Department of Physics, University of Colombo

PH – 3032 Embedded System Laboratory

Automated Medicine Dispenser

Name: P. V. Nipun Lakshitha
Index No: 15367
Reg No: 2020s18114

Abstract

This project is driven by the increasing challenges faced by older individuals in managing their medications effectively, particularly when faced with the complexities of multiple prescriptions and strict schedules. In order to improve medicine administration in healthcare and help older individuals, the Medicine Dispenser with Alarm, Buzzer, and GSM SIM900A project offers an innovative strategy. It is intended to address the increasing issue of people failing to take their medications on time. This method is originally used for a simple, automated procedure that guarantees on-time medicine intake.

The system is based on the Atmega32 microcontroller interfacing with a stepper motor, an alarm system, and a GSM SIM900A module. When a medication dose is due, the alarm system triggers the dispenser. The stepper motor rotates, precisely dispensing the required medication. Simultaneously, the GSM module sends a reminder SMS to the patient's mobile phone, ensuring that the patient is aware of the medication schedule.

This report provides a comprehensive overview of the project, covering the hardware and software components, circuit design, and system functionality. It discusses the initial setup and configuration process, making it user-friendly and accessible for healthcare providers and patients. Furthermore, we delve into the technical aspects of how the alarm, stepper motor, and GSM module work in harmony to facilitate medication reminders.

By combining these technologies, the Medicine Dispenser project not only simplifies the medication management process but also significantly improves patient compliance. The SMS reminders serve as an effective means of ensuring patients stay on track with their treatment plans. This project represents a significant step forward in leveraging technology to enhance healthcare outcomes and patient well-being.

Contents

Figure Table

Introduction

Medication adherence is essential for the proper management of long-term medical conditions and the success of treatment procedures in the fields of healthcare and patient well-being. Patients may experience worse health outcomes, higher healthcare expenses, and a lower quality of life if they do not follow their prescription regimens. This problem is particularly severe in the elderly, who frequently experience memory loss and have trouble taking their medications as prescribed. Seeing the importance of this problem, our initiative attempts to provide a new solution by merging innovative medical treatment with advanced technology, specifically designed to meet the needs of elderly patients who often have trouble sticking to their drug regimens.

This report details the development and implementation of a Medicine Dispenser with Alarm, Buzzer, and GSM SIM900A, a system engineered to address medication adherence issues. Medication management can be a complex and demanding task for both patients and healthcare providers, often leading to missed doses and reduced therapeutic outcomes. Our project aims to mitigate these challenges by providing a comprehensive solution that automates medication reminders and precise medication dispensing, while simultaneously ensuring patient engagement through real-time communication.

The primary objective of this project is to provide patients with a reliable and automated medication management system. The heart of this system is the Atmega32 microcontroller. A packaging of a vine bottle used for solution for our equipment, modifying its internal components with the use of hardboard. Within this casing, a small circular chamber was created, equipped with eight individual windows, designed to hold, and dispense medical pills. The innovative packaging not only ensures the safe storage of medication but also facilitates an organized and user-friendly experience for patients.

Atmgea32 Microcontroller

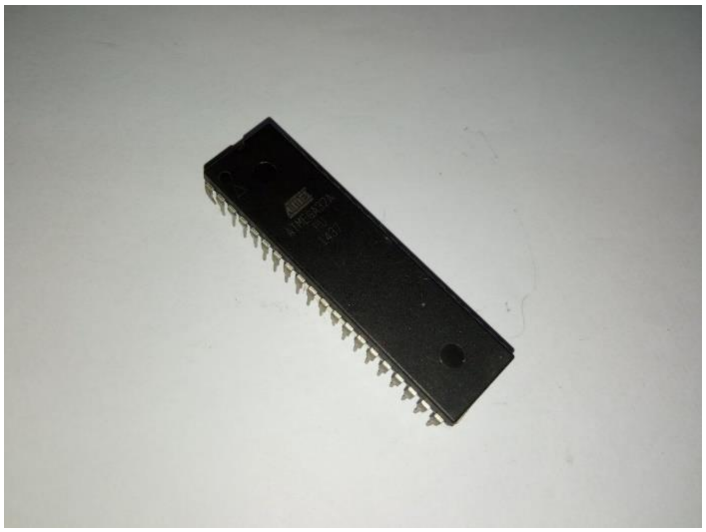


Figure 1: Atmega32 Microcontroller (https://www.microchip.com/en-us/product/ATmega32)

In this test we made a digital watch mainly using the atmega32 microcontroller. Before making this, it is very important to have a good understanding of the times, counters, and internal interrupts of the atmega32. Atmega32 microcontroller has three timers in which Timer0 is 8 bits, Timer1 is 16 bit and Timer2 is 8 bits.

In most cases, timers and counters are used to generate time delays, waveforms and counting events. In studying timers, it is very important to study the basic registers. Each timer in the AVR has a timer/counter register called TCNTn. The 8-bit timers are termed as TCNTn while the 16-bit timers are termed as two 8-bit registers (TCNTnH, TCNTnL). Every timer has its own TCCR (Timer/Counter Control Register) register that is responsible to choose different operational modes. The TCNTn is basically a counter. It counts up with each pulse. It contains zero when reset. Each of the timers has TOV called Time Overflow. The town flag is set when the timer overflows, where 'n' is any number of a timer. These timers also have Timer/Counter Control Register. It is denoted as TCCRn. It is used for setting up the modes of the timer. Another register of the timer is OCRn which is named as "Output Compare register". It is used as Compare match register. Here the contents of OCRn are compared with the contents of TCNTn.

Stepper motor

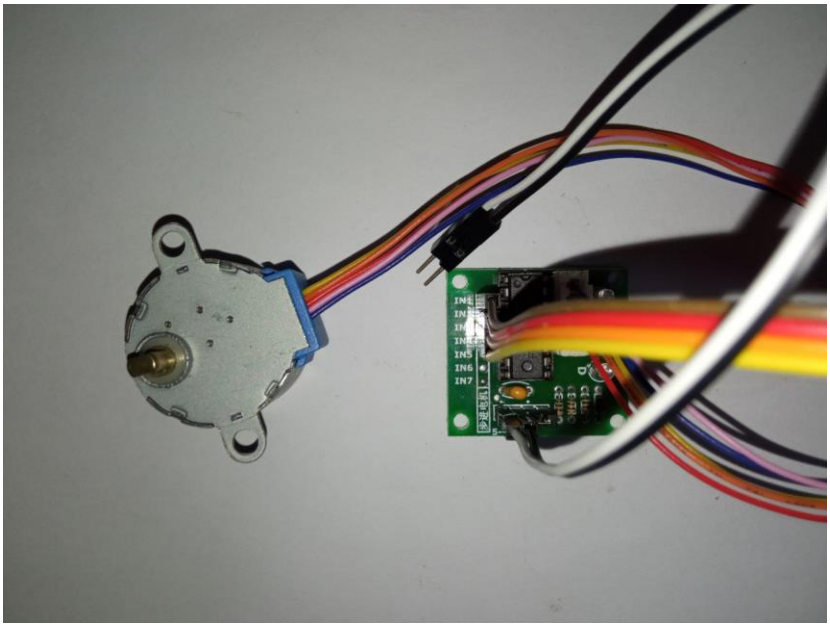


Figure 2: Stepper motor with motor driver(https://microdigisoft.com/28byj-48-stepper-motor-anduln2003-stepper-motor-driver)

Stepper motor is a brushless dc motor in control signals are applied to stepper motor to rotate it in steps. Its speed of rotation depends upon rate at which control signals are applied. There are various stepper motors available with minimum required step angle. Stepper motor is made up of mainly two parts, a stator and rotor. Stator is of coil winding and rotor is mostly permanent magnet or ferromagnetic

material. 28-BYJ48 Stepper motor is most used unipolar motor. The unipolar stepper motor has five or six wires and four coils. The center connections of the coils are tied together and used as the power connection. They are called unipolar steppers because power always comes in on this one pole.

GSM Sim 900 module

Figure 3: GSM Sim 900 module

The SIM900 is a versatile GSM module that revolutionizes wireless communication, facilitating robust, low-power, and reliable cellular connectivity. It is essential for voice, SMS, and data communication over GSM networks, making it an integral part of modern wireless technologies. The module uses various interfaces for communication with external microcontrollers or devices, typically using UART or other serial communication protocols. It requires a SIM card, which contains subscriber information and authentication data. The module initializes and attempts to register on the available GSM network, sending registration information to the nearest cell tower. It supports voice calls, SMS, and data communication over GPRS. AT commands control the module, and response handling is handled by the microcontroller or device. The module requires an external GSM antenna for optimal signal reception and transmission.

A 5V, 2A power supply must be provided for this sim900 GSM shield. If less current passes through this module, the sim900 chip will go to sleep mode. Therefore, it is important to provide the power through a power supply. In the same way, when the power is given to this GSM shield, the status LED is on and the net light blinks for only six seconds and goes off, indicating that the power received for the chip is not enough. If the module is connected to the nearby telephone tower, the net light will blink every third of a second.

Theory

Timer1

We often use timers to generate time delay, count in events, PWM generation, capturing event and generate waveforms. There are three main types of timers in the atmega32 microcontroller, and they are shown below.

- **Timer 0** - 8 bit counter - Counts up to 255
- **Timer 1** - 16 bit counter - Counts up to 65535
- **Timer 2** - 8 bit counter - Counts up to 255

It is very important to study the basic registers and flags while studying the timers of the atmega32 microcontroller.

1. **TCNTn** - Timer / Counter Register
    - Every timer has a timer/counter register. It is zero upon reset. We can access value or write a value to this register. It counts up with each clock pulse.

2. **TOVn** - Timer Overflow Flag
    - Each timer has a Timer Overflow flag. When the timer overflows, this flag will get set.

3. **TCCRn** - Timer Counter Control Register
    - This register is used for setting the modes of timer/counter.

4. **OCRn** - Output Compare Register
    - The value in this register is compared with the content of the TCNTn register. When they are equal, the OCFn flag will get set.

Timer0 Registers

Each timer in the AVR has a timer/counter register called TCNTn. Every timer has its own TCCR (Timer/Counter Control Register) register that is responsible to choose different operational modes. The TCNTn is basically a counter. It counts up with each pulse. It contains zero when reset. Each of the timers has TOV called Time Overflow. The town flag is set when the timer overflows, where 'n' is any number of a timer. These timers also have Timer/Counter Control Register. It is denoted as TCCRn. It is used for setting up the modes of the timer. Another register of the timer is OCRn which is named as "Output Compare register". It is used as Compare match register. Here the contents of OCRn are compared with the contents of TCNTn.

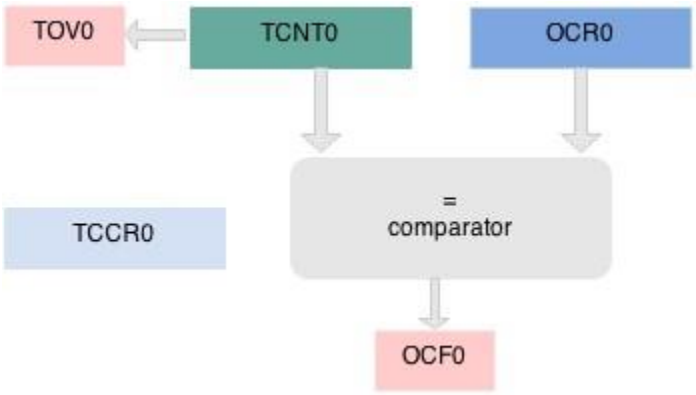
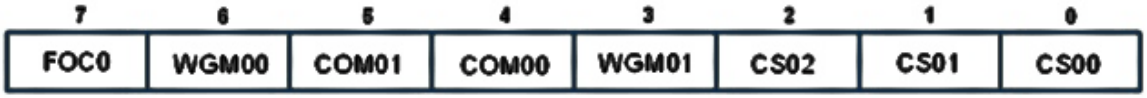


Figure 4: Timer0 registers in atmega32 microcontroller

1. **TCNT0 - Timer / Counter Register 0**
    - TCNT0 is an 8 bit register and it counts each pulse.

2. **TCCR0 - Timer / Counter Control register 0**
    - TCCR0 is the 8 bits register and it's used to operation mode and clock source selection.



I. **FOC0 (Bit 7)** - Force compare match
    a. Write only a bit, which can be used while generating a wave. Writing 1 to this bit causes the wave generator to act as if a compare match has occurred.

II. **WGM00, WGM01 (Bit 6, 3)** - Waveform Generation Mode

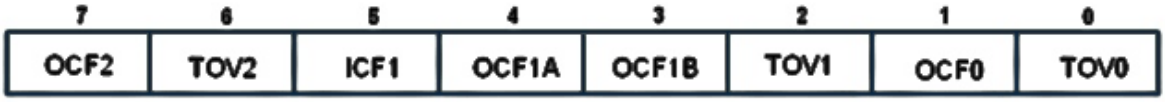
WGM00	WGM01	Timer0 mode selection bit
0	0	Normal
0	1	CTC
1	0	PWM, Phase Correct
1	1	Past PWM

I. **COM01:00 (Bit 5:4)** - Compare Output Mode
   a. These bits control the waveform generator. We will see this in the compare mode of the timer.

II. **CS02:CS00 (Bit 2:0)** - Clock Source Select
   a. These bits are used to select a clock source. When CS02: CS00 = 000, then timer is stopped. As it gets a value between 001 to 101, it gets a clock source and starts as the timer.

CS02	CS01	CS00	Description
0	0	1	No clock source (Timer / Counter stopped)
0	0	0	clk (no pre-scaling)
0	1	1	clk / 8
0	1	0	clk / 64
1	0	1	clk / 256
1	0	0	clk / 1024
1	1	1	External clock source on T0 pin. Clock on falling edge
1	1	0	External clock source on T0 pin. Clock on ricing edge

Timer Counter Interrupt Flag Register (TIFR)



I. TOV0 (Bit 0) - Timer0 Overflow flag

   0 = Timer0 did not overflow

   1 = Timer0 has overflown (going from 0xFF to 0x00)

II. OCF0(Bit 1) - Timer0 Output Compare flag

0 = Compare match did not occur

1 = Compare match occurred

III. TOV1(Bit 2) - Timer1 Overflow flag

IV. OCF1B (Bit 3) - Timer1 Output Compare B match flag

V. OCF1A (Bit 4) - Timer1 Output Compare A match flag

VI. ICF1(Bit 5) - Input Capture flag

VII. TOV2(Bit 6) - Timer2 Overflow flag

VIII. OCF2(Bit 7) - Timer2 Output Compare match flag

Analogue to Digital Converter

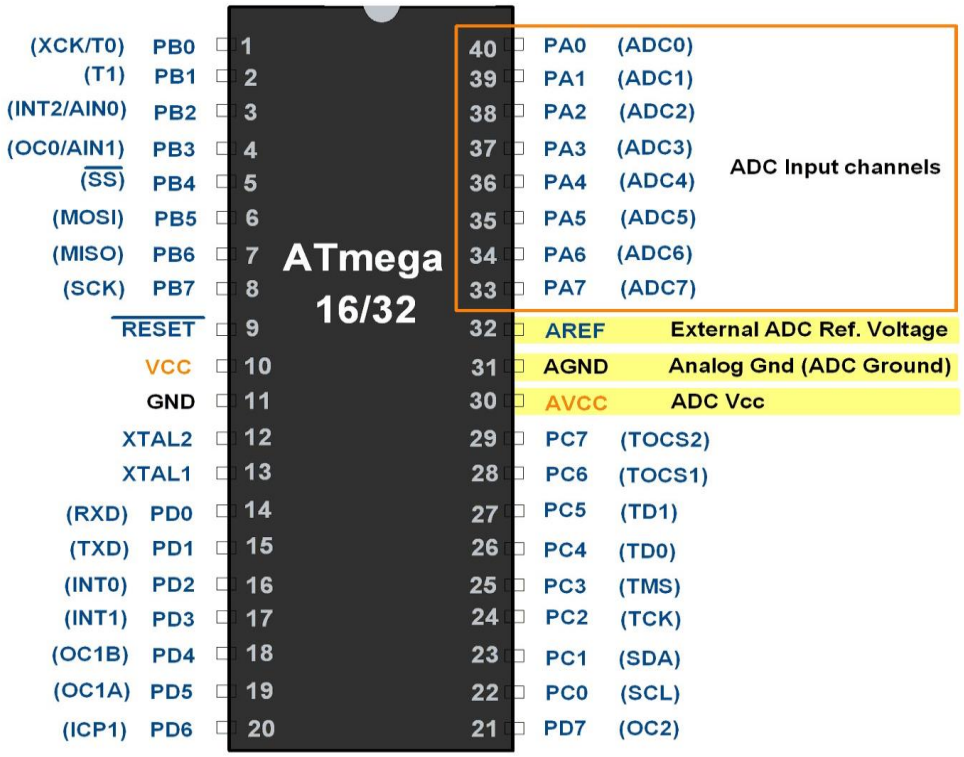


Figure 5: Atmaga32A microcontroller ADC Read Pins (https://www.electronicwings.com/avratmega/atmega1632-gpio-ports-and-registers)

An analogue to digital converter is the digital conversion of analogue data from a sensor for light, temperature, distance, power, etc. Here the analogue pin takes the voltage values and gives it as digital output. The Atmega32 has a 10 bit and it measure we can get a digital output 0 to 1023. That is, if we

give zero voltage as the input voltage, ADC will give 0 as the output and if we give five voltages as the reference voltage, the maximum ADC value will be 1023.

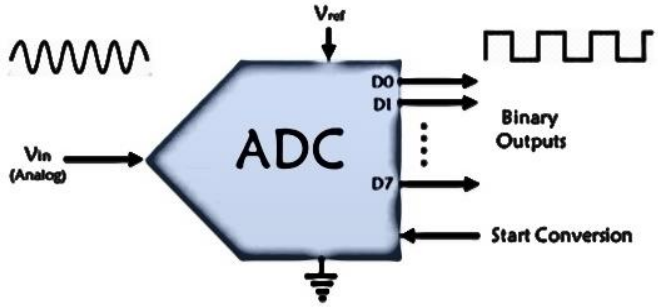


Figure 6: Atmega32 microcontroller Analogue to digital converter

Step size and Resolution

Where step size is the reference voltage divided by the number of bits in the microcontroller divided by the power. That is, the step size of a 10-bit microcontroller that delivers 5V as a reference voltage is 4.88mv. That is, we get the digital data output as the value of the output voltage divided by the step size. The smaller the step size, the higher the resolution, which is called high-resolution. Similarly, when the step size increases, it is called low-resolution. We can reduce the reference voltage or increase the number of bits to increase the resolution.

$$Step\ Size = \left(\frac{V_{ref}}{2_n}\right)$$

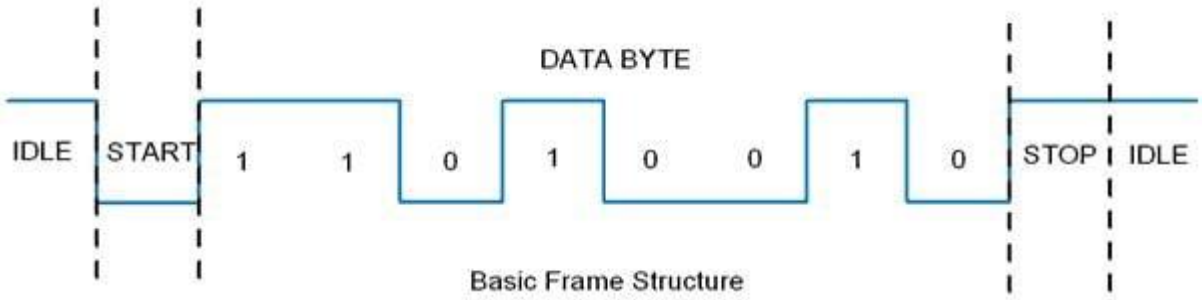
$$Data\ Output = \left(\frac{V_{out}}{Step\ Size}\right)$$

USART

Serial data framing

While sending/receiving data, some bits are added for the purpose of knowing the beginning/ending of data, etc. commonly used structure is: 8 data bits, 1 start bit (logic 0), and 1 stop bit (logic 1), as shown

There are also other supported frame formats available in UART, like parity bit, variable data bits (5-9 data bits).



Basic Frame Structure

Speed (Baud rate)

As we know the bit rate is "Number of bits per second (bps)", also known as Baud rate in Binary system. Normally this defines how fast the serial line is. There are some standard baud rates defined e.g. 1200, 2400, 4800, 19200, 115200 bps, etc. Normally 9600 bps is used where speed is not a critical issue.

Wires and Hardware connection

Normally in USART, we only need Tx (Transmit), Rx(Receive), and GND wires.

- AVR ATmega USART has a TTL voltage level which is 0 v for logic 0 and 5 v for logic 1.
- In computers and most of the old devices, RS232 protocol is used for serial communication, where normally 9 pin 'D' shape connector is used. RS232 serial communication has different voltage levels than ATmega serial communication i.e. +3 v to +25 v for logic zero and -3 v to -25 v for logic 1.
- So to communicate with RS232 protocol, we need to use a voltage level converter like MAX232 IC.

Although there are 9 pins in the DB9 connector, we don't need to use all the pins. Only 2nd Tx(Transmit), 3rd Rx(Receive), and 5th GND pin need to be connected.

To program, first, we need to understand the basic registers used for USART

AVR basic Registers

1. **UDR: USART Data Register**

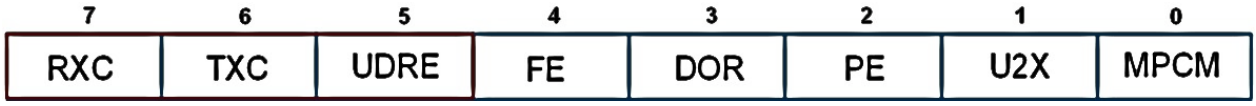
It has basically two registers, one is Tx. Byte and the other is Rx Byte. Both share the same UDR register. Do remember that, when we write to the UDR reg. Tx buffer will get written and when we read from this register, Rx Buffer will get read. Buffer uses the FIFO shift register to transmit the data.

2. **UCSRA**: USART Control and Status Register A. As the name suggests, is used for control and status flags. In a similar fashion, there are two more USART control and status registers, namely UCSRB and UCSRC.

3. **UBRR**: USART Baud Rate Register, this is a 16-bit register used for the setting baud rate.

We will see this register in detail:

UCSRA: USART Control and Status Register A



- **Bit 7 – RXC:** USART Receive Complete

This flag bit is set when there is unread data in UDR. The RXC Flag can be used to generate a Receive Complete interrupt.

- **Bit 6 – TXC:** USART Transmit Complete

This flag bit is set when the entire frame from Tx Buffer is shifted out and there is no new data currently present in the transmit buffer (UDR). The TXC Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXC Flag can generate a Transmit Complete interrupt.

- **Bit 5 – UDRE:** USART Data Register Empty

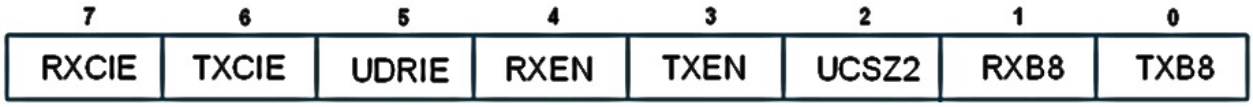
If UDRE is one, the buffer is empty which indicates the transmit buffer (UDR) is ready to receive new data. The UDRE Flag can generate a Data Register Empty Interrupt. UDRE is set after a reset to indicate that the transmitter is ready.

- **Bit 4 – FE:** Frame Error
- **Bit 3 – DOR:** Data OverRun

This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters) and a new character is waiting in the receive Shift Register.

- **Bit 2 – PE:** Parity Error
- **Bit 1 – U2X:** Double the USART Transmission Speed
- **Bit 0 – MPCM:** Multi-processor Communication Mode

UCSRB: USART Control and Status Register B

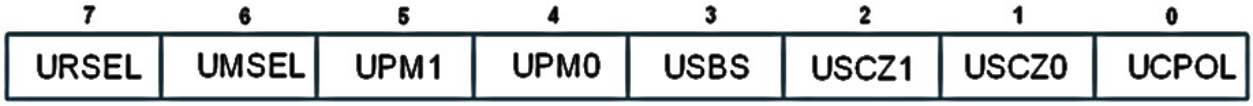


- **Bit 7 – RXCIE:** RX Complete Interrupt Enable
  Writing one to this bit enables interrupt on the RXC Flag.
- **Bit 6 – TXCIE:** TX Complete Interrupt Enable
  Writing one to this bit enables interrupt on the TXC Flag.
- **Bit 5 – UDRIE:** USART Data Register Empty Interrupt Enable
  Writing one to this bit enables interrupt on the UDRE Flag.
- **Bit 4 – RXEN:** Receiver Enable
  Writing one to this bit enables the USART Receiver.
- **Bit 3 – TXEN:** Transmitter Enable
  Writing one to this bit enables the USART Transmitter.
- **Bit 2 – UCSZ2:** Character Size

The UCSZ2 bits combined with the UCSZ1:0 bit in UCSRC sets the number of data bits (Character Size) in a frame the receiver and transmitter use.

- **Bit 1 – RXB8:** Receive Data Bit 8
- **Bit 0 – TXB8:** Transmit Data Bit 8

UCSRC: USART Control and Status Register C



- **Bit 7 – URSEL:** Register Select

This bit selects between accessing the **UCSRC** or the **UBRRH** Register, as both register shares the same address. The URSEL must be one when writing the UCSRC or else data will be written in the UBRRH register.

- **Bit 6 – UMSEL:** USART Mode Select
  This bit selects between the Asynchronous and Synchronous mode of operation.
  0 = Asynchronous Operation
  1 = Synchronous Operation

- **Bit 5:4 – UPM1:0**: Parity Mode

These bits enable and set the type of parity generation and check. If parity a mismatch is detected, the PE Flag in UCSRA will be set.

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

- **Bit 3 – USBS**: Stop Bit Select

This bit selects the number of Stop Bits to be inserted by the Transmitter. The Receiver ignores this setting.

0 = 1-bit
1 = 2-bit

- **Bit 2:1 – UCSZ1:0**: Character Size

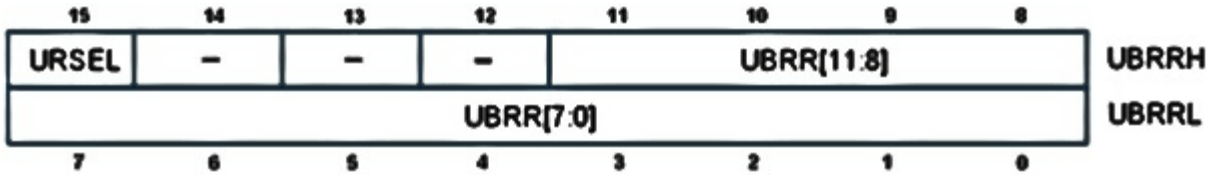
The UCSZ1:0 bits combined with the UCSZ2 bit in UCSRB sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	**1**	**1**	**8-bit**
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

- **Bit 0 – UCPOL:** Clock Polarity

This bit is used for synchronous mode only. Write this bit to zero when the asynchronous mode is used.

UBRRL and UBRRH: USART Baud Rate Registers



- **Bit 15 – URSEL:** Register Select

This bit selects between accessing the **UCSRC** or the **UBRRH** Register, as both register shares the same address. The URSEL must be one when writing the UCSRC or else data will be written in the UBRRH register.

- **Bit 11:0 – UBRR11:0:** USART Baud Rate Register.

  Used to define the baud rate

$$\mathrm{UBBR} = \frac{\mathrm{Fosc}}{16 * \mathrm{BaudRate}} - 1 \qquad \mathrm{BaudRate} = \frac{\mathrm{Fosc}}{16 * (\mathrm{UBBR} + 1)}$$

Methodology

The methodology for the development of the Medicine Dispenser involves a layered structural design comprising three layers.

Figure 7: Medicine Dispenser Box

Bottom Layer:

The bottom layer houses essential electronic components, including the PCB (Printed Circuit Board), GSM module, LCD display, push buttons, and switches. These components are central to the control and communication aspects of the Medicine Dispenser system.

Second Layer:

The second layer accommodates a drawer mechanism designed for the secure and controlled retrieval of medications. This drawer is a fundamental component in the medication dispensing process.

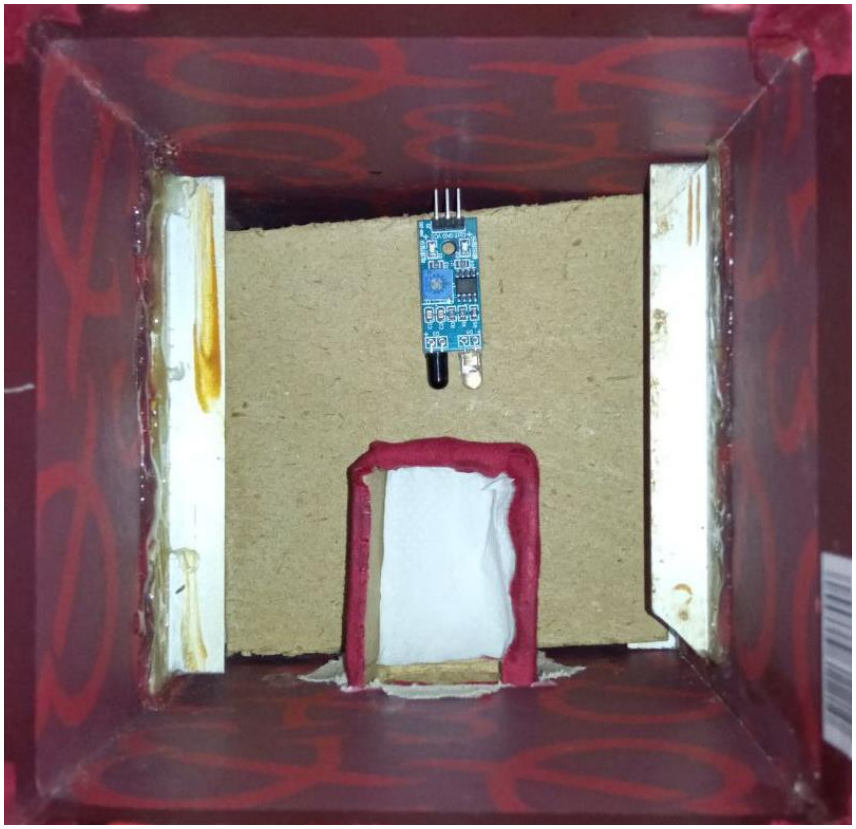


Figure 8: Second layer upper view

Top Layer:

The top layer features the mechanical components responsible for the precise medication dispensing operation. This layer includes a stepper motor and a medicine chamber, constructed using rigid foam material. The medicine chamber is partitioned into eight sections, each dedicated to housing a specific medication. These sections are engineered to release medications sequentially as the stepper motor rotates, aligning with the prescribed dosing schedule.

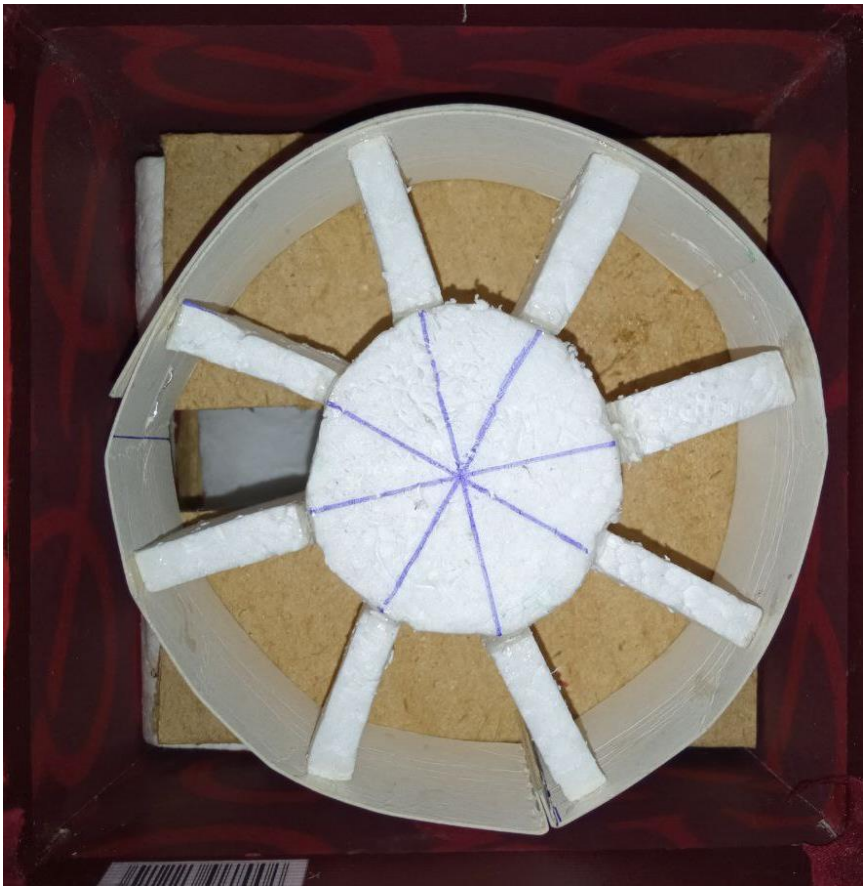


Figure 9: Top layer upper view

The structural arrangement of these layers is integral to the successful operation of the Medicine Dispenser. The bottom layer's electronics facilitate user interaction and communication, while the second layer ensures controlled access to medications. The top layer's mechanical components, including the stepper motor and medicine chamber, automate the dispensing process, providing a reliable and user-friendly solution for medication management.

First a PCB was prepared to implement the atmega32 chip which acts as our main operating unit. An 8Mhz crystal and a Reset push button were used in the design of this PCB. Female Barrel jack were used to get external power. After that PCB was routed using components, a PDF of its top skin layer was obtained. After that, the PDF of the obtained circuit was printed out by a laser printer, and it was cut with a border of about five millimeters. Then place the circuit sheet on a copper cad board that has been cleaned well with sandpaper and iron it well with an electric iron for only 5 minutes. Then, after the toner dust and copper clad board are firmly attached, remove the stickers on top by applying water. Then put it in ferric chloride solution and remove unnecessary copper. Then the thinner solution was used to remove the toner on the remaining copper strips. Drill holes were drilled using a hand drill in the PCB made in this way.

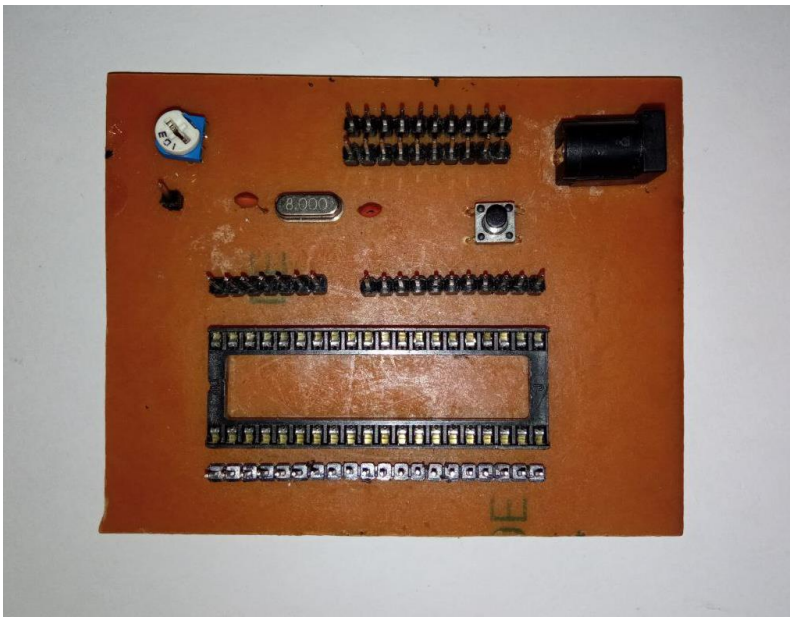


Figure 10: Main PCB

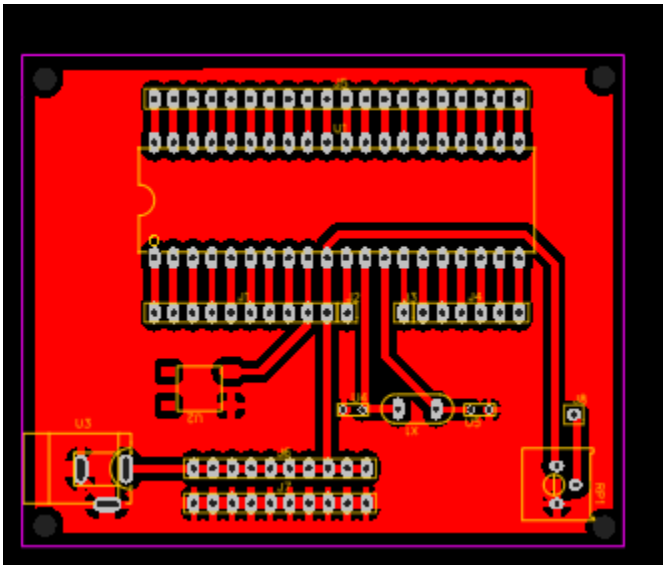


Figure 11: Schematic diagram of Main PCB

Four push buttons and one switch are used in this medical box. When this push button is pushed, a bouncing effect may cause inconvenience to the user. A separate PCB was designed to avoid the bouncing effect. A 100nf capacitor was applied in parallel with the push button and it was pulled up by a 10K resistor. Then until the capacitor is charged and discharged, even if you push the push button again, the numbers on the LCD will stop moving continuously. Similarly, the time taken to charge and discharge the capacitor is as small as one ns, so this is a very successful method.

Figure 12: Bounce PCB

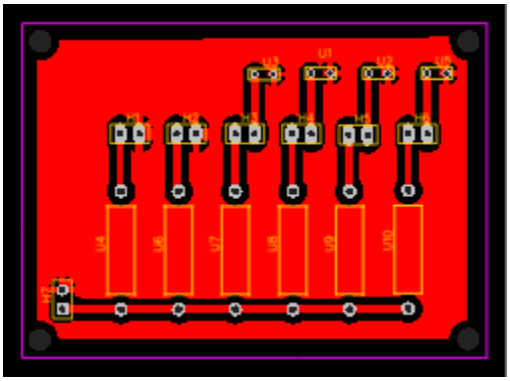


Figure 13: Schematic diagram of Bounce PCB

This device indicates to get medicine by a red LED, buzzer will turn on for 15 seconds and patient will be informed by a text message.

How the components are connected to Atmega32

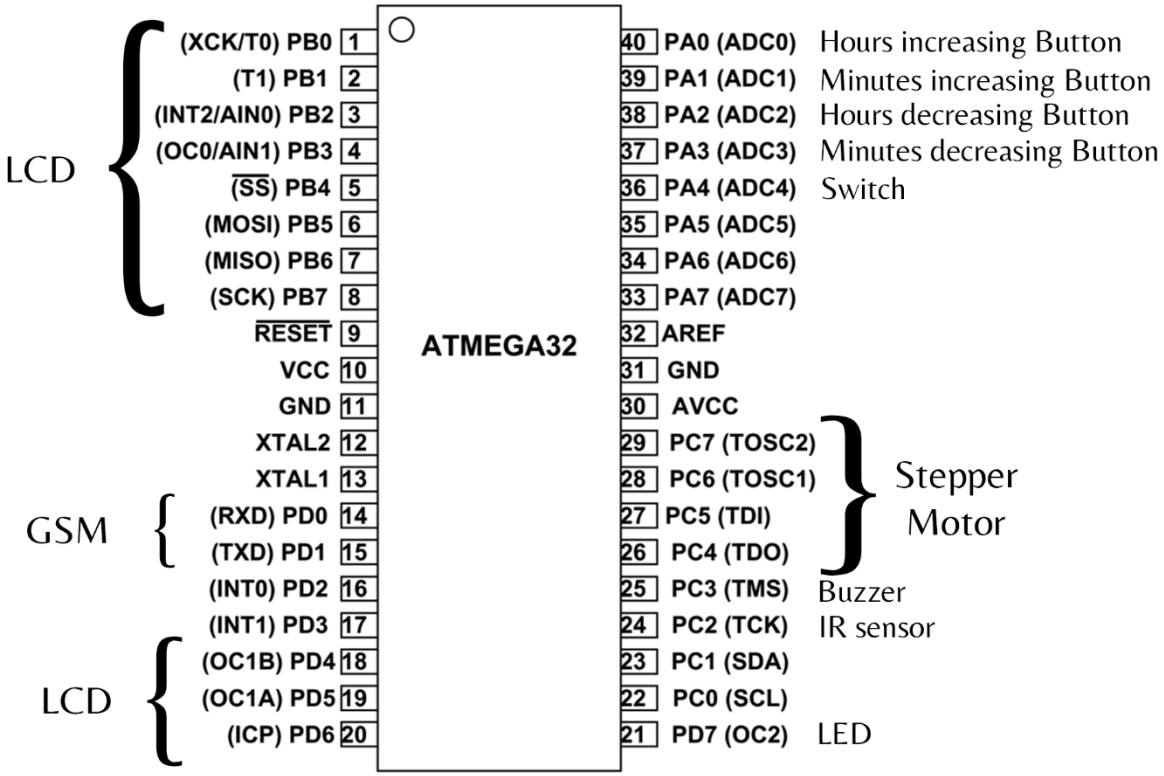


Figure 14: how components are connected

Code Implementation

Main Function:

- This code checks three conditions: "alarm_triggered", "motorRotationDone", and "medicineTaken".
- It's primarily concerned with a stepper motor that should rotate 64 steps if the conditions are met.
- The loop iterates 64 times and changes the state of PORTC to control the motor's rotation for 45. It waits for a certain period defined by the "period" variable.
- Finally, it sets " motorRotationDone " to 1 to indicate that the rotation has been done to prevent rotate it repeatedly.

```
DDRC |= (1 << PC4) | (1 << PC5) | (1 << PC6) | (1 << PC7); // Set PC4-PC7 as
outputs
int period = 5;

if (alarm_triggered && !motorRotationDone && !medicineTaken) {
            // Perform a 64 steps rotation
            for (int i = 0; i < 64; i++) {
                PORTC = 0x10;
                _delay_ms(period);
                PORTC = 0x30;
                _delay_ms(period);
                PORTC = 0x20;
                _delay_ms(period);
                PORTC = 0x60;
                _delay_ms(period);
                PORTC = 0x40;
                _delay_ms(period);
                PORTC = 0xC0;
                _delay_ms(period);
                PORTC = 0x80;
                _delay_ms(period);
                PORTC = 0x90;
                _delay_ms(period);
            }
            motorRotationDone = 1;
        }
```

- This code checks two conditions: "alarm_triggered" and "massageSend".
- If these conditions are met, it initializes UART communication with a GSM module.
- It sends a sequence of AT commands to set up the message, recipient, and message text.

- Finally, it sets "massageSend" to 1 to indicate that the message has been sent to prevent sending it repeatedly.

```
if (alarm_triggered && !massageSend) {
            UART_init(9600);        // Initialize UART communication with a baud
rate of 9600
            UART_SendString("AT\r\n");  // Send AT command to the GSM module
            _delay_ms(3000);
            UART_SendString("ATE0\r\n");    // Disable command echo
            _delay_ms(3000);
            UART_SendString("AT+CMGF=1\r\n");// Set SMS text mode
            _delay_ms(3000);
            UART_SendString("AT+CMGS=\"+94778797936\"\r\n");// Set recipient
phone number
            _delay_ms(3000);
            UART_SendString("It's time to take your medicine");// Send SMS text
            UART_TxChar(26);        // Send Ctrl+Z to indicate end of SMS
        massageSend = 1;
```

UART Functions:

You have functions to initialize and use UART communication for sending SMS messages through a GSM module.

```
// Initialize UART with the specified baud rate
void UART_init(long USART_BAUDRATE){
    UCSRB |= (1 << RXEN) | (1 << TXEN);     // Turn on transmission and reception
    UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Use 8-bit character
sizes
    UBRRL = BAUD_PRESCALE;          // Load lower 8-bits of the baud rate value
    UBRRH = (BAUD_PRESCALE >> 8);       // Load upper 8-bits
}

// Receive a character from UART
unsigned char UART_RxChar(void){
    while ((UCSRA & (1 << RXC)) == 0);      // Wait till data is received
    return(UDR);                // Return the received byte
}

// Transmit a character through UART
void UART_TxChar(char ch){
    while (!(UCSRA & (1<<UDRE)));       // Wait for empty transmit buffer
    UDR = ch;
}
```

```
// Send a string through UART
void UART_SendString(char *str){
    unsigned char j=0;
    while (str[j] != 0){            // Send string till null terminator
        UART_TxChar(str[j]);
        j++;
    }
}
```

LCD Functions:

These functions deal with initializing and controlling the LCD. You can send commands and strings to the LCD for display.

```
// LCD
void LCD_Command(unsigned char cmnd)
{
    LCD_Data_Port= cmnd;
    LCD_Command_Port &= ~(1<<RS);    // RS=0 command reg.
    LCD_Command_Port &= ~(1<<RW);    // RW=0 Write operation
    LCD_Command_Port |= (1<<EN);     // Enable pulse
    _delay_us(1);
    LCD_Command_Port &= ~(1<<EN);
    _delay_ms(3);
}

void LCD_Char (unsigned char char_data) // LCD data write function
{
    LCD_Data_Port= char_data;
    LCD_Command_Port |= (1<<RS);     // RS=1 Data reg.
    LCD_Command_Port &= ~(1<<RW);    // RW=0 write operation
    LCD_Command_Port |= (1<<EN);     // Enable Pulse
    _delay_us(1);
    LCD_Command_Port &= ~(1<<EN);
    _delay_ms(1);
}

void LCD_Init (void)            // LCD Initialize function
{
    LCD_Command_Dir = 0xFF;      // Make LCD command port direction as o/p
    LCD_Data_Dir = 0xFF;         // Make LCD data port direction as o/p
    _delay_ms(50);           // LCD Power ON delay always >15ms
```

```
    LCD_Command (0x38);     // Initialization of 16X2 LCD in 8bit mode
    LCD_Command (0x0C);     // Display ON Cursor OFF
    LCD_Command (0x06);     // Auto Increment cursor
    LCD_Command (0x01);     // Clear display
    LCD_Command (0x80);     // Cursor at home position
}

void LCD_String (char *str)     // Send string to LCD function
{
    int i;
    for(i=0;str[i]!=0;i++)      // Send each char of string till the NULL
    {
        LCD_Char (str[i]);
    }
}

void LCD_String_xy (char row, char pos, char *str)// Send string to LCD with xy
position
{
    if (row == 0 && pos<16)
    LCD_Command((pos & 0x0F)|0x80); // Command of first row and required
position<16
    else if (row == 1 && pos<16)
    LCD_Command((pos & 0x0F)|0xC0); // Command of first row and required
position<16
    LCD_String(str);        // Call LCD string function
}

void LCD_Clear()
{
    LCD_Command (0x01);     // clear display
    LCD_Command (0x80);     // cursor at home position
}
```

Button Functions:

You've defined functions to initialize buttons and update the alarm or time based on button presses. Depending on the state of PA4, you're updating either the alarm time or the clock time.

IR Sensor:

The code initializes the IR sensor, and checkMedicine() is used to check if an obstacle (indicating medicine taken) is detected.

```
// IR Sensor
```

```
void initIRSensor(void) {
    DDRD &= ~(1 << IR_SENSOR_PIN);  // Set IR sensor pin as input
    PORTD |= (1 << IR_SENSOR_PIN);  // Enable pull-up resistor for IR sensor
    }

void checkMedicine(void){
    if (PIND & (1 << IR_SENSOR_PIN)) {
        medicineTaken = 1;          // No obstacle detected
    } else {
        medicineTaken = 0;          // Obstacle detected
    }
}
```

Alarm Function:

This function manages the alarm logic. It handles turning on the LED and buzzer when the alarm is triggered. The buzzer is turned off after 30 seconds. If medicine is taken, both the LED and buzzer are turned off. The alarmActive flag is used to keep track of the alarm state.

```
//CheckAlarms
void CheckAlarms() {            // Function to check if an alarm is triggered
    if (hours == alarm_hours && minutes == alarm_minutes) {
        alarm_triggered = 1;
        AlarmFunction();        //Call the function for Alarm
    }
}
```

```
void AlarmFunction() {
    static uint8_t alarmActive = 0;  // Flag to indicate if alarm is active

    if (alarm_triggered) {
        if (!alarmActive) {
            alarmActive = 1;
            massageSend = 0;
            motorRotationDone = 0;
            PORTD |= (1 << LED_PIN);  // Turn on the LED
            PORTD |= (1 << BUZZER_PIN);  // Turn on the buzzer
        } else if (seconds >= 15) {  // Check if 15 seconds have passed
            PORTD &= ~(1 << BUZZER_PIN);  // Turn off the buzzer after 15 seconds
        }
    } else {
        alarmActive = 0;  // Reset the alarm active flag when the alarm is not
triggered
        PORTD &= ~(1 << LED_PIN);  // Turn off the LED
        PORTD &= ~(1 << BUZZER_PIN);  // Turn off the buzzer
```

```
    }

    if (medicineTaken) {
        PORTD &= ~(1 << LED_PIN);  // Turn off the LED if medicine is taken
        PORTD &= ~(1 << BUZZER_PIN);  // Turn off the buzzer if medicine is taken
    }
}
```

Interrupt Service Routine (ISR):

The Timer1A compare match interrupt service routine (ISR) is used to keep track of time. It updates the seconds, minutes, and hours variables.

```
// Timer1A compare match interrupt service routine
ISR(TIMER1_COMPA_vect) {
    seconds = (seconds + 1) % 60;   // Called every 1 second

    if (seconds == 0) {         // If seconds reach 60, increment minutes
        minutes = (minutes + 1) % 60;

        if (minutes == 0) {     // If minutes reach 60, increment hours
            hours = (hours + 1) % 24;
            }
        }
}
```

Discussion

Challenges and Solutions

In this project, we have designed and implemented a multifunctional embedded system using the Atmega32 microcontroller. Our system integrates various hardware components and features, such as an LCD display, user input through buttons, an infrared obstacle sensor, and output mechanisms including an LED and a buzzer. The system's primary functionality revolves around time management, user-specific alarms, and medication reminders. Furthermore, it employs GSM communication capabilities to send text message notifications when an alarm is triggered. This project discussion will delve into the design choices, challenges faced, and the system's overall performance. We will also explore possible improvements and future enhancements to extend the system's utility and reliability.

This medicine box makes it easy for senior patients and those suffering from dementia to take their daily doses by conveniently storing a week's worth of medication. It is now intended for once-daily prescription drugs. For people who need to take their medication more frequently, it does provide flexibility. Setting an alarm for every meal can help facilitate multiple doses by providing timely reminders. Expanding the gadget to hold several alarms, simulating the operation of a single alarm, is another improvement. Due to this enlargement, there must be more medicine compartments added; a total of twenty-one compartments are needed to accommodate three daily doses for a week.

Furthermore, it's important to confirm that the medicine chamber is positioned exactly and that the stepper motor's revolutions are configured correctly. Unintentional outcomes from an incorrect design include the patient receiving two doses of medication at the same time or not getting any medication at all. This highlights how important accurate engineering and calibration are to ensuring the device's dependability and efficiency in medicine administration.

The alarm clock in this device is designed for a 1 MHz frequency and uses interrupts for precision. However, it's important to note that when the device loses power, the real-time clock (RTC) requires recalibration. To overcome this challenge, we recommend the integration of a DS1307 or DS3231 RTC module.

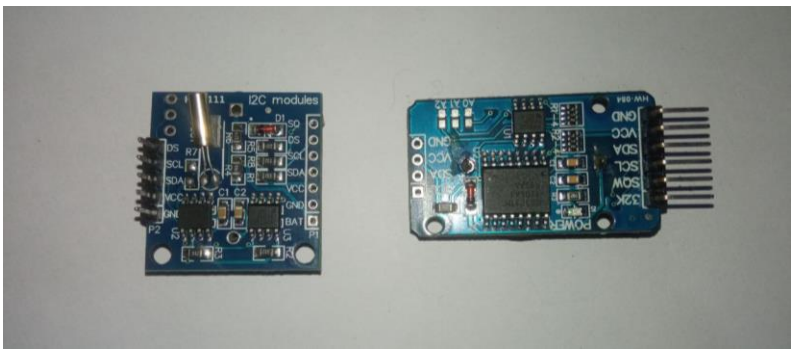


Figure 15: Real time clock module

I2C communication is used to display real-time data, connecting the RTC module to the ATmega32 microcontroller. The device's timekeeping can drift if idle, so a code allows the LCD to read time from the RTC module and update it as needed.

Future Improvements

Our project establishes a solid basis for further advancements in the attempt to improve healthcare accessibility and efficiency. One of the most important areas that needs improvement is remote monitoring and control. Our goal is to give patients and healthcare providers remote control over the dispensing process and real-time medication adherence data by seamlessly integrating the dispenser with a smartphone app or web interface. This makes managing medications easier and creates opportunities for integrating telehealth.

Refill Alerts are another crucial aspect of our project's future development. A smart sensor system can detect when medication is running low and instantly notify users.

Furthermore, we've decided to include a Battery Backup feature to strengthen the device's stability. With this feature, patients' health will be protected since the dispenser will continue to perform its essential tasks even in the event of a power outage. When combined, these upgrades will significantly boost our project's effectiveness and accessibility, resulting in a more reliable and user-focused medication management solution.

Conclusion

The objective of the project is to develop the Medicine Dispenser, an innovative and user-friendly medication management tool. By resolving the issue of timely medicine intake, this gadget improves patient quality of life and healthcare outcomes. For accurate, automated, and secure dispensing, the device makes use of an infrared sensor, a stepper motor, and an alarm clock system. The project's strengths are found in its functionality and in its potential for further improvements, like battery backup, refill warnings, and remote monitoring and control. The project aims to make an important boost to medication management by providing a dependable and workable solution that improves lives. It also acknowledges the growing significance of technology in healthcare. The initiative is dedicated to advancement and giving people the tools, they need to live longer, healthier lives.

References

- www.electronicwings.com. (n.d.). *USART in AVR ATmega16/ATmega32 | AVR ATmega Controllers*. [online] Available at: https://www.electronicwings.com/avr-atmega/atmega1632-usart.
- www.electronicwings.com. (n.d.). *sim900a GSM module Interfacing AVR ATmega16/ATmega32 | AVR ATmega Controllers*. [online] Available at: https://www.electronicwings.com/avr-atmega/sim900a-gsm-module-interfacingwith-atmega1632.
- www.electronicwings.com. (n.d.). *LCD16x2 Interfacing with AVR ATmega16/ATmega32 | AVR ATmega Contr.*. [online] Available at: https://www.electronicwings.com/avr-atmega/lcd16x2-interfacing-with-atmega16-32.
- www.electronicwings.com. (n.d.). *Stepper Motor with AVR ATmega16/ATmega32 | AVR ATmega Controllers.* [online] Available at: https://www.electronicwings.com/avr-atmega/stepper-motor-interfacing-withatmega32.
- Tutorials. (n.d.). *AVR Timer programming*. [online] Available at: https://exploreembedded.com/wiki/AVR_Timer_programming.

Appendix

```
#define F_CPU 8000000UL
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)


#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>


#define LCD_Data_Dir DDRB       /* Define LCD data port direction */
#define LCD_Command_Dir DDRD        /* Define LCD command port direction register
*/
#define LCD_Data_Port PORTB     /* Define LCD data port */
#define LCD_Command_Port PORTD      /* Define LCD data port */
#define RS PD4              /* Define Register Select (data/command reg.)pin */
#define RW PD5              /* Define Read/Write signal pin */
#define EN PD6              /* Define Enable signal pin */

#define BUTTON_PORT PORTA
#define BUTTON_PIN PINA
#define BUTTON_DDR DDRA
#define HOUR_BUTTON_MAX PA0
#define MINUTE_BUTTON_MAX PA1
#define HOUR_BUTTON_MIN PA2
#define MINUTE_BUTTON_MIN PA3
#define SWITCH PA4

#define LED_PIN PD7
#define BUZZER_PIN PD2

#define IR_SENSOR_PIN    PD3
#define OBSTACLE_DETECTED_THRESHOLD  10



volatile uint8_t hours = 12;    // Initial hours
volatile uint8_t minutes = 30;   // Initial minutes
volatile uint8_t seconds = 55;   // Initial seconds
```

```
volatile uint8_t alarm_hours = 12;
volatile uint8_t alarm_minutes = 31;

volatile uint8_t alarm_triggered = 0;       // Flag to indicate if alarm is
triggered
volatile uint8_t medicineTaken = 0;         // Flag to indicate if medicine have
taken
volatile uint8_t massageSend = 0;           // Flag to indicate if massage has
send
volatile uint8_t motorRotationDone = 0;     // Flag to indicate if motor has
rotated

void Button_Init();
void UpdateAlarmOrTime();
void DisplayTime();

void CheckAlarms();
void DisplayAlarms();

void initIRSensor(void);
void checkMedicine(void);

void AlarmFunction();

// LCD Function prototypes
void LCD_Char (unsigned char char_data);
void LCD_Init (void);
void LCD_String (char *str);
void LCD_String_xy (char row, char pos, char *str);
void LCD_Clear();

// GSM Function prototypes
void UART_init(long USART_BAUDRATE);
unsigned char UART_RxChar(void);
void UART_TxChar(char ch);
void UART_SendString(char *str);



int main() {
    LCD_Init();
    Button_Init();          // Initialize
    initIRSensor();
```

```
    DDRC |= (1 << PC4) | (1 << PC5) | (1 << PC6) | (1 << PC7); // Set PC4-PC7 as
outputs
    int period = 5;

    // Configure Timer1 to generate a 1-second interrupt
    TCCR1B |= (1 << CS12) | (1 << WGM12); // Prescaler 256 and CTC mode
    OCR1A = 31249;          // Compare value for 1 second at 8MHz
    TIMSK |= (1 << OCIE1A);         // Enable Timer1A compare match interrupt
    sei();              // Enable global interrupts

    while (1) {             // Main loop
        UpdateAlarmOrTime();            // Update time based on button presses
        CheckAlarms();
        DisplayTime();          // Display the updated time
        DisplayAlarms();
        checkMedicine();
        AlarmFunction();

        if (alarm_triggered && !motorRotationDone && !medicineTaken) {
            // Perform a 64 steps rotation
            for (int i = 0; i < 64; i++) {
                PORTC = 0x10;
                _delay_ms(period);
                PORTC = 0x30;
                _delay_ms(period);
                PORTC = 0x20;
                _delay_ms(period);
                PORTC = 0x60;
                _delay_ms(period);
                PORTC = 0x40;
                _delay_ms(period);
                PORTC = 0xC0;
                _delay_ms(period);
                PORTC = 0x80;
                _delay_ms(period);
                PORTC = 0x90;
                _delay_ms(period);
            }
            motorRotationDone = 1;
        }

        if (alarm_triggered && !massageSend) {
            UART_init(9600);        // Initialize UART communication with a baud
rate of 9600
            UART_SendString("AT\r\n");  // Send AT command to the GSM module
```

```
            _delay_ms(3000);
            UART_SendString("ATE0\r\n");    // Disable command echo
            _delay_ms(3000);
            UART_SendString("AT+CMGF=1\r\n");// Set SMS text mode
            _delay_ms(3000);
            UART_SendString("AT+CMGS=\"+94778797936\"\r\n");// Set recipient
phone number
            _delay_ms(3000);
            UART_SendString("It's time to take your medicine");// Send SMS text
            UART_TxChar(26);        // Send Ctrl+Z to indicate end of SMS
        massageSend = 1;
        }

    }
    return 0;
}




// Initialize UART with the specified baud rate
void UART_init(long USART_BAUDRATE){
    UCSRB |= (1 << RXEN) | (1 << TXEN);     // Turn on transmission and reception
    UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Use 8-bit character
sizes
    UBRRL = BAUD_PRESCALE;          // Load lower 8-bits of the baud rate value
    UBRRH = (BAUD_PRESCALE >> 8);       // Load upper 8-bits
}

// Receive a character from UART
unsigned char UART_RxChar(void){
    while ((UCSRA & (1 << RXC)) == 0);      // Wait till data is received
    return(UDR);                // Return the received byte
}

// Transmit a character through UART
void UART_TxChar(char ch){
    while (!(UCSRA & (1<<UDRE)));       // Wait for empty transmit buffer
    UDR = ch;
}

// Send a string through UART
void UART_SendString(char *str){
    unsigned char j=0;
```

```
    while (str[j] != 0){            // Send string till null terminator
        UART_TxChar(str[j]);
        j++;
    }
}




// LCD
void LCD_Command(unsigned char cmnd)
{
    LCD_Data_Port= cmnd;
    LCD_Command_Port &= ~(1<<RS);   // RS=0 command reg.
    LCD_Command_Port &= ~(1<<RW);   // RW=0 Write operation
    LCD_Command_Port |= (1<<EN);    // Enable pulse
    _delay_us(1);
    LCD_Command_Port &= ~(1<<EN);
    _delay_ms(3);
}

void LCD_Char (unsigned char char_data) // LCD data write function
{
    LCD_Data_Port= char_data;
    LCD_Command_Port |= (1<<RS);    // RS=1 Data reg.
    LCD_Command_Port &= ~(1<<RW);   // RW=0 write operation
    LCD_Command_Port |= (1<<EN);    // Enable Pulse
    _delay_us(1);
    LCD_Command_Port &= ~(1<<EN);
    _delay_ms(1);
}

void LCD_Init (void)            // LCD Initialize function
{
    LCD_Command_Dir = 0xFF;     // Make LCD command port direction as o/p
    LCD_Data_Dir = 0xFF;        // Make LCD data port direction as o/p
    _delay_ms(50);          // LCD Power ON delay always >15ms

    LCD_Command (0x38);     // Initialization of 16X2 LCD in 8bit mode
    LCD_Command (0x0C);     // Display ON Cursor OFF
    LCD_Command (0x06);     // Auto Increment cursor
    LCD_Command (0x01);     // Clear display
    LCD_Command (0x80);     // Cursor at home position
}
```

```
void LCD_String (char *str)     // Send string to LCD function
{
    int i;
    for(i=0;str[i]!=0;i++)      // Send each char of string till the NULL
    {
        LCD_Char (str[i]);
    }
}

void LCD_String_xy (char row, char pos, char *str)// Send string to LCD with xy
position
{
    if (row == 0 && pos<16)
    LCD_Command((pos & 0x0F)|0x80); // Command of first row and required
position<16
    else if (row == 1 && pos<16)
    LCD_Command((pos & 0x0F)|0xC0); // Command of first row and required
position<16
    LCD_String(str);        // Call LCD string function
}

void LCD_Clear()
{
    LCD_Command (0x01);     // clear display
    LCD_Command (0x80);     // cursor at home position
}




// Button
void Button_Init() {            // Set HOUR_BUTTON and MINUTE_BUTTON as input
with pull-up resistors
    BUTTON_DDR &= ~(1 << HOUR_BUTTON_MAX);
    BUTTON_DDR &= ~(1 << MINUTE_BUTTON_MAX);
    BUTTON_PORT |= (1 << HOUR_BUTTON_MAX);
    BUTTON_PORT |= (1 << MINUTE_BUTTON_MAX);

    BUTTON_DDR &= ~(1 << HOUR_BUTTON_MIN);
    BUTTON_DDR &= ~(1 << MINUTE_BUTTON_MIN);
    BUTTON_PORT |= (1 << HOUR_BUTTON_MIN);
    BUTTON_PORT |= (1 << MINUTE_BUTTON_MIN);
}

void UpdateAlarmOrTime() {
```

```
    // Check the state of PA4 to determine if you want to set alarms or time
    if (PINA & (1 << SWITCH)) {

        if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MAX))) {// Hour button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MAX))) {
            alarm_hours = (alarm_hours + 1) % 24;       // Increment hours
            _delay_ms(500);
        }
    }

    if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MAX))) { // Minute button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MAX))) {
            alarm_minutes = (alarm_minutes + 1) % 60;   // Increment minutes
            _delay_ms(500);             // Button press delay
        }
    }


    if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MIN))) {// Hour button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MIN))) {
            alarm_hours = (alarm_hours - 1) % 24;       // Decrement hours
            _delay_ms(500);
        }
    }

    if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MIN))) { // Minute button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MIN))) {
            alarm_minutes = (alarm_minutes - 1) % 60;   // Decrement minutes
            _delay_ms(500);             // Button press delay
        }
    }


    } else {

        if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MAX))) {// Hour button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MAX))) {
            hours = (hours + 1) % 24;       // Increment hours
            _delay_ms(500);
        }
```

```
    }

    if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MAX))) { // Minute button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MAX))) {
            minutes = (minutes + 1) % 60;   // Increment minutes
            _delay_ms(500);             // Button press delay
        }
    }


    if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MIN))) {// Hour button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << HOUR_BUTTON_MIN))) {
            hours = (hours - 1) % 24;       // Decrement hours
            _delay_ms(500);
        }
    }

    if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MIN))) { // Minute button is pressed
        _delay_ms(50);              // Debounce delay
        if (!(BUTTON_PIN & (1 << MINUTE_BUTTON_MIN))) {
            minutes = (minutes - 1) % 60;   // Decrement minutes
            _delay_ms(500);             // Button press delay
        }
    }

    }
}

void DisplayTime() {            // Function to display the time
    char timeStr[12];           // Display hours and minutes on LCD
    snprintf(timeStr, sizeof(timeStr), "%02d:%02d:%02d", hours, minutes,
seconds);
    LCD_String_xy(0, 0, timeStr);
}




//CheckAlarms
void CheckAlarms() {            // Function to check if an alarm is triggered
    if (hours == alarm_hours && minutes == alarm_minutes) {
        alarm_triggered = 1;
        AlarmFunction();        //Call the function for Alarm
```

```
    }
}

void DisplayAlarms() {          // Function to display alarms on the LCD
    char alarmStr[12];
    snprintf(alarmStr, sizeof(alarmStr), "%02d:%02d", alarm_hours,
alarm_minutes);
    LCD_String_xy(1, 0, alarmStr);
}




// IR Sensor
void initIRSensor(void) {
    DDRD &= ~(1 << IR_SENSOR_PIN);  // Set IR sensor pin as input
    PORTD |= (1 << IR_SENSOR_PIN);  // Enable pull-up resistor for IR sensor
    }

void checkMedicine(void){
    if (PIND & (1 << IR_SENSOR_PIN)) {
        medicineTaken = 1;          // No obstacle detected
    } else {
        medicineTaken = 0;          // Obstacle detected
    }
}




void AlarmFunction() {
    static uint8_t alarmActive = 0;  // Flag to indicate if alarm is active

    if (alarm_triggered) {
        if (!alarmActive) {
            alarmActive = 1;
            massageSend = 0;
            motorRotationDone = 0;
            PORTD |= (1 << LED_PIN);  // Turn on the LED
            PORTD |= (1 << BUZZER_PIN);  // Turn on the buzzer
        } else if (seconds >= 15) {  // Check if 30 seconds have passed
            PORTD &= ~(1 << BUZZER_PIN);  // Turn off the buzzer after 30 seconds
        }
    } else {
```

```
        alarmActive = 0;  // Reset the alarm active flag when the alarm is not
triggered
        PORTD &= ~(1 << LED_PIN);  // Turn off the LED
        PORTD &= ~(1 << BUZZER_PIN);  // Turn off the buzzer
    }

    if (medicineTaken) {
        PORTD &= ~(1 << LED_PIN);  // Turn off the LED if medicine is taken
        PORTD &= ~(1 << BUZZER_PIN);  // Turn off the buzzer if medicine is taken
    }
}




// Timer1A compare match interrupt service routine
ISR(TIMER1_COMPA_vect) {
    seconds = (seconds + 1) % 60;   // Called every 1 second

    if (seconds == 0) {         // If seconds reach 60, increment minutes
        minutes = (minutes + 1) % 60;

        if (minutes == 0) {     // If minutes reach 60, increment hours
            hours = (hours + 1) % 24;
            }
        }
}
```