Desenvolvimento do Firmware de Controle de um Elevador Controlado por Bluetooth

Guilherme Costa, Jordano dos Santos, Thalis Ianzer, Victor Cruz

Palavras-chave: Firmware, Eletrônica Embarcada, PIC16F, Microcontrolador, Periféricos.

1 INTRODUÇÃO

A eletrônica embarcada tem desempenhado um papel fundamental na automação de diversos sistemas, proporcionando maior eficiência e controle preciso em várias áreas da tecnologia moderna. Nesse contexto, o presente relatório apresenta o desenvolvimento de um firmware de controle para um elevador controlado por Bluetooth, utilizando o microcontrolador PIC16F1827. O objetivo principal deste trabalho foi implementar um firmware embarcado capaz de controlar um elevador de forma remota, utilizando o Bluetooth como interface de comunicação.

Fig. 1: Protótipo do elevador para testes

2 METODOLOGIA

O Porjeto foi desenvolvido em linguagem C, utilizando o ambiente de desenvolvimento (IDE) *MPLABX IDE* da microchip, a documentação foi gerada pelo DOxygen.

2.1 Divisão de Tarefas e Integração

Cada membro do grupo foi responsável por uma parte do projeto, desta forma, foi divido:

Guilherme Costa: responsável pela comunicação SPI e teste relativos;
Jordano dos Santos: responsável pelo controle do motor e teste relativos;
Thalis Ianzer: responsável pela aquisição e transformação de dados e teste relativos;
Victor Cruz: responsável pela comunicação UART e teste relativos;

Apesar das divisões todos os integrantes auxiliaram no desenvolvimento de outras partes conforme a pessoa responsável por ela fosse necessitando, por tanto todos os integrantes participaram do desenvolvimento do projeto como um todo.

Para que trabalhassemos de forma conjunta e organizada, optamos pelo uso da ferramenta de versionamento de códigos Git. Dessa forma, criamos um repositório no GitHub para hospedar nosso trabalho e suas versões, além disso, fizemos o uso de branchs para cada funcionalidade que seria implementada no projeto.

2.2 Sensores magneticos

Para identificar que o elevador alcançõu os andares utilizou-se as interrupções ligadas aos pinos do microcontrolador, para as funções vinculadas aos sensores S1 e S2 optou pelas interrupções IOC, por estarem conectados a pinos do *PORTB* que possuem esse periférico vinculado.Ja para os sesores S3 e S4, optou-se pela interrupções dos comparadores analógicos por estarem conectados ao *PORTA*.

Os sensores estão conectados a resistores de pull-up, interno a pic no caso de S1 e S2, por tanto todas as interrupções são ativadas em borda de descida.

2.3 Sensor de temperatura

Para garantir um bom funcionamento do motor, foi usado um sensor que mede a temperatura na ponte H que controla o motor. Esse valor é obtido e convertido de 0.0 à 99.9 antes de cada envio de dados por Bluetooth.

2.4 Comunicação SPI

A comunição foi desenvolvida em 3 partes: atualização da matriz de dados, conversão da matriz de dados para o formato exigido pela Matriz de *LED*, transmissão da Nova matriz por SPI. Sendo cada parte desenvolvida em uma função diferente.

2.4.1 Atualização Matriz de Dados

A matriz de dados é a variavel que continha os dados a serem enviados para a matriz de *LED*, que correspondem as colunas da matriz de *LED*, nela deveriam ser mostrado: o andar atual; O sentido de movimento do elevador; Os andares restantes no trajeto atual.
Para atualização dessa variavel utlizou-se as funções de interrupção dos sensores. Ao alcançar o andar a função de interrupção já atualiza todas as colunas da matriz de Dados e chama a função de conversão da matriz de Dados.

2.4.2 Conversão da Matriz de dados para o formato da Matriz de LED

Para transmissão da Matriz de Dados é necessário além de enviar além dos dados da coluna da Matriz, deve ser enviando também a posição da coluna de dados. Essa conversão é feita na função Matriz update, que envia envia para a função de transmissão a posição da coluna e o seu dados de forma unitária até enviar as 8 colunas.

```

void matrixUpdate(){
    uint8_t data[2];                    // Buffer
    para tx spi
    if (flip_matrix){                          // No
    Lab Remoto a imagem aparece invertida na
    horizontal
        for(uint8_t i=8;i>0;i--){          // Endere
    ?a digitos 7..0
            data[0] = i;                   // Digito
     i da Matriz
            data[1] = MatrixLed[i-1];    // Valor do
     digito i da Matriz
            txSpi(data, 2);         // Tx valores dos
     d?gitos dig para as matrizes
        }
    }else{
        uint8_t index = 7;                  //Indice
     da matrix de dados
        for(uint8_t i=1;i<9;i++){          // Endere
    ?a digitos 0..7

            data[0] = i;                   // Digito
     i da Matriz
            data[1] = MatrixLed[index];    // Valor
    do digito i da Matriz
            txSpi(data, 2);         // Tx valores dos
     d?gitos dig para as matrizes
            index--;
        }
    }
}
```

Listing 1: Código Atualização da matriz

2.4.3 Transmissão dos dados

A transmissão dos dados é feita a partir da função *SPI1 ExchangeBlock()* gerada biblioteca do *MPLABX IDE*, que recolhe os dados já formatados e o tamanho do vetor e envia para a matriz de LED por meio de SPI

```

void txSpi( uint8_t *data, size_t dataSize){
    CS_SetLow();             // Ativa CS
    SPI1_ExchangeBlock(data,dataSize);// Tx
    CS_SetHigh();           // Desativa CS
}
```

Listing 2: Código Transmissão de Dados

2.5 Comunicação UART

A comunicação EUSART é dividida em duas partes, uma de recepção de dados e outra de envio de dados. Ambos são feitos em ASCII. Toda comunicação está sendo feita com um Baud Rate de 19200 BPS, sendo, para a recepção, 1 byte de início ('$'), 2 bytes de dados (números de 0 a 3) e 1 byte de parada (0x0D). Para a acquisição desses dados foi feita uma máquina de estados como visto no código à seguir:

```

switch(state){
        case START:
            if(rxValue == '$'){
                state = FIRST_NUM;
            }
            break;
        case FIRST_NUM:
            if(isValidFloor(rxValue)){
                oTemp = rxValue - 0x30;
                state = SECOND_NUM;
            }else{
                state = START;
            }
            break;
        case SECOND_NUM:
            if(isValidFloor(rxValue)){
                dTemp = rxValue - 0x30;
                state = CR;
            }else{
                state = START;
            }
            break;
        case CR:
            if(rxValue == 0x0D){
                origem = oTemp;
                destino = dTemp;
                // Chamar alguma funcao
            }
            state = START;
            break;
        default:
            state = START;
    }
}
```

Listing 3: Código Transmissão de Dados

Sendo assim, ele apenas guarda as variáveis de origem e destino do elevador quando recebe o carriage return e na ordem certa, senão ele volta a aguardar o '$'.

Para o envio de dados, foi utilizado a interrupção do timer0 para que os dados fossem enviados a cada 300ms. Para que todos os dados fossem enviados corretamente, teve que manter sempre comunicação com quem estava fazendo as funções da interrupção do CCP4. Assim, foi possível fazer os cálculos da altura e da velocidade.

```
    altura = (1.5 * pulsoEncoder);  // 180mm/120
    pulsos
    velocidadeMotor = (abs(altura - aux_altura)/
    300.f) * 1000; // (mm/pulsos) / (tempo(s))
    aux_altura = altura;    // Salva a  ltima
    altura

    velocidade = (uint16_t)(velocidadeMotor * 10);
    // Ajustando o valor da velocidade para ser
    enviado
    temperatura = (ADC_GetConversion(2) / 1024.f) *
    999;    // Calcula a temperatura
```

Listing 4: Cálculo da altura e velocidade

]

A função que envia as informações também converte os números para BCD por meio da função *bin2bcd()* para que elas sejam enviadas em ASCII.

```
    void sendInfo(){

    bcd16_t bcd;
    uint16_t velocidade;

    velocidade = (uint16_t)(velocidadeMotor * 10);
    // Ajustando o valor da velocidade para ser
    enviado
    temperatura = (ADC_GetConversion(2) / 1024) *
    999; // Calcula a temperatura

    EUSART_Write('$'); // Caracter inicial
    EUSART_Write(0x30 + destino); // Envia o andar
    destino em ASCII
    EUSART_Write(0x2C); // Envia a virgula
    EUSART_Write(0x30 + andarAtual); // Envia o
    andar atual em ASCII
    EUSART_Write(0x2C); // Envia a virgula
    EUSART_Write(0); // Envia o estado atual do
    motor
    EUSART_Write(0x2C); // Envia a virgula
    // Enviando a altura em mm
    bcd.v = bin2bcd(altura);
    EUSART_Write(bcd.num2 + 0x30); // Envia o
    primeiro digito
    EUSART_Write(bcd.num3 + 0x30); // Envia o
    segundo digito
    EUSART_Write(bcd.num4 + 0x30); // Envia o
    terceiro digito
    EUSART_Write(0x2C); // Envia a virgula
    // Enviando a velocidade em mm/s
    bcd.v = bin2bcd(velocidade);
    EUSART_Write(bcd.num2 + 0x30); // Envia o
    primeiro digito
    EUSART_Write(bcd.num3 + 0x30); // Envia o
    segundo digito
    EUSART_Write(0x2E); // Envia o ponto
    EUSART_Write(bcd.num4 + 0x30); // Envia o digito
     decimal
    EUSART_Write(0x2C); // Envia a virgula
    // Enviando a temperatura em C
    bcd.v = bin2bcd(temperatura);
    EUSART_Write(bcd.num1 + 0x30); // Envia o
    primeiro digito
    EUSART_Write(bcd.num2 + 0x30); // Envia o
    segundo digito
    EUSART_Write(bcd.num3 + 0x30); // Envia o
    terceiro digito
    EUSART_Write(0x2E); // Envia o ponto
    EUSART_Write(bcd.num4 + 0x30); // Envia o digito
     decimal
    EUSART_Write(0x0D); // Envia o carriage return
}
```

Listing 5: Envio das informações do elevador

2.6 Controle do Motor

O controle do motor foi feito a partir de uma ponte H, que por meio do microcontrolador recebia o sinal de PWM e um valor binário para controlar sua direção. O sinal do PWM foi definido para operar na frequência de 3.9kHz e com ciclo útil de 50%. O acionamento do motor foi feito por meio da Interrupção do *Timer 4*, com período de 500ms. O acionamento foi otimizado por meio do ordenamento dos andares de destino e origem, para que o elevador faça o trajeto da maneira mais eficaz possível.

3 Simulações

3.1 Transmissão SPI

A transmissão SPI não pode ser simulada pelo MPLABX IDE por tanto para testa-la foi adcionado a main a chamada das funções de interrupção dos sensores separadas por um *delay* para ser possivel a observação de cada estado isoladamente

```
    while (1)
    {
        // Add your application code
        chegadaS1();
        __delay_ms(1000);
        chegadaS2();
        __delay_ms(1000);
        chegadaS3();
        __delay_ms(1000);
        chegadaS4();
    }
```

Listing 6: Código Atualização da matriz

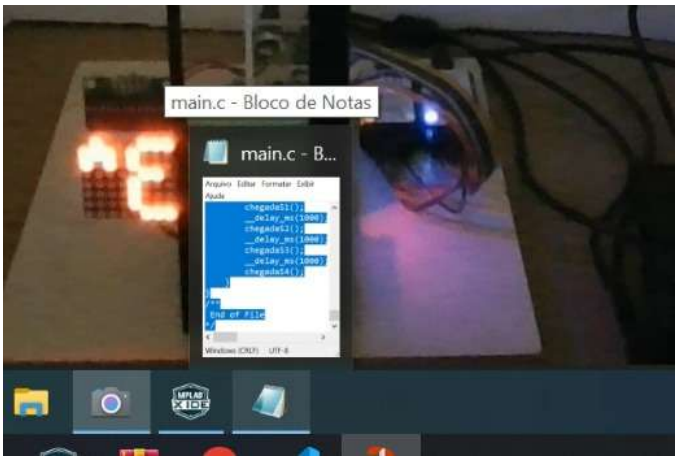

Fig. 2: Teste SPI (função chegadaS4())

A figura 2 mostra o momento a transmissão dos dados a partir da função *chegadaS4()*, acionada quando o elevador chega no último andar, exemplificando o funcionamento da matriz de *LED*

3.2 Transmissão USART e tratamento de dados

Para simular o recebimento de dados utilizou-se o estimulo síncrono *register injection* e para validação se os dados estavam sendo adquiridos corretamente utilizou-se o watches RCREG que guarda o byte de entrada do RX.

Para simulação da Transmissão utilizou-se a janela watches, na qual se alterava o valor das variáveis a serem transmitidas manualmente se observava see elas entavam sendo transmitadas corretamente a partir do registrador TXREG, que armazena o valor a ser transmitido pelo pino TX.

A figura 4 mostram os registradores sendo alterados durante a simulação

3.3 Controle Motor

Para simulação do controle do motor foi utilizado o modo debug do MPLAB sendo as entradas simuladas por estimulos. Para tal foram utilizados os estímulos síncronos

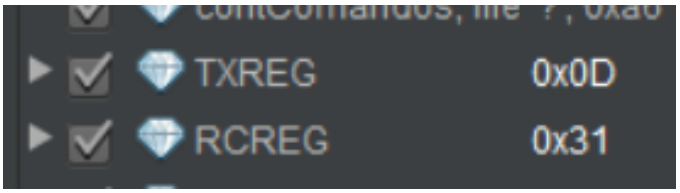

Fig. 3: Registradores RCREG e TXREG no tempo 1

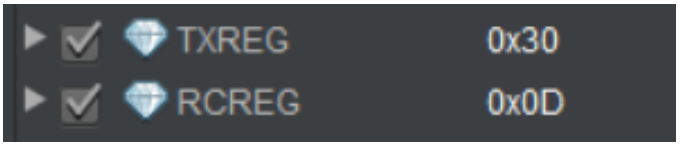

Fig. 4: Registradores RCREG e TXREG no tempo 2

do *register injection* para simular as entradas do USART e os estímulos assíncronos de pulso em 0 para símular as entradas dos sensores.

A validação dos resultados foi feita a partir da janela *Logic Analyser* do MPLAB, que permite observar em função do tempo a variação do sinal dos registradores internos da Pic. Caso o registrador CCP3, ligado ao pino RA3, estivesse gerando a onda PWM assumisse que o motor estaria se movendo, ao mesmo tempo caso o pino RA7 estivesse em nível lógico alto, esse movimento seria ascendente, e se estivesse em baixo, descentente.

Para validação das paradas do elevador observou-se pela janela *watches* se as variáveis de controle de trajeto a **destinoDesc** e **destinoSub** estavam sendo apagadas corretamente ao passar pelo andares que escolhidos e se o PWM estava sendo interrompido e sendo reacionado corretamente. Após essas condições serem compridas, pode-se assumir que o motor estava realizando os trajetos corrente.

A figura 5 demonstra o comportamento do PWM e do pino de direção ao atingir a interrupção do pino S3 e alterar seu movimento para descida, parando o motor por um tempo e reativando ele com o sentido descentente.

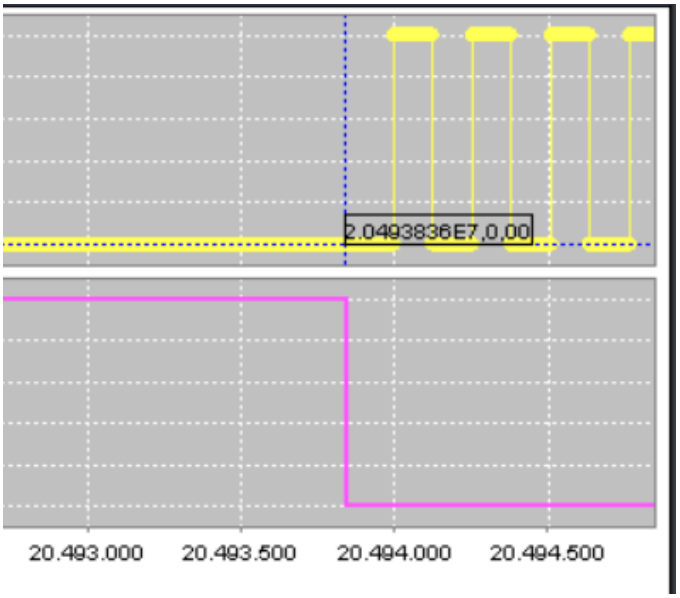

Fig. 5: inciando descida

4 Conclusão

A partir das simulações o elevador funcionou da forma esperada, cumprindo todos os requisitos solicitados pelo roteiro, enviando e recebendo dados, calculando o trajeto a se fazer e respeitando os tempos de parada. Ao aplicar os estímulos assíncronos simulando as chegadas as funções apagaram os destinos corretamente e definiram a direção correta de movimento do motor, podendo-se assim assumir que na implementação do código na *Pic16F1827* ocorreu como o desejado.

Appendix A

Diretório do firmware: https://github.com/Projeto-final-EE

Appendix B

Referencias Bibliográficas

url:http://ww1.microchip.com/downloads/en/devicedoc/41391d.pdf

References

[1] http://www.newtoncbraga.com.br/index.php/robotica/5169-mec071a
[2] https://toshiba.semicon-storage.com/eu/semiconductor/product/motor-driver-ics/brushed-dc-motor-driver-ics/detail.TB6612FNG.html
[3] http://www.newtoncbraga.com.br/index.php/artigos/54-dicas/6083-art762.html
[4] https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf
[5] https://www.asciitable.com/
[6] https://www.olimex.com/Products/Components/RF/BLUETOOTH-SERIAL-HC-06/resources/hc06.pdf