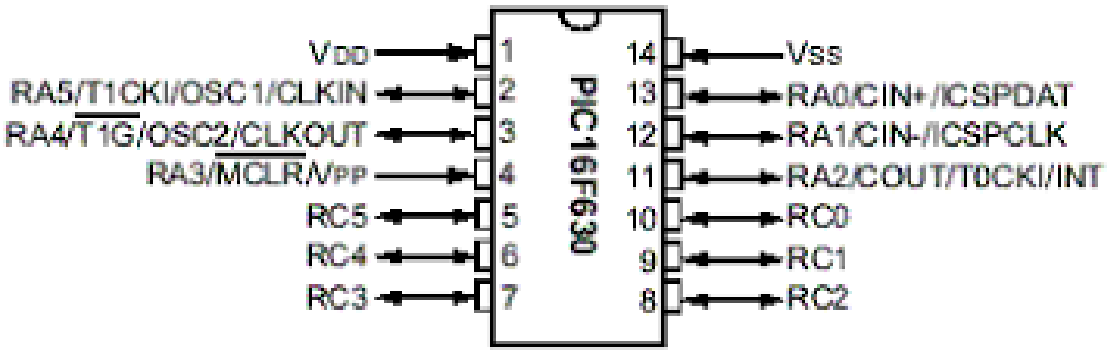
PLANO DE AULA

PIC16F630:

O PIC16F630, fabricado pela Microchip, é um microcontrolador de 8 bits amplamente utilizado devido à sua simplicidade, custo acessível e versatilidade. Ele é ideal para projetos que envolvem entrada e saída de dados digitais, lógica de programação básica e até mesmo funcionalidades mais avançadas, como temporização e interrupções. A seguir, destacamos os aspectos mais relevantes do microcontrolador, incluindo suas características de pinagem.



Assim como outros modelos da série PIC, o PIC16F630 possui uma disposição de pinos que oferece funcionalidades diversas. Abaixo, explicamos os principais pinos e suas utilidades com base no que será comumente utilizado:

- **VDD (Alimentação):** Este pino é responsável por receber a fonte de alimentação do microcontrolador, que normalmente opera com uma tensão de 5V. Ele fornece a energia necessária para o funcionamento interno do dispositivo.
- **VSS (GND – Ground):** O pino de referência de terra (ground), essencial para completar o circuito elétrico e fornecer um referencial de tensão estável ao microcontrolador.
- **RA (PORTA) e RC (PORTC) – Pinos de Entrada e Saída de Dados:**

    - Esses conjuntos de pinos (PORTA e PORTC) são dedicados à comunicação de dados e podem ser configurados como entrada ou saída digital, dependendo da aplicação.
    - Cada pino possui funções específicas que podem ser habilitadas ou configuradas por meio de registradores internos.
    - Além das funções digitais, alguns desses pinos podem desempenhar papéis adicionais, como entradas analógicas, saídas de PWM (modulação por largura de pulso) ou sinais de clock.

Características Básicas do PIC16F630

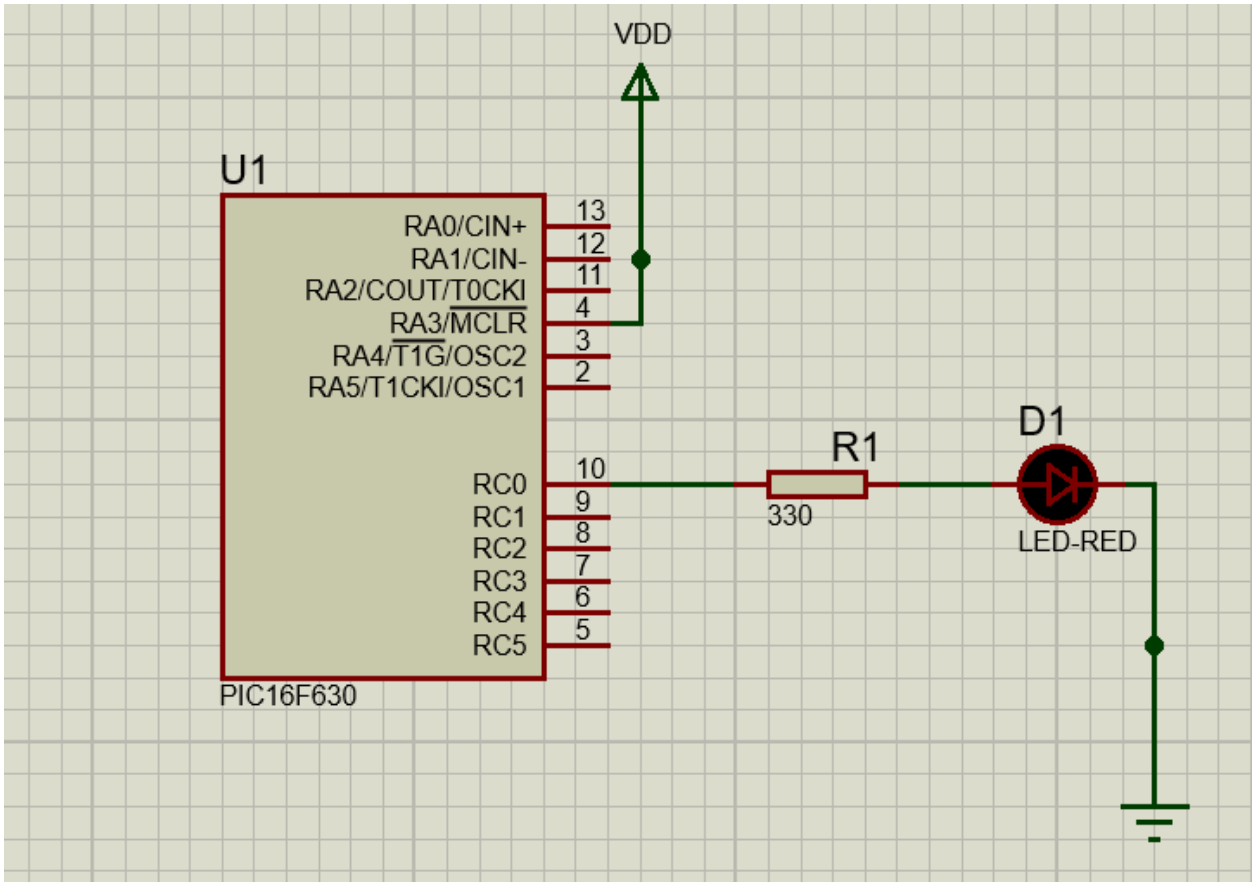
O PIC16F630 é um microcontrolador compacto, mas potente, com as seguintes características principais:

- **Clock Interno:** Possui um oscilador interno configurável, eliminando a necessidade de componentes externos para geração de clock em muitos projetos.
- **Memória:** Integra memória flash para armazenamento do programa, memória RAM para dados voláteis e EEPROM para dados permanentes.
- **I/O Versátil:** Com 12 pinos disponíveis para entrada e saída de dados, o microcontrolador oferece flexibilidade para diversos projetos.
- **Periféricos:** Inclui temporizadores, comparadores analógicos e outros recursos que ampliam suas possibilidades de uso.
- **Baixo Consumo de Energia:** Pode operar com eficiência em sistemas alimentados por baterias.

  Link datasheet:
  https://ww1.microchip.com/downloads/en/devicedoc/40039c.pdf

Exercícios:

Projeto: Ligar o LED



Para iniciarmos nosso primeiro projeto com microcontroladores, é essencial compreender os comandos básicos, especialmente os relacionados aos registradores **TRIS** e **PORT**. Na IDE utilizada, o bloco principal do programa está contido na função `void main`, onde implementaremos nosso código.

Configurando a Direção do Pino

Se nosso objetivo for acender um LED, precisamos que o microcontrolador envie uma tensão para um pino específico. Vamos escolher, por exemplo, o pino **RC0** do grupo de pinos C.

O primeiro passo é configurar este pino como **saída de dados**. Para isso, utilizamos o comando **TRIS**, que determina a direção dos dados na porta. Ao definir **TRISC = 0**, configuramos **todos os pinos do grupo C** como saída. Contudo, se quisermos configurar apenas o **pino RC0**, podemos usar o modificador `bits`. Dessa forma, utilizamos o comando:

- TRISCbits.RC0 = 0;

Aqui:

- `TRISCbits` nos permite configurar um único pino do grupo C.
- `.RC0` indica que estamos configurando especificamente o pino RC0.
- `= 0` define o pino como saída (se fosse `= 1`, ele seria configurado como entrada).

Controlando o Nível Lógico

Após definir a direção do pino, podemos configurar seu **nível lógico**. Para acender o LED conectado ao pino **RC0**, utilizamos o comando:

- PORTCbits.RC0 = 1;

Essa linha faz com que o **pino RC0** seja colocado em nível lógico **alto** (HIGH), enviando tensão para o LED e acendendo-o.

Montagem do Circuito

Para proteger o LED, conectamos um resistor de **220 Ohms** em série com ele. O resistor reduz a corrente que passa pelo LED, evitando danos causados pela alta tensão do microcontrolador. No Proteus, o PIC já é alimentado internamente, mas ao montar na protoboard, lembre-se de fornecer alimentação ao circuito.

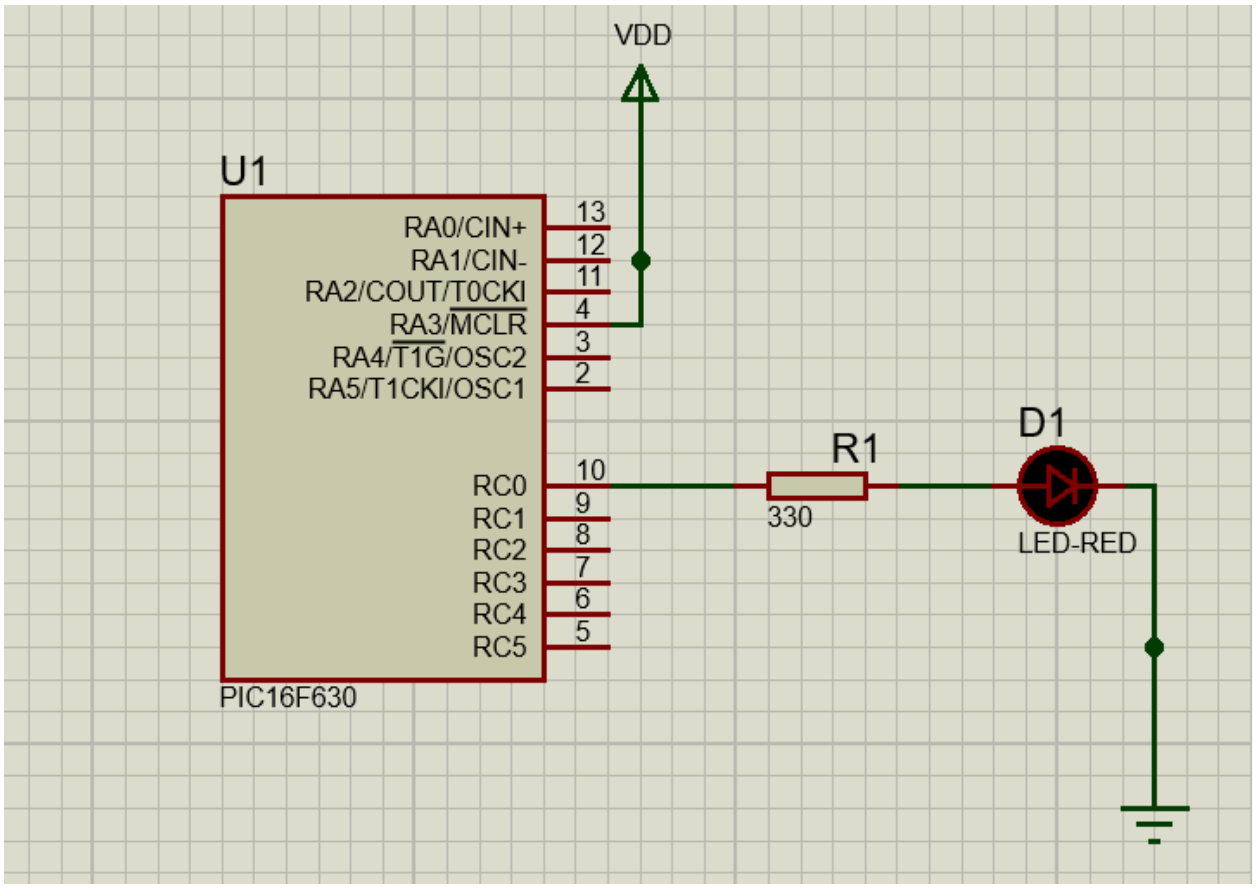
Resumo do Código

Abaixo está o código básico para configurar o pino RC0 como saída e acender um LED:

```
#include <xc.h>

#pragma config FOSC = INTRCCLK
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = ON
#pragma config BOREN = OFF
#pragma config CP = OFF
#pragma config CPD = OFF
```

```
#define botao PORTCbits.RC5

#define _XTAL_FREQ 4000000
void main(void) {
    TRISC = 0b000000;
    PORTC = 0x00;
    PORTC = 0b01;

    while(1){
   }
}
```

Projeto: Pisca Led



Após o sucesso em nosso primeiro projeto, onde acendemos um LED, é hora de avançarmos para a **Parte 2**: fazer o LED **piscar**. Vamos introduzir alguns conceitos adicionais, explicando cada passo de forma detalhada.

Configurando a Frequência do Microcontrolador

Antes de tudo, precisamos definir a frequência na qual nosso microcontrolador opera. Isso já foi configurado no *header* do código, mas também precisamos especificar esse valor em uma variável para que certas funções funcionem corretamente.

Como nosso microcontrolador possui um **oscilador interno de 4 MHz**, adicionamos a linha abaixo no início do código:

- #define _XTAL_FREQ 4000000

Essa definição será usada para calcular os tempos de delay no programa.

Estrutura do Código

A lógica do programa será organizada em duas partes principais:

1. **Setup (Configurações Iniciais)**: Configuramos o pino do LED como saída e definimos outras configurações iniciais.
2. **Loop Principal**: Dentro de um loop infinito, a lógica do programa será executada continuamente.

Utilizaremos o seguinte padrão:

- Tudo o que vem **antes do** `while(1)` será usado para configurar o microcontrolador.
- Tudo o que está **dentro do** `while(1)` representará a lógica de execução do programa.

Fazendo o LED Piscar

Para fazer o LED piscar, alternaremos entre ligar e desligar o pino **RC0**:

- **Ligar o LED**: `PORTCbits.RC0 = 1`
- **Desligar o LED**: `PORTCbits.RC0 = 0`

Entretanto, como o microcontrolador executa as instruções rapidamente, não seria possível perceber o LED piscando. Por isso, utilizaremos a função `__delay_ms()` para criar uma pausa entre cada mudança de estado.

Para que o LED pisque a cada **2 segundos**, faremos:

-

Como Funciona o Delay?

A função `__delay_ms()` depende da constante `_XTAL_FREQ` para calcular o tempo de pausa com base na frequência do oscilador. Sem essa definição, a função não funcionará corretamente.

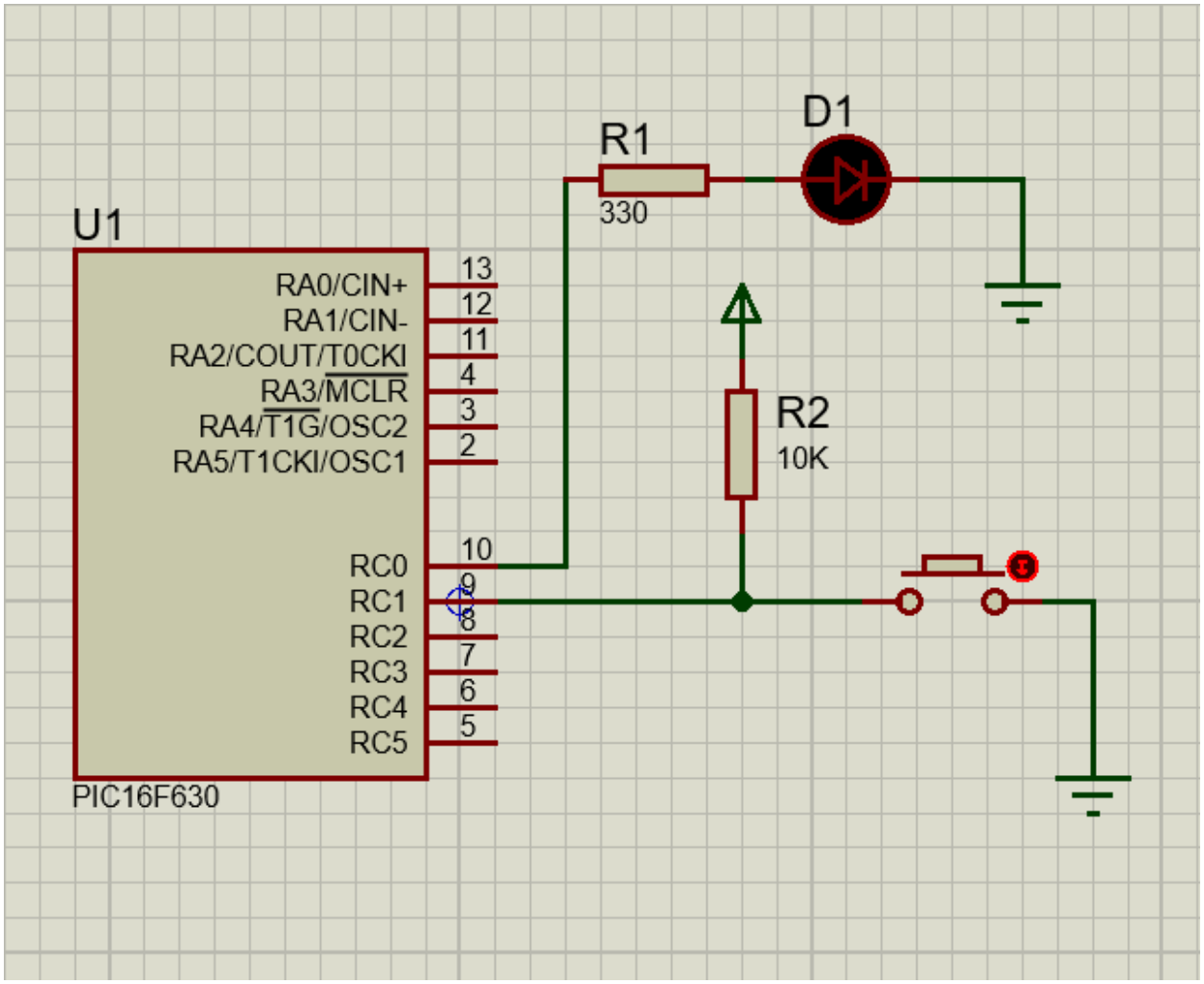
Montagem no Circuito

Certifique-se de conectar um resistor de **220 Ohms** em série com o LED para protegê-lo. Com o código e o circuito configurados corretamente, o LED deve piscar em intervalos de 2 segundos.

```
#include <xc.h>

#pragma config FOSC = INTRCCLK
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = ON
#pragma config BOREN = OFF
#pragma config CP = OFF
#pragma config CPD = OFF
#define botao PORTCbits.RC5

#define _XTAL_FREQ 4000000
```

```
void main(void) {
    TRISC = 0b000000;
    PORTC = 0b000001;

    while(1){
   //codigo

   }
}
```

Projeto: Input de Botões



Texto Explicativo: Entrada com Botão (Pull-up)

Neste projeto, implementaremos um circuito simples para acionar um LED utilizando um botão configurado com **resistor de pull-up**. Esse tipo de configuração garante que o pino do microcontrolador esteja em nível lógico **alto (1)** quando o botão não estiver pressionado, e em nível lógico **baixo (0)** quando pressionado.

Entendendo o Circuito e o Código

Resistor de Pull-up

- O resistor de pull-up é utilizado para manter o pino do botão em nível **alto (1)** enquanto o botão não é pressionado.
- Quando o botão é pressionado, o pino é conectado ao **GND**, assumindo nível **baixo (0)**.

Configuração do Código

1. **Definição dos Pinos**
    - O botão será conectado ao pino **RC1** (entrada).
    - O LED será conectado ao pino **RC0** (saída).
2. **Configuração da Direção dos Pinos**
    - Para configurar um pino como entrada ou saída, usamos o comando `TRIS`.
        - `TRISCbits.TRISC1 = 1`: Configura o pino **RC1** como entrada (para o botão).
        - `TRISCbits.TRISC0 = 0`: Configura o pino **RC0** como saída (para o LED).
3. **Lógica do Botão**
    - Com o resistor de pull-up, quando o botão não está pressionado, o pino **RC1** estará em nível **alto (1)**.
    - Quando o botão é pressionado, o pino **RC1** muda para nível **baixo (0)**.
    - Utilizamos a condição `if (!BUTTON1)` para detectar quando o botão está pressionado.

4. **Controle do LED**
    - Se o botão estiver pressionado (nível lógico 0), o LED será aceso (`LED = 1`).
    - Caso contrário, o LED permanecerá apagado (`LED = 0`).

Funcionamento no Circuito

1. **LED Inicialmente Desligado**
   Ao inicializar o código, o LED estará apagado.

2. **Botão Pressionado**
   Quando o botão conectado ao pino **RC1** for pressionado, o LED acenderá.

3. **Botão Solto**
   Ao soltar o botão, o LED apagará novamente.

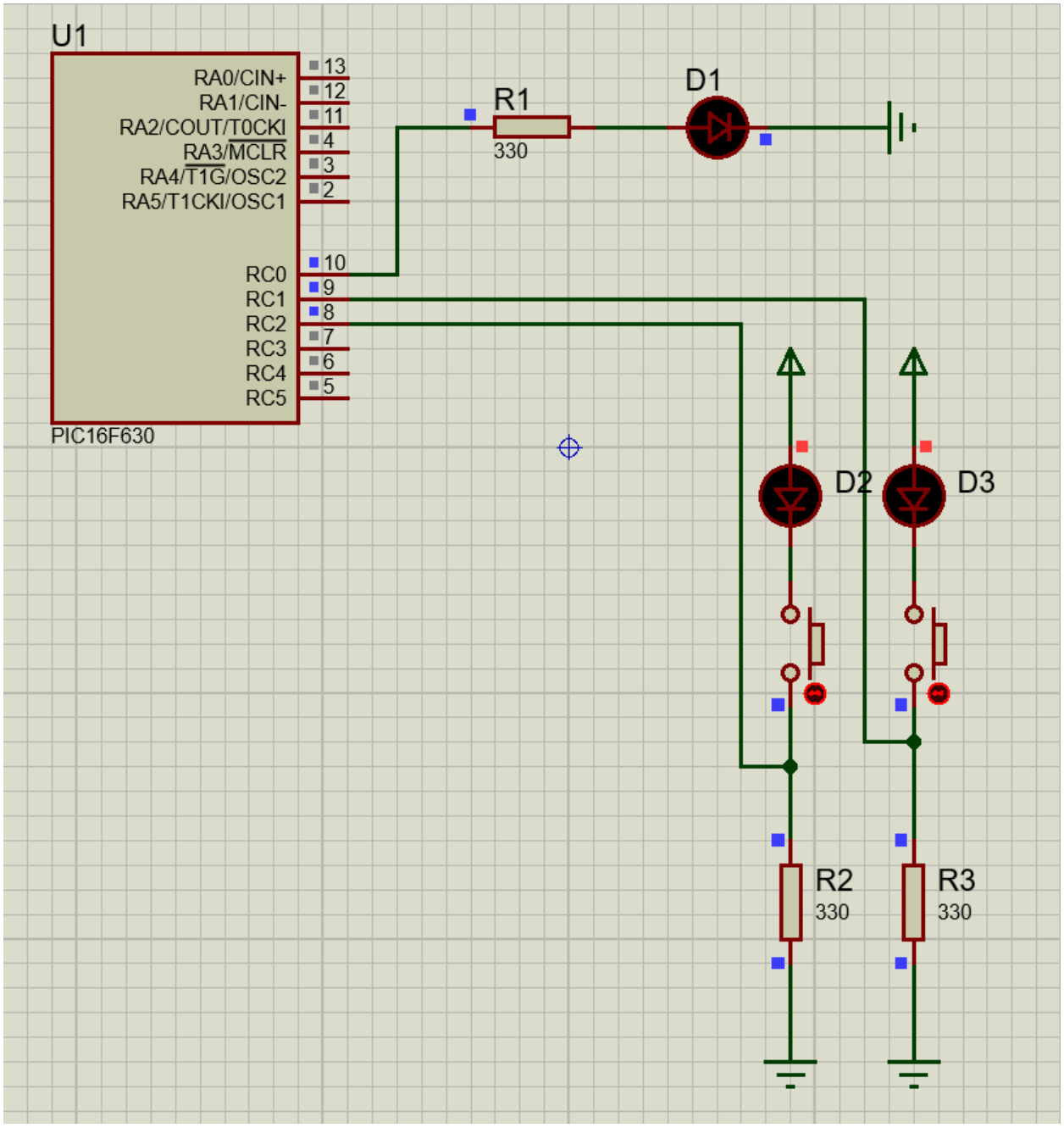
Dicas para Prototipagem

- Certifique-se de utilizar um resistor de pull-up, interno ou externo.
- Um resistor de **220 Ohms** deve ser conectado em série com o LED para proteger o componente de sobrecorrente.
- Conecte o botão de forma que um terminal esteja ligado ao pino **RC1** e o outro ao **GND**.

```
#define BUTTON1 PORTCbits.RC1  // Botão conectado ao pino RC1
#define LED     PORTCbits.RC0  // LED conectado ao pino RC0

void main() {
   // Configuração inicial
```

```
    TRISCbits.TRISC1 = 1;  // Configura RC1 como entrada (botão)
    TRISCbits.TRISC0 = 0;  // Configura RC0 como saída (LED)
    LED = 0;               // Inicializa o LED desligado

    // Loop principal
    while (1) {
        if (!BUTTON1) {    // Verifica se o botão está pressionado (nível 0)
            LED = 1;       // Liga o LED
        } else {
            LED = 0;       // Desliga o LED
        }
    }
}
```

Projeto: AND com Input de Botões



Texto Explicativo: Lógica AND com Dois Botões

Neste projeto, vamos explorar o funcionamento de dois botões conectados ao microcontrolador. O objetivo é acender um LED somente quando **ambos os botões** estiverem pressionados simultaneamente, utilizando a lógica **AND**.

Funcionamento do Circuito

1. **Conexão dos Botões**
    - O botão **1** está conectado ao pino **RC1**.
    - O botão **2** está conectado ao pino **RC2**.
    - Ambos os botões estão configurados com resistores de pull-up. Isso significa que os pinos estarão em nível lógico **alto (1)** quando os botões não estiverem pressionados e em nível lógico **baixo (0)** quando pressionados.
2. **LED**
    - O LED está conectado ao pino **RC0**, que será configurado como saída.

Configuração do Código

1. **Direção dos Pinos**
    - `TRISCbits.TRISC1 = 1`: Configura o pino **RC1** como entrada para o botão 1.
    - `TRISCbits.TRISC2 = 1`: Configura o pino **RC2** como entrada para o botão 2.
    - `TRISCbits.TRISC0 = 0`: Configura o pino **RC0** como saída para o LED.
2. **Lógica AND**
    - Utilizamos a expressão `if (BUTTON1 && BUTTON2)` para verificar se ambos os botões estão pressionados.
    - A lógica AND exige que **ambas as condições sejam verdadeiras** para que o LED seja acionado.
3. **Controle do LED**
    - Se **BUTTON1** e **BUTTON2** estiverem em nível lógico **alto (1)**, o LED será aceso (`LED = 1`).
    - Caso contrário, o LED permanecerá apagado (`LED = 0`).

Funcionamento no Circuito

1. **LED Inicialmente Desligado**
    - Quando nenhum dos botões ou apenas um deles está pressionado, o LED permanece apagado.
2. **Botões Pressionados**
    - O LED será aceso **somente** se ambos os botões estiverem pressionados ao mesmo tempo.

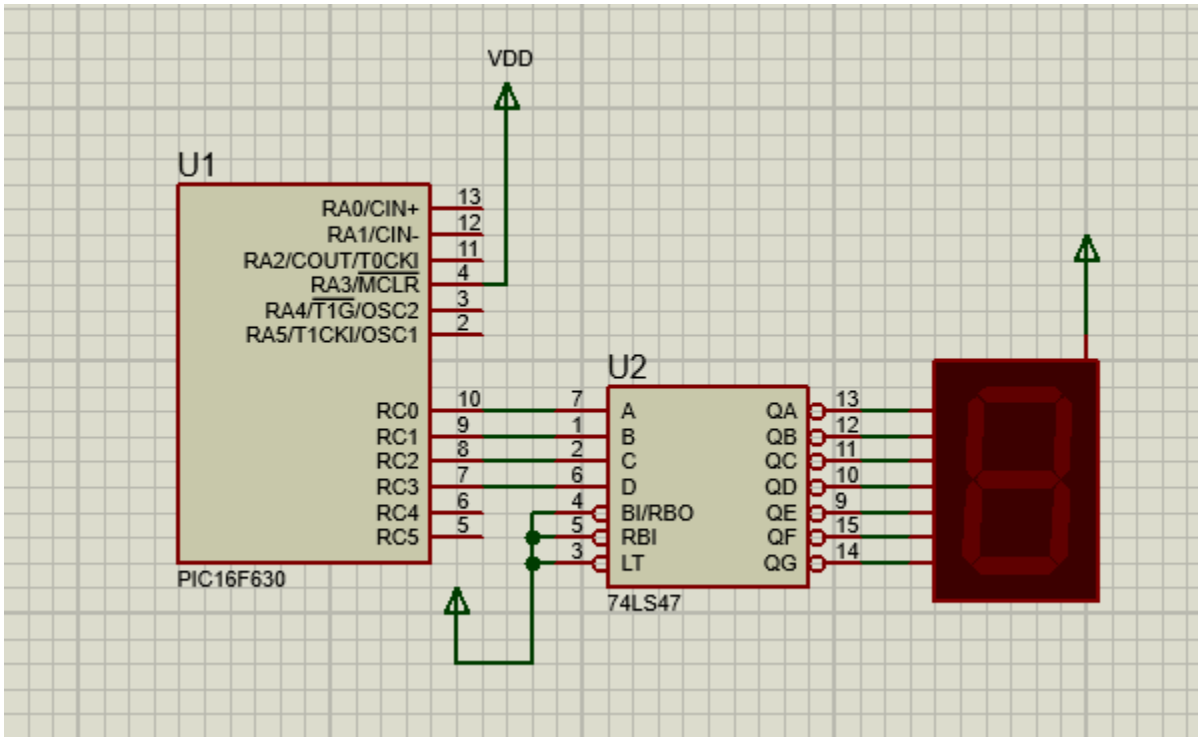
Dicas de Montagem

1. **Resistores de Pull-up**
    - Certifique-se de usar resistores de pull-up nos pinos dos botões, internos ou externos, para garantir um comportamento correto.
2. **Resistor para o LED**
    - Utilize um resistor de **220 Ohms** em série com o LED para limitar a corrente e evitar danos ao componente.
3. **Teste Progressivo**
    - Teste individualmente os botões antes de verificar a lógica AND para garantir que o hardware está corretamente conectado.

```
#define BUTTON1 PORTCbits.RC1 //definindo macro para o botao 1
#define BUTTON2 PORTCbits.RC2 //definindo macro para o botao 2
#define LED     PORTCbits.RC0     //definido macro para o estado do led de saída

void main() {

   TRISCbits.TRISC1 = 1; // RC1 configurado como entrada (botão 1)
   TRISCbits.TRISC2 = 1; // RC2 configurado como entrada (botão 2)
   TRISCbits.TRISC0 = 0; // RC0 configurado como saída (LED)
```

```
    LED = 0; // LED desligado inicialmente

    while (1) {
        if (BUTTON1 && BUTTON2) {//botao 1 e botao 2 em HIGH
            LED = 1; // Acende o LED
        } else {
            LED = 0; // senao led permanece apagado
        }
    }
}
```

Projeto: For com 7segmentos



Neste código, o objetivo é utilizar um microcontrolador PIC para exibir uma contagem de 0 a 9 em um display de 7 segmentos, controlado pelo driver **74LS47**. A contagem é feita enviando valores binários às entradas A, B, C e D do driver, que converte essas entradas para os segmentos do display.

Passo a Passo do Código

1. **Definição da Frequência**
    - O comando `#define _XTAL_FREQ 4000000` define a frequência de operação do oscilador interno do microcontrolador como 4 MHz. Essa

definição é essencial para o funcionamento correto de funções como `__delay_ms()`, que depende dessa frequência para calcular os intervalos de tempo.

2. **Configuração dos Pinos**
    - `TRISC = 0x00`: Configura todos os pinos do PORTC como **saídas digitais**. Isso é necessário porque os pinos do PORTC irão enviar os valores binários para o **74LS47**.
    - `PORTC = 0x00`: Inicializa todos os pinos do PORTC em nível lógico baixo, garantindo que o display comece apagado.
3. **Loop Principal**
    - `while (1) { }`: Este é o laço infinito que mantém o programa em execução contínua, típico de aplicações em microcontroladores.
    - Dentro do loop, há um **for** que percorre os números de `0` a `9`:
        - `PORTC = i;`: A variável `i` (do tipo `unsigned char`) representa os números de 0 a 9. A cada iteração, seu valor é enviado ao PORTC. Como o PORTC está conectado às entradas A, B, C e D do **74LS47**, o driver interpreta esses valores e exibe o número correspondente no display.
        - `__delay_ms(200);`: Após cada valor ser enviado ao PORTC, o programa aguarda 200 milissegundos antes de passar para o próximo número. Esse delay permite que o número fique visível no display por um curto período.
4. **Conexão com o Driver 74LS47**
    - O driver **74LS47** converte os valores binários enviados pelo PORTC para os segmentos do display. Ele é projetado para trabalhar com displays de **cátodo comum**, ativando os segmentos correspondentes ao número.

Resumo do Funcionamento

O microcontrolador envia valores binários para o **74LS47**, de 0 a 9, continuamente. Esses valores controlam o driver, que exibe os números no display de 7 segmentos. O delay de 200 ms cria uma transição visível entre os números, resultando em uma contagem progressiva de 0 a 9 no display.

Ajustes e Testes

- Para aumentar o intervalo entre os números, você pode modificar o valor do `__delay_ms()` para um tempo maior (por exemplo, `__delay_ms(1000)` para 1 segundo).

- Certifique-se de que as conexões elétricas entre o **74LS47**, o display e o microcontrolador estejam corretas, com resistores limitadores de corrente entre o driver e o display.

Código:

```
#define _XTAL_FREQ 4000000  // Frequência do oscilador interno (4 MHz)

void main() {
   // Configuração dos pinos do PORTC como saída
   TRISC = 0x00;  // Configura todos os pinos do PORTC como saída
   PORTC = 0x00;  // Inicializa todos os pinos do PORTC em nível baixo

   // Loop principal
   while (1) {
      for (unsigned char i = 0; i < 10; i++) {
         PORTC = i;        // Envia o valor binário para o PORTC (A, B, C, D)
         __delay_ms(200);  // Aguarda 1 segundo antes de incrementar
      }
   }
}
```

Projeto: Semáforo

Explicação do Código: Exibição de Valores Personalizados no Display de 7 Segmentos

Este código utiliza um microcontrolador PIC para exibir diferentes padrões no display de 7 segmentos, baseado em valores definidos em um contador que incrementa de 0 a 15. Os padrões exibidos são controlados pelo **74LS47**, e cada valor do contador corresponde a uma configuração específica dos segmentos do display.

Passo a Passo do Código

1. **Configurações do PIC**

- O microcontrolador é configurado para operar com seu oscilador interno de 4 MHz (`FOSC = INTRCIO`).
- O `Watchdog Timer` e outras funções que não são necessárias (como o `Brown-out Reset`) estão desativadas.
- `#define _XTAL_FREQ 4000000`: Define a frequência do oscilador interno, necessária para calcular os delays no programa.

2. **Configuração Inicial**

   - `ANSEL = 0`: Configura todos os pinos como digitais. Isso é essencial, pois os pinos podem ser configurados como entradas analógicas por padrão em alguns microcontroladores.
   - `TRISC = 0`: Configura todos os pinos da porta C como **saídas**.
   - `PORTC = 0`: Inicializa todos os pinos da porta C em nível lógico baixo.

3. **Lógica Principal**

   - A variável `contador` é utilizada para controlar os padrões exibidos no display.
   - Um **switch case** define os valores enviados ao PORTC para determinados valores do contador:
     - `case 0`: Envia o valor `0b00100001` para o PORTC.
     - `case 5`: Envia o valor `0b00100010` para o PORTC.
     - `case 7`: Envia o valor `0b00001100` para o PORTC.
     - `case 12`: Envia o valor `0b00010100` para o PORTC.
     - `default`: Para outros valores de `contador`, nenhuma ação é tomada.
   - `__delay_ms(500)`: Aguarda 500 ms entre as mudanças no contador, criando uma pausa visível na transição.

4. **Controle do Contador**

   - O contador é incrementado continuamente (`contador++`).
   - Quando o contador atinge o valor 15 (`if(contador == 15)`), ele é reiniciado para 0.

Resumo do Funcionamento

- O código faz o display exibir diferentes padrões para os valores 0, 5, 7 e 12 do contador. Esses padrões são definidos pelos valores binários enviados ao PORTC.
- O **74LS47** interpreta os valores enviados ao PORTC e controla os segmentos do display para exibir os padrões correspondentes.
- Um delay de 500 ms entre as atualizações do contador permite visualizar as mudanças no display.

Possíveis Ajustes

1. **Alterar os Padrões:**
    - Para exibir outros números ou padrões, basta modificar os valores no `switch case` para o padrão binário correspondente.
2. **Alterar o Tempo de Delay:**
    - Modifique o valor de `__delay_ms(500)` para ajustar a velocidade de mudança dos padrões no display.
3. **Adicionar Mais Casos:**
    - Você pode expandir o `switch case` para incluir outros valores do contador e controlar o comportamento do display de maneira mais personalizada.

Observações

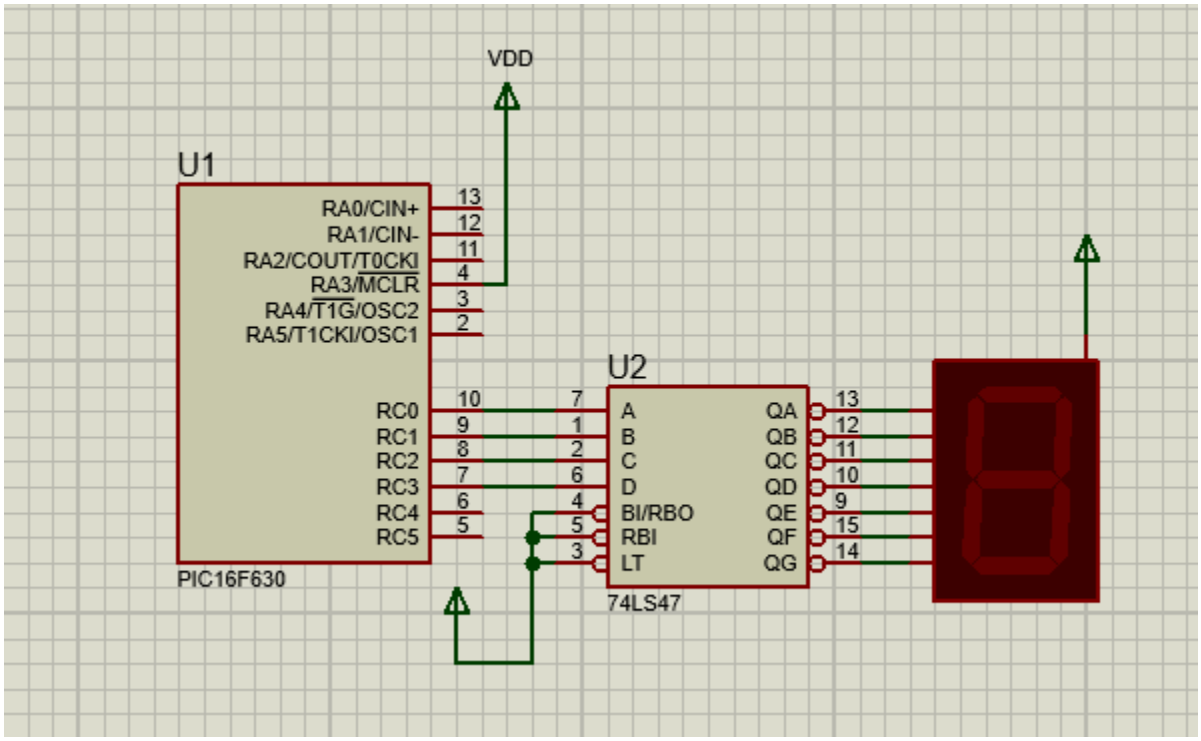
- Certifique-se de que o display de 7 segmentos e o driver **74LS47** estejam corretamente conectados ao PORTC.
- Verifique que os valores binários definidos no `switch case` estão alinhados com o comportamento desejado para os segmentos do display.

Código:

```
#include <xc.h>

// Configuração do PIC (Erros inseridos propositalmente)
#pragma config FOSC = XT       // Oscilador externo (deveria ser interno)
#pragma config WDTE = ON       // Watchdog Timer ligado (pode causar resets indesejados)
```

```
#pragma config PWRTE = OFF     // Power-up Timer desligado (pode causar problemas na inicialização)
#pragma config MCLRE = ON      // Pino MCLR ativado (não está configurado corretamente no hardware)
#pragma config CP = ON         // Proteção de código ativada (pode impedir gravação)
#pragma config CPD = ON        // Proteção de dados ativada (pode causar erro ao gravar EEPROM)
#pragma config BOREN = ON      // Brown-out Reset ligado (pode gerar resets inesperados)

#define XTAL_FREQ 4000000  // Erro: definição incorreta (faltou o underscore inicial)

void main(void) {
    // Configuração inicial (erros intencionais)
    ANSEL = 0xFF; // Configura todos os pinos como analógicos (deveriam ser digitais)
    TRISC = 1;    // Configura todos os pinos da porta C como entrada (deveriam ser saídas)
    PORTC = 0xFF; // Inicializa todos os pinos da porta C em nível alto (deveria ser baixo)

    unsigned int contador;  // Variável não inicializada (valor indefinido)

    while (1) {
        switch (contador) {
        case 0:
            PORTC = 0b00100001;
            break;
        case 5:
            PORTC = 0b00100010;
            break;
        case 7:
            PORTC = 0b00001100;
            break;
        case 12:
            PORTC = 0b00010100;
            break;
        default:
            break;
        }
        __delay_ms(500);  // Erro: diretiva de atraso pode não funcionar devido à definição incorreta do oscilador
        contador++;       // Incremento sem verificação adequada
        if (contador > 15) contador = 0;  // Condição desnecessária, deve ser "== 15"
    }
}
```

Projeto: Semáforo com Timers



O Papel dos Temporizadores no PIC16F630

Ao explorar as funcionalidades do microcontrolador PIC16F630, destacam-se os temporizadores, que desempenham um papel essencial no controle de eventos temporais de maneira eficiente. Vamos detalhar como os temporizadores operam e sua importância em aplicações práticas.

Os temporizadores são usados para contar ciclos de clock ou pulsos externos. Eles permitem a execução de tarefas em intervalos regulares sem interromper completamente o funcionamento do programa principal, diferentemente do atraso gerado pela função `__delay_ms()`. Essa característica os torna ideais para sistemas que precisam executar várias tarefas simultaneamente, como controle de LEDs, semáforos ou dispositivos mais complexos.

No **PIC16F630**, o temporizador TMR0 é amplamente utilizado. Ele é um registrador de 8 bits que pode contar até 255 antes de emitir um sinal de estouro. O **prescaler** associado ao TMR0 ajusta a taxa de contagem, proporcionando flexibilidade para atender diferentes requisitos temporais. O estouro do temporizador pode ser configurado para acionar uma interrupção, permitindo que o microcontrolador execute ações específicas no momento certo.

Vantagens do Uso de Temporizadores

- Permitem a execução de tarefas em paralelo com o programa principal.
- Fornecem maior precisão para controle de tempo.
- São ideais para aplicações que exigem intervalos regulares, como a troca de estados em um semáforo.

Código do Semáforo com Temporizador (PIC16F630)

O código abaixo implementa a lógica de um semáforo usando o **Timer0** do PIC16F630. Ele alterna entre diferentes estados com base na contagem do temporizador.

```
#include <xc.h>

// Configuração do PIC
#pragma config FOSC = INTRCIO   // Oscilador interno
#pragma config WDTE = OFF       // Watchdog Timer desligado
#pragma config PWRTE = ON       // Power-up Timer ligado
#pragma config MCLRE = OFF      // Pino MCLR como entrada digital
#pragma config CP = OFF         // Proteção de código desligada
#pragma config CPD = OFF        // Proteção de dados desligada
#pragma config BOREN = OFF      // Brown-out Reset desligado
#define _XTAL_FREQ 4000000      // Frequência do oscilador interno

// Variáveis globais
unsigned int contador = 0;

// Configuração inicial
void setup(void) {
   OPTION_REG = 0x07;     // Configura Timer0 com prescaler 1:256
   INTCON = 0xE4;         // Habilita interrupção do Timer0 e interrupções globais
   TMR0 = 0x00;           // Inicializa o Timer0
   ANSEL = 0x00;          // Configura todos os pinos como digitais
   TRISC = 0x00;          // Configura todos os pinos da porta C como saída
   PORTC = 0x00;          // Inicializa todos os pinos da porta C em nível baixo
}

void main(void) {
   setup();
```

```
    // Loop principal
    while (1) {
        // Lógica do semáforo baseada no valor do contador
        switch (contador) {
            case 0:
                PORTC = 0b00000001;  // Verde ligado
                break;
            case 100:
                PORTC = 0b00000010;  // Amarelo ligado
                break;
            case 150:
                PORTC = 0b00000100;  // Vermelho ligado
                break;
            case 250:
                PORTC = 0b00000101;  // Vermelho e amarelo ligados
                break;
            default:
                break;
        }

        // Reinicia o contador ao atingir o limite
        if (contador >= 300) {
            contador = 0;
        }
    }
}

// Rotina de interrupção
void __interrupt() ISR() {
    if (INTCONbits.T0IF) {  // Verifica estouro do Timer0
        contador++;         // Incrementa o contador
        TMR0 = 0x00;        // Reinicia o Timer0
        INTCONbits.T0IF = 0; // Limpa a flag de interrupção do Timer0
    }
}
```

Explicação do Código

1. **Configuração do Timer0:**
    - O `OPTION_REG` ajusta o prescaler para reduzir a frequência do clock do temporizador, permitindo intervalos maiores entre os estouros.
    - O `TMR0` é inicializado em 0 para começar a contagem.
2. **Interrupções:**
    - O estouro do Timer0 dispara a interrupção, incrementando a variável `contador`.
    - A rotina de interrupção limpa a flag do Timer0 para permitir novos eventos.
3. **Lógica do Semáforo:**
    - A cada valor específico do contador, o estado do semáforo muda (LEDs diferentes são ligados).
    - O contador é reiniciado após atingir o limite.

Benefícios do Timer0 no Semáforo

- **Precisão:** Os estados são alternados em tempos exatos, sem atrasos cumulativos.
- **Eficiência:** O uso de interrupções garante que o programa principal não fique bloqueado.
- **Flexibilidade:** O prescaler permite ajustes finos no tempo entre as transições.

Esse exemplo mostra como o Timer0 e as interrupções tornam o controle de tempo mais eficaz, possibilitando a execução de tarefas paralelas em sistemas embarcados.

```
// Configuração para o PIC16F630

#include <xc.h>


// Definir frequência do oscilador (4MHz por exemplo)

#define _XTAL_FREQ 4000000
```

```
// Definição dos LEDs
#define LED1 RA0
#define LED2 RA1

// Variável volátil para armazenar o LED atual
volatile unsigned char currentLED = 0;

// Protótipos das funções
void __interrupt() isr(void);

void main() {
    // Configurar LEDs como saída
    TRISAbits.TRISA0 = 0; // LED1 (RA0 como saída)
    TRISAbits.TRISA1 = 0; // LED2 (RA1 como saída)

    // Configurar botão como entrada
    TRISAbits.TRISA2 = 1; // Botão em RA2 como entrada

    // Habilitar pull-up interno para o botão (se aplicável)
    OPTION_REGbits.nRAPU = 0; // Ativar pull-ups globais
    WPUA2 = 1; // Ativar pull-up em RA2

    // Configurar interrupções
```

```
INTCON = 0b10010000;  // Habilitar interrupção externa e global
OPTION_REGbits.INTEDG = 0; // Interrupção na borda de descida

// Inicializar LEDs
LED1 = 0;
LED2 = 0;
currentLED = 0; // Inicia com LED1

while (1) {
    // Loop principal

    // Efeito Fade In
    for (unsigned char i = 0; i < 255; i++) {
        if (currentLED == 0) {
            LED1 = 1;
            __delay_ms(10);
        } else {
            LED2 = 1;
            __delay_ms(10);
        }
    }

    // Efeito Fade Out
```

```
        for (unsigned char i = 255; i > 0; i--) {
            if (currentLED == 0) {
                LED1 = 0;
                __delay_ms(10);
            } else {
                LED2 = 0;
                __delay_ms(10);
            }
        }
    }
}

// Função de interrupção
void __interrupt() isr(void) {
    if (INTCONbits.INTF) { // Verifica se a interrupção externa foi disparada
        currentLED = !currentLED; // Alterna entre os LEDs
        INTCONbits.INTF = 0; // Limpa a flag de interrupção
    }
}
```

Projeto: LED com PWM

No contexto de sistemas embarcados, a manipulação da intensidade de LEDs é uma aplicação comum para testes ou feedback visual em dispositivos. O código apresentado faz uso de uma abordagem simples de **controle de intensidade de LED** em um microcontrolador PIC16F630, onde a intensidade do brilho do LED é ajustada ao variar o **duty cycle** de um ciclo de pulso. Embora não utilize temporizadores ou

interrupções, esse tipo de controle ainda possui seu valor, especialmente em sistemas simples que não requerem multitarefas complexas.

Funcionamento do Código

O código abaixo ajusta o brilho do LED conectado ao pino **RC0** do PIC16F630. Ele implementa um controle de intensidade de LED que varia de acordo com o **duty cycle**, ou ciclo de trabalho, em que o LED é aceso ou apagado.

1. Configuração Inicial

O código inicia com a configuração dos fusíveis do PIC16F630, como o oscilador interno (`FOSC = INTRCCLK`) e a desativação do **Watchdog Timer** (`WDTE = OFF`). O pino **RC0** é configurado como saída utilizando o comando `TRISCbits.TRISC0 = 0`, permitindo que o microcontrolador controle o LED através deste pino.

2. Controle de Intensidade do LED

A lógica principal é composta por dois loops aninhados que ajustam o brilho do LED:

- **Loop de Aumento de Intensidade:** O primeiro loop aumenta gradualmente o brilho do LED. A variável `duty` representa o tempo em que o LED fica aceso em cada ciclo. À medida que `duty` aumenta, o LED fica aceso por mais tempo durante o ciclo e apagado por menos tempo. Isso cria o efeito de aumento de intensidade.
- **Loop de Diminuição de Intensidade:** O segundo loop diminui a intensidade do LED de forma reversa. O valor de `duty` é decrementado, fazendo com que o LED fique aceso por menos tempo a cada ciclo, até que atinja o valor mínimo de `duty = 0`, apagando o LED completamente.

3. Atrasos e Controle de Tempo

Os **delays** são gerados pela função `__delay_us(10)`, que cria um intervalo de 10 microssegundos. Este atraso é usado para controlar o tempo em que o LED permanece aceso e apagado, ajustando o **duty cycle** para que o brilho do LED varie suavemente de 0 a 255.

A função de atraso, no entanto, tem algumas limitações, como o bloqueio da execução de outras tarefas enquanto o atraso está em andamento. Esse é um ponto onde o uso de temporizadores poderia trazer mais eficiência.

Vantagens e Limitações do Método Usado

Vantagens

- **Simplicidade:** O código é simples e fácil de entender, sendo uma boa introdução para quem está começando com o controle de LEDs ou programação de microcontroladores.
- **Controle Básico de Brilho:** Embora não seja a forma mais eficiente, o controle de brilho usando `__delay_us` é eficaz para sistemas simples onde a prioridade é apenas alternar entre os estados de brilho do LED.

Limitações

- **Uso de CPU:** Como o código depende de atrasos ativos (`__delay_us`), ele bloqueia a execução do microcontrolador, tornando-o incapaz de realizar outras tarefas enquanto o atraso está em andamento.
- **Falta de Precisão e Eficiência:** A variação do brilho é controlada por **loops de software**, o que pode ser impreciso e ineficiente. Isso ocorre porque o tempo de atraso pode ser afetado por outras operações internas do microcontrolador.

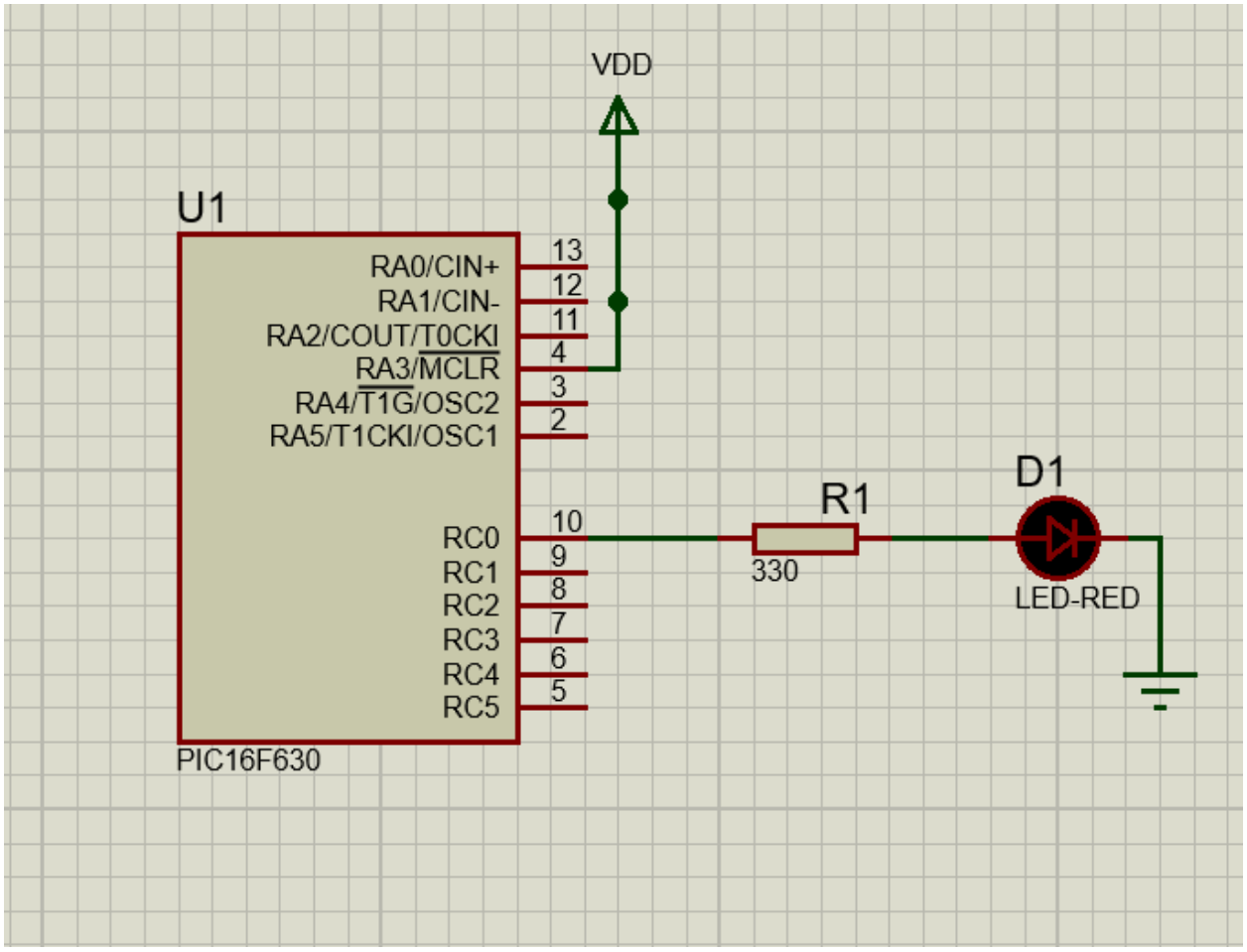
Melhoria: Uso de Temporizadores

Embora o método atual funcione bem para sistemas simples, ele pode ser otimizado. O uso de **temporizadores** e **PWM (modulação por largura de pulso)** poderia permitir que o controle de brilho fosse realizado de forma mais eficiente. Ao usar um temporizador ou PWM, o microcontrolador poderia manter o controle do LED sem a necessidade de bloquear sua execução com atrasos, permitindo que o sistema execute outras tarefas simultaneamente.

Em sistemas mais complexos, onde a simultaneidade e a precisão são cruciais, uma abordagem baseada em temporizadores e interrupções é mais adequada.

```
#include <xc.h>

// Configuração dos fusíveis (PIC16F630)
#pragma config FOSC = INTRCCLK  // Oscilador interno com clock I/O
#pragma config WDTE = OFF       // Watchdog Timer desabilitado
#pragma config PWRTE = ON       // Power-up Timer habilitado
#pragma config MCLRE = ON       // Pino MCLR habilitado
#pragma config BOREN = ON       // Brown-out Reset habilitado
#pragma config CP = OFF         // Proteção de código desabilitada
#pragma config CPD = OFF        // Proteção de dados desabilitada
```

```
#define _XTAL_FREQ 4000000     // Frequência do oscilador interno: 4 MHz

void main() {
    // Configurar RB0 como saída
    TRISCbits.TRISC0 = 0; // RC0 configurado como saída

    while (1) {
        // Aumentando a intensidade do LED
        for (unsigned char duty = 0; duty <= 255; duty++) {
            // Liga o LED por um tempo proporcional ao duty cycle
            PORTCbits.RC0 = 1; // Liga o LED
            for (unsigned char i = 0; i < duty; i++) {
                __delay_us(10); // Tempo ligado
            }

            // Desliga o LED pelo restante do ciclo
            PORTCbits.RC0 = 0; // Desliga o LED
            for (unsigned char i = duty; i < 255; i++) {
                __delay_us(10); // Tempo desligado
            }
        }

        // Diminuindo a intensidade do LED
        for (unsigned char duty = 255; duty > 0; duty--) {
            // Liga o LED por um tempo proporcional ao duty cycle
            PORTCbits.RC0 = 1; // Liga o LED
            for (unsigned char i = 0; i < duty; i++) {
                __delay_us(10); // Tempo ligado
            }

            // Desliga o LED pelo restante do ciclo
            PORTCbits.RC0 = 0; // Desliga o LED
            for (unsigned char i = duty; i < 255; i++) {
                __delay_us(10); // Tempo desligado
            }
        }
    }
}
```



Projeto: LED com Botões e interrupções

O código implementa um controle de LEDs usando uma interrupção externa para alternar entre dois LEDs (LED1 e LED2) no microcontrolador **PIC16F630**. O controle de LEDs também inclui efeitos de **fade in** e **fade out**, alternando a intensidade do LED de maneira gradual.

Configuração Inicial

1. **Configuração dos LEDs (RA0 e RA1):**
    - Os pinos **RA0** e **RA1** são configurados como saídas para controlar os LEDs `LED1` e `LED2` respectivamente.
2. **Configuração do Botão (RA2):**
    - O pino **RA2** é configurado como entrada, onde um botão será conectado. O botão acionará uma interrupção para alternar entre os LEDs.
    - A opção `OPTION_REGbits.nRAPU = 0` ativa os **pull-ups** internos no PIC16F630, e `WPUA2 = 1` ativa o pull-up para o pino **RA2**.

3. **Interrupções Externas:**
    - As interrupções externas são habilitadas configurando o registro `INTCON = 0b10010000`. Isso ativa as interrupções globais e externas.
    - A borda de ativação da interrupção é configurada para **borda de descida** (quando o botão for pressionado), através da configuração `OPTION_REGbits.INTEDG = 0`.

Efeito Fade In e Fade Out

O código implementa um **efeito de fade** (aumento e diminuição gradual de intensidade) nos LEDs. O loop principal alterna entre os dois LEDs, e o efeito é criado ajustando a intensidade do LED ligado, com um delay de 10 milissegundos.

- **Fade In:** O loop começa com o valor de `i` em 0 e vai até 255, acendendo o LED gradualmente. Para o LED atual (`LED1` ou `LED2`), ele é aceso (`LED1 = 1` ou `LED2 = 1`), e um atraso de 10ms é aplicado a cada iteração.
- **Fade Out:** O segundo loop diminui o valor de `i` de 255 até 0, apagando o LED ligado de forma gradual, com o mesmo atraso de 10ms.

Interrupção Externa (Alternância de LEDs)

A interrupção externa é configurada para ser acionada quando um botão é pressionado (transição de nível alto para nível baixo no pino RA2). Quando o botão é pressionado, a interrupção é disparada, e a função de interrupção alterna o LED atual (`currentLED`), alternando entre `LED1` e `LED2`.

- **Alternância de LEDs:** Quando a interrupção é gerada (pressionamento do botão), a variável `currentLED` é invertida (`currentLED = !currentLED`), trocando o LED atual. A flag de interrupção externa (`INTCONbits.INTF`) é limpa para permitir que novas interrupções sejam processadas.

Principais Funções

1. **`__interrupt()` (Função de Interrupção):**
    - Verifica se a interrupção externa foi disparada pela borda de descida no pino **RA2**.
    - Quando a interrupção ocorre, o código alterna entre os LEDs, alterando a variável `currentLED` e limpando a flag de interrupção para permitir futuras interrupções.
2. **Loop Principal:**

    - Realiza os efeitos de fade in e fade out para o LED ativo, que é determinado pela variável `currentLED`.
    - O loop nunca termina, mantendo o microcontrolador ativo e gerenciando o controle dos LEDs continuamente.

Considerações e Melhorias

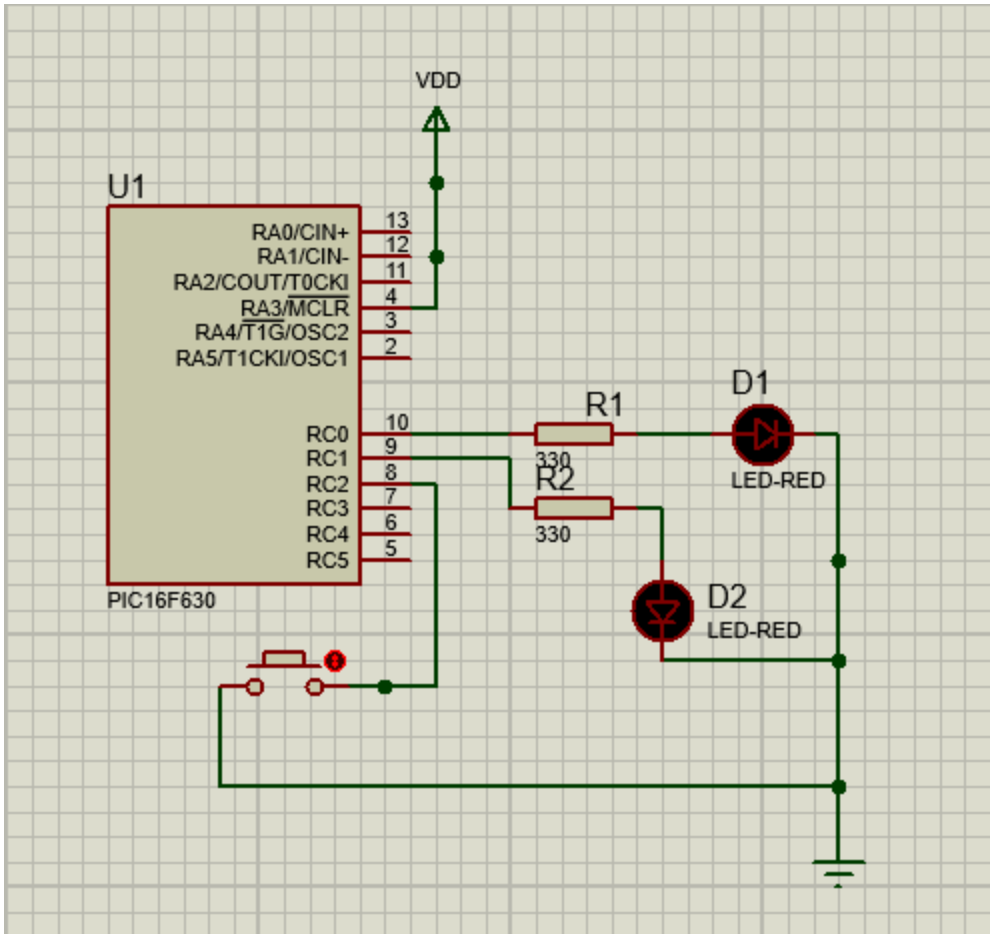
- **Efeito de Fade:** O efeito de fade é simulado através de delays simples (`__delay_ms(10)`), mas este método pode não ser o mais eficiente em sistemas mais complexos. Uma alternativa seria usar um **PWM (Pulse Width Modulation)** para controlar a intensidade do LED de forma mais precisa.
- **Interrupção para Alternar LEDs:** A utilização de interrupção para alternar entre LEDs ao pressionar o botão permite que o microcontrolador continue executando o efeito de fade nos LEDs sem a necessidade de checar constantemente o estado do botão, otimizando o processo.

```
// Configuração para o PIC16F630

#include <xc.h>


// Definir frequência do oscilador (4MHz por exemplo)

#define _XTAL_FREQ 4000000


// Definição dos LEDs

#define LED1 RA0

#define LED2 RA1


// Variável volátil para armazenar o LED atual

volatile unsigned char currentLED = 0;


// Protótipos das funções
```

```
void __interrupt() isr(void);

void main() {
    // Configurar LEDs como saída
    TRISAbits.TRISA0 = 0; // LED1 (RA0 como saída)
    TRISAbits.TRISA1 = 0; // LED2 (RA1 como saída)

    // Configurar botão como entrada
    TRISAbits.TRISA2 = 1; // Botão em RA2 como entrada

    // Habilitar pull-up interno para o botão (se aplicável)
    OPTION_REGbits.nRAPU = 0; // Ativar pull-ups globais
    WPUA2 = 1; // Ativar pull-up em RA2

    // Configurar interrupções
    INTCON = 0b10010000;  // Habilitar interrupção externa e global
    OPTION_REGbits.INTEDG = 0; // Interrupção na borda de descida

    // Inicializar LEDs
    LED1 = 0;
    LED2 = 0;
    currentLED = 0; // Inicia com LED1
```

```
while (1) {
    // Loop principal

    // Efeito Fade In
    for (unsigned char i = 0; i < 255; i++) {
        if (currentLED == 0) {
            LED1 = 1;
            __delay_ms(10);
        } else {
            LED2 = 1;
            __delay_ms(10);
        }
    }

    // Efeito Fade Out
    for (unsigned char i = 255; i > 0; i--) {
        if (currentLED == 0) {
            LED1 = 0;
            __delay_ms(10);
        } else {
            LED2 = 0;
            __delay_ms(10);
        }
```

```
        }
    }
}

// Função de interrupção
void __interrupt() isr(void) {
    if (INTCONbits.INTF) { // Verifica se a interrupção externa foi disparada
        currentLED = !currentLED; // Alterna entre os LEDs
        INTCONbits.INTF = 0; // Limpa a flag de interrupção
    }
}
```


VDD
U1
RA0/CIN+
RA1/CIN-
RA2/COUT/T0CKI
RA3/MCLR
RA4/T1G/OSC2
RA5/T1CKI/OSC1
RC0
RC1
RC2
RC3
RC4
RC5
13
12
11
4
3
2
10
9
8
7
6
5
PIC16F630
R1
330
R2
330
D1
LED-RED
D2
LED-RED