSP430FR2433) is also needed.

2 Luhn Algorithm

LuhnAlgorithm requires a series of numbers from the user to validate, for example, 16 – *digit identification numbers* found on the payment card, etc. in this assignment. It will process numbers and eventually come out with a check digit. If the check digit is 0 after the check digit mods with 10 (check digit mod 10), the numbers are valid. Otherwise, the numbers are invalid. Our *LuhnAlgorithm* will first check the type of input. If the input is a valid integer which ranges from 0 to 9, the following process will be executed to check step by step. If not, the check digit *sum* will be assigned to 1 directly. If the input is a valid integer, the algorithm will check the index of the input. If the index is an even number, a *tmp* will be created, take the corresponding element and multiply the element with 2. After that, if *tmp* is greater or equal to 10, *tmp* will be minus 9 because each individual digit in the element which has 2 digits will be added together according to *LuhnAlgorithm*. Then, *tmp* will be added into a check digit *sum*. The *sum* will finally mod with 10 to check whether the numbers are valid or not.

```
// Luhn Algorithm Pseudocode
char userInput[16];
int sum = 0;
for (each index j in userInput[16]) { // from 0 to 15 / from 15 to 0
    int data = convert character into integer; // userInput[j]
    if (data is integer) { // userInput[j] is integer
        if (index j is even) {
            int tmp = data * 2;
            if (tmp >= 10) { // element which has 2 digits
                sum += (tmp - 9); // each individual digit will be added together
            } else { // element which has 1 digit only
                sum += tmp; // do nothing
            }
        } else { // index j is odd
            sum += data;
        }
    } else { // data (userInput[j]) is not integer
        sum = 1;
    }
}
if ((sum % 10) == 0) {
    return valid;
} else { // (sum % 10) != 0
    return invalid;
}
```

Fig. 1: Luhn Algorithm Pseudocode

3 Design Approach

3.1 MSP430FR2433 Overall Design

The overall design of the assignment's implementation is based on *UARTSampleCode.c* provided by Dr Roshan Weerasekera. The initial idea is to initialise UART communication between MSP430FR2433 and PC. A *LuhnAlgorithm* is also implemented in codes. Counter *index* was created for counting the number of inputs to make sure that there is a 16-digit input. *charList*[16] is used for filling the input. The result of validation is shown by *PASS* or *Fail*. A simple delay mechanism was utilised by *i* and *delay*. The codes are designed to be 3 parts: luhnAlgorithm() helper function, main() function and interrupt. The figure below (see figure 2) shows the basic design idea of overall embedded systems including PC software level and MSP430 hardware level.

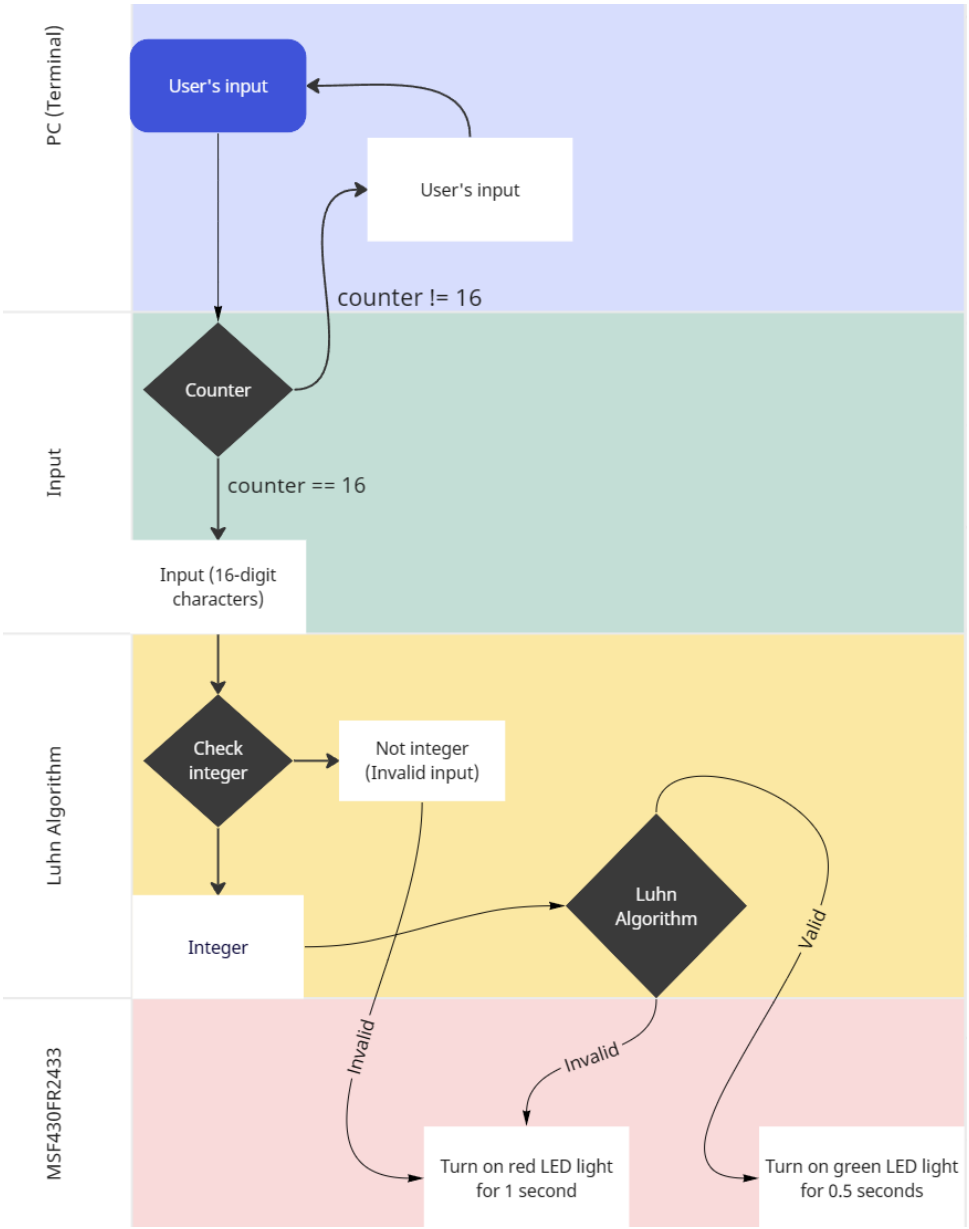

Fig. 2: Flow chart of overall embedded system

3.1.1 LuhnAlgorithm Design

In *LuhnAlgorithm* Design, achieving *LuhnAlgorithm* and flashing the correct colour of LED light is the main functionality. The diagram below (see Figure 3) displays the general design of *LuhnAlgorithm*. To be more specific, in this function, input has been checked to see whether the input is integer or not. After that, if the even index of input has been detected, the corresponding element will be multiplied by 2. If the input has 2 digits, according to *LuhnAlgorithm*, each single digit will be added together. Finally, all elements will be added together to form a sum and the sum will be modded with 10. If the sum is 0, numbers are valid, or vice versa.

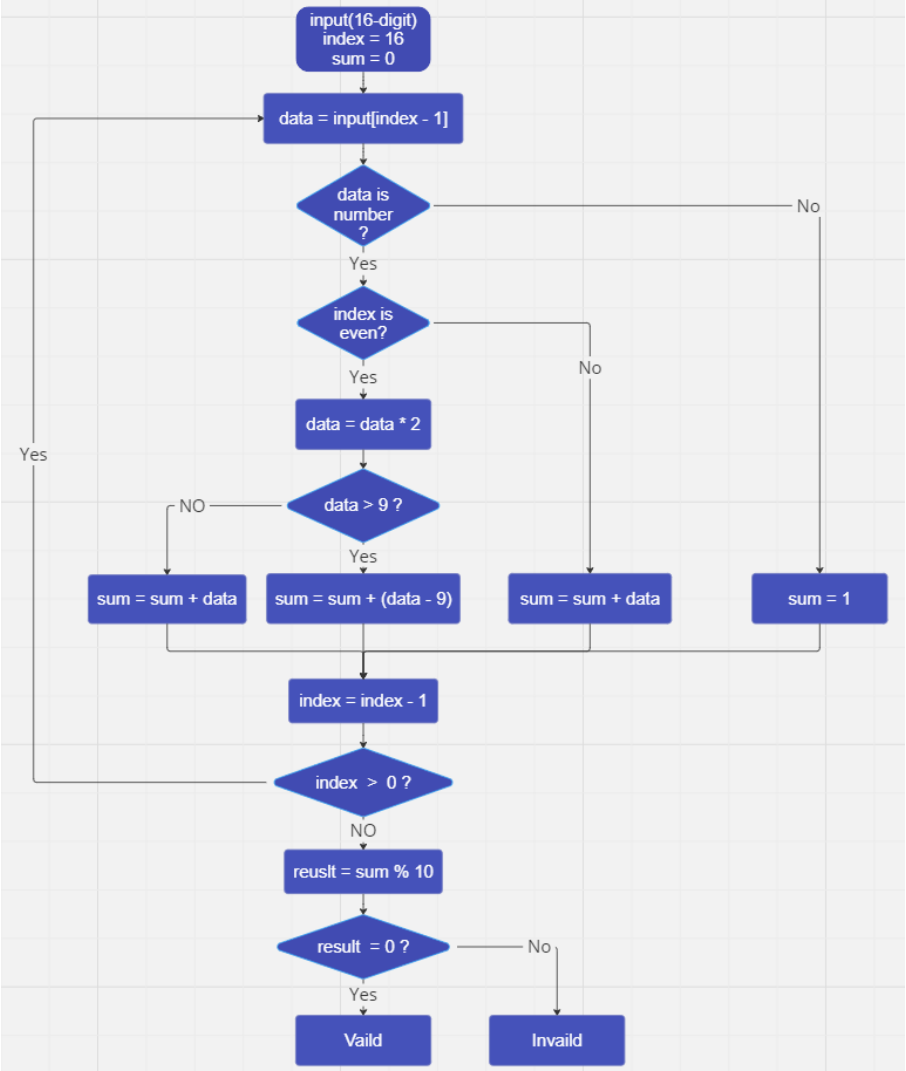

Fig. 3: Flow chart of Luhn Algorithm in codes

3.1.2 Initialisation Design

The main() function is mainly used to stop the watchdog timer, disable the GPIO power-on default high-impedance mode, initialize LED light ports, configure UART pins, calculate the Baud rate, and activate interrupt.

3.1.3 Interrupt Design

The essential functionality of this function is to echo the user's input in the terminal and update the global variable *charList*[16] by using counter *index*. There is a simple if statement to detect user inputs

backspace or not. If the user inputs *Backspace*, the index will be moved to the previous one, and there is a space applied to display empty in the terminal. Besides, an if statement has been applied to ensure that the index would only move to the minimum index which is 0. If the if statement does not apply in codes, an overflow issue will happen (out of range). Then, if the user inputs a new character, the new character will override the old character in $charList[index]$. Once the counter *index* reaches 16, *LuhnAlgorithm* will be executed to process numbers and *index* will be reset. The figure below (see Figure 4) shows the flow chart of the interrupt.

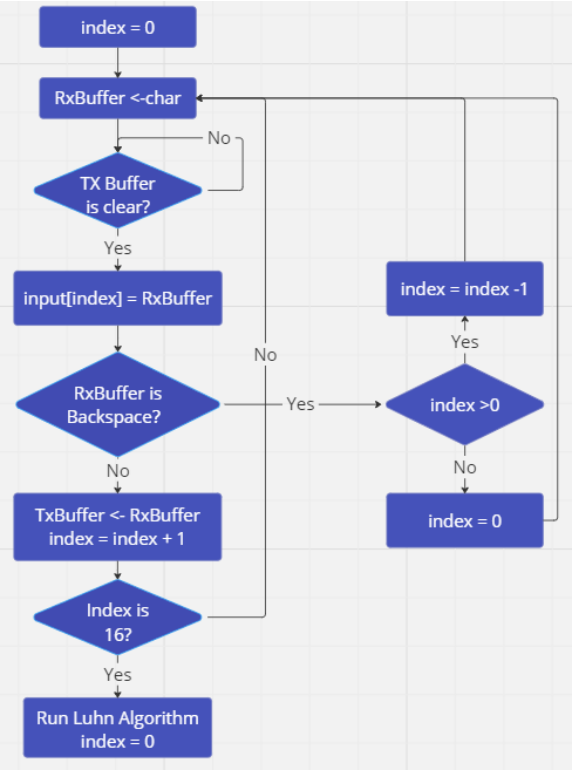

Fig. 4: Flow chart of Interrupt

4 Implementation

4.1 MSP430FR2433 Overall Implementation

The general implementation of the assignment is to design an embedded system in which the user can input a $16-digit\ identification\ numbers$ to verify whether it is valid or not. Once the numbers are valid, the green LED light will be turned on for 0.5 seconds. Once the numbers are invalid, a red LED light will be turned on for 1 second. In this assignment, echo what user's input, line feed, carriage return and override old input are also implemented. In the terminal, the user can input $16-digit\ identification\ numbers$ directly, and then $PASS$ or $Fail$ will be shown on the screen of the terminal followed by a line feed and a carriage return. If the user inputs a wrong character, a *Backspace* can be used to override the old element. After that, the embedded system will wait for the user to input the next $16-digit\ identification\ numbers$.

4.1.1 Initialisation

The below diagram (see Figure 5) shows the initialisation part of this system. The initialisation is also important for embedded systems. So, there are need to set and open a port (which is needed in this implementation) in the main function first. The below C-code figure (see Figure below) shows how to set port and UART. The first two lines are to stop the watchdog (which makes sure that prevents the

watchdog from resetting the board) and set up the msp430 board to use low power consumption. Next, the $P1DIR$ and $P1OUT$ are made use of setting red LED light (Port 1.0) and green LED light (Port 1.1). Then, the code is to configure the URAT. The focus is changed to $UCA0CTLW0$ because initialising UART pins and eUSCI are the key tasks to achieve. The baud rate is calculated to be 115200 baud to reach the requirements of the Baud rate for connecting PC. In the end, *__bis_SR_register* is used to enter LPMO CPU off, and a monitor to check whether the requirement of achieving interrupt is achieved or not. In this case, the interrupt will be triggered by terminal input.

```
63 int main(void)
64 {
65     WDTCTL = WDTPW | WDTHOLD;
66     PM5CTL0 &= ~LOCKLPM5;
67
68     P1DIR |= BIT0;
69     P1OUT &= ~BIT0;
70     P1DIR |= BIT1;
71     P1OUT &= ~BIT1;
72     // Configure UART pins
73     P1SEL0 |= BIT4 | BIT5;
74     // Configure UART for Power UP default
75     UCA0CTLW0 |= UCSWRST;
76     UCA0CTLW0 |= UCSSEL__SMCLK;
77     // Baud Rate calculation 115200 baud
78     UCA0BR0 = 8;
79     UCA0MCTLW = 0xD600;
80     UCA0CTLW0 &= ~UCSWRST;
81     UCA0IE |= UCRXIE;
82
83     __bis_SR_register(LPM0_bits|GIE);
84     __no_operation();
85
86 }
```

Fig. 5: Initialisation Part(main) C code

4.1.2 Interrupt

The below figure (see Figure 6) shows the interrupt part C code. Under the $USCI\_A0\_ISR$ interrupt, a $UCA0IV$ switch statement is made to communicate with the RX and TX buffer in UART. In the case of $USCI\_UART\_UCRXIFG$, it will be executed once it detects that the RX buffer is receiving items. However, the first statement in this case is to wait TX buffer to be ready for new data, because if the waiting statement is not placed here, the TX buffer will overflow, and some unexpected errors might happen. Echoing the user's input has been executed by $UCA0TXBUF = UCA0RXBUF$; after waiting for the TX buffer to be ready for new data. To store the input data, a 16-digit register has been created as a global variable. Once the user inputs the wrong character, the user can use *Backspace* (Line 101 to 112 in Figure 6) to back to the previous digit. At the same time, the index counter also will be updated (to back to the previous number). In codes, there is a *DeleteList* used to place a space with a backspace behind in order to overwrite the last element if the user presses *Backspace*. The space is utilised to make the last digit empty in the terminal. The backspace behind is used to back to the index of the last element. Then, the user can input a new character to overwrite the previous element after the *Backspace* is pressed. The counter is also used for detecting whether the user's input reaches 16 or

not. If the counter counts to 16, *LuhnAlgorithm* will be executed. After the void luhnAlgorithm(char *input) function (*LuhnAlgorithm*), Valid data will make the green light be turned on for 0.5 seconds. Invalid data will activate a red light for 1 second.

```
#pragma vector = USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
    switch(UCA0IV)
      {
          case USCI_NONE: break;
          case USCI_UART_UCRXIFG:
              while(!(UCA0IFG & UCTXIFG)); // Wait fo
              char receviedChar = UCA0RXBUF;
              UCA0TXBUF = UCA0RXBUF;
              for(i=0;i<delay;i++); //make sure the
              if(index != 16){    //check the 16 numb
                  if (receviedChar == '\x08'){// to
                     unsigned int k;
                     for(k = 0; k <2;k++){
                       while(!(UCA0IFG & UCTXIFG));
                       UCA0TXBUF = DeleteList[k];
                       for(i=0;i<delay;i++); //make
                     }
                     if (index > 0){
                         index -= 1;
                     }else{
                         index = 0;
                     }
                  }else{
                     charList[index] = receviedChar;
                     index++;
                  }
              }
              if(index == 16){
                  luhnAlgorithm(charList);
                  index = 0;
              }
              break;
          case USCI_UART_UCTXIFG: break;
          case USCI_UART_UCSTTIFG: break;
          case USCI_UART_UCTXCPTIFG: break;
      }

}
```

Fig. 6: Interrupt C code

4.1.3 Luhn Algorithm

The below figure (see Figure 7) is the slice of the Luhn Algorithm function.
The basic introduction of the algorithm is illustrated below:

1. The input is 16-digit, so it needs to check each digit value. To check this, there is for loop to judge each digit in turn. In C, arrays/lists are stored in order from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). Still, when determining parity, it is necessary to start with the least significant bit (LSB), thus using reverse order (descending order from 16) in the for loop.

2. In the loop, the first step is to get the value (the type is char) of the current digit from the input. The next step is to check whether this character is a number. Since ASCII assigns a unique integer value to each character, this is why it is straightforward to determine if a character is a number by comparing the character forms 0 and 9. If this character is not the number, the total sum will be set to 1(1 mode 10 will not be 0 which means this ID is not valid). And then, the input[index] subtracting '0' can convert data value to an integer, due to ASCII. After getting the number in integer form, then there will determine whether the current digit is even- numbered position or not. If it is an even-numbered position, the integer data needs to multiply by 2, and then the result will be checked to see if it is greater than or equal to 10 to determine if it is a two-digit number. If the result is a two-digit number, then each digit should be added and then added to the total sum. If it does not occur, then this result is added directly to the total sum. If this current digit is an odd-numbered position, then add this integer to the total sum. This loop will repeat 16 times to check each digit of the 16-digit ID and update the total sum.

3. After finishing the for loop, the total sum has been updated 16 times. The below figure (see figure below) shows how to determine if an ID is valid. The sum modulo 10 will be checked for verification. If it is 0, the number is valid. If it is not 0, the number is invalid. To display the outcome, this code creates *PassList* and *FailList* and then sends them to TX. There is a timer used to make sure that the LED lights have been turned on for a specific period. In codes, $TA0CTL = MC\_1 \mid ID\_0 \mid TASSEL\_1 \mid TACLR$; has been created. The $TA0CTL$ timer is used to measure the period of turning on red LED light or green LED light. $TA0CCR0$ is used. $TA0CCR0$ will be set to 16384 for 0.5 seconds and will be set to 32768 for 1 second because $ACLK$ is $32.768kHz$. The green LED of the chip will be turned on for 0.5 seconds if the input ID is valid; the red LED of the chip will be turned on for 1 second if the input ID is invalid.

```
void luhnAlgorithm(char *input)
{
    int sum = 0;
    unsigned int j;

  for(j = 16; j >0 ; j--){
       int data = input[j-1] - '0';
     if(input[j-1]>='0' && input[j-1]<='9'){
       if ((j-1) % 2 == 0){
          int tmp = data*2;
          if(tmp >= 10){
              sum += (tmp - 9);
          }
          else{
              sum += tmp;
          }
       }
       else{
           sum += data;
       }
     }
     else{
         sum = 1;
     }
   }

  unsigned int k;
  TA0CTL = MC_1 | ID_0 | TASSEL_1 |TACLR;
  if((sum %10) == 0){
      for(k = 0; k <6;k++){
          while(!(UCA0IFG & UCTXIFG));
          UCA0TXBUF = PassList[k];
          for(i=0;i<delay;i++); //make sure the
      }
     P1OUT |= BIT1;
     TA0CCR0 = 16384;
     while(!(TA0CTL & TAIFG)){}
     TA0CTL &= ~TAIFG; // Clear overflow flag
     P1OUT &= ~BIT1;
     }
  else{
     for(k = 0; k <6;k++){
        while(!(UCA0IFG & UCTXIFG));
        UCA0TXBUF = FailList[k];
        for(i=0;i<delay;i++); //make sure the T
       }
     P1OUT |= BIT0;
     TA0CCR0 = 32768;
     while(!(TA0CTL & TAIFG)){}
     TA0CTL &= ~TAIFG; // Clear overflow flag
     P1OUT &= ~BIT0;
     }
}
```

Fig. 7: Luhn Algorithm C code

5 Design Correctness

There is a main issue which can be discussed here. It is about displaying. The figure below (see Figure 8) illustrates the problems encountered before. The display results become messy once the user inputs characters or numbers very fast.

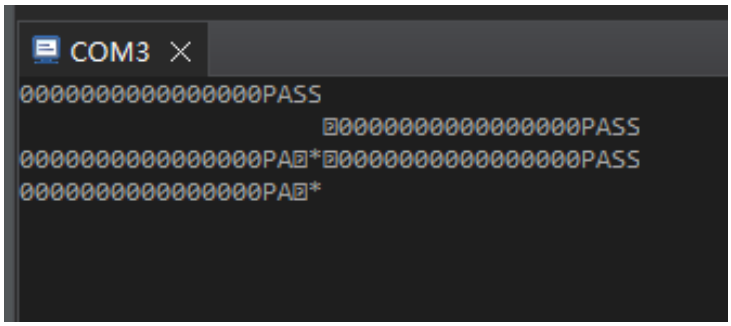

Fig. 8: Screenshot of display problem

There is a simple $for\ loop$ used for making sure that the TX buffer has shown characters on the screen of the terminal. $for(i = 0; i < delay; i++)$; statement has been made to produce the correct result, which i and $delay$ are volatile integers in codes. $delay$ was initialised as 100. Without the delay statement, if the user inputs numbers or characters very fast, the results would display like the figure shown above. The final thought and discussion about this is that hardware will output as fast as possible (TX will output as fast as possible) without a proper delay. This is a typical timing problem, which is an unexpected error as well. It might be a race condition inside the hardware. Luckily, delay can be used in codes to solve this problem. There is a picture (see Figure 9) that shows the correct display result of embedded systems.

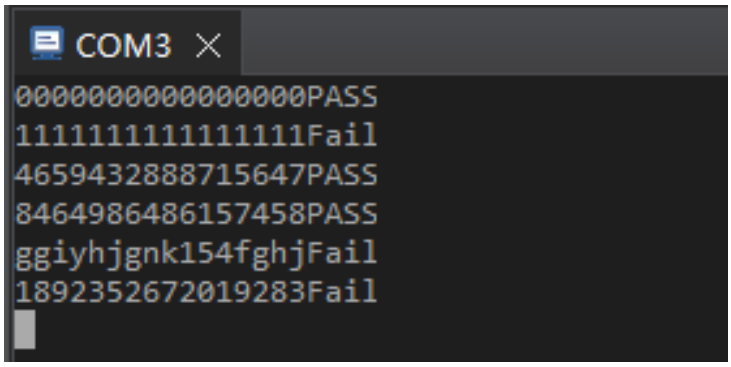

Fig. 9: Correct display result

6 Conclusion

In conclusion, a successful validation of $16-digit\ identification\ numbers$, good communication between MSP430FR2433 and PC, turning on the correct LED colour after verification, and good results displayed on the screen of the terminal have been implemented in this coursework. Implementing $LuhnAlgorithm$, setting timers for LED lights, configuring RX and TX buffer, and setting a good display on the terminal with delay to behave correctly are the tasks to do.