DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 8: Entradas e Saídas Digitais Paralelas
Características dos Sinais Digitais, Modos de Operação dos Pinos GPIO, Comunicação Paralela, Programação Paralela, Protocolo LPT, Interfaces Paralelas Personalizadas (*Display* de 7 Segmentos, LCD, Teclado Matricial)

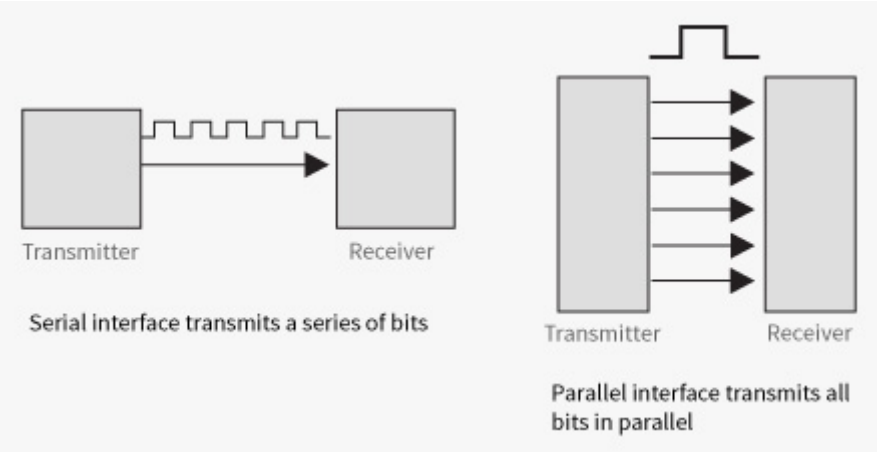
Profs. Antonio A. F. Quevedo e Wu Shin-Ting

FEEC / UNICAMP

Revisado e modificado em abril de 2025 por Ting com auxílio do Chatgpt
Revisado em setembro de 2024

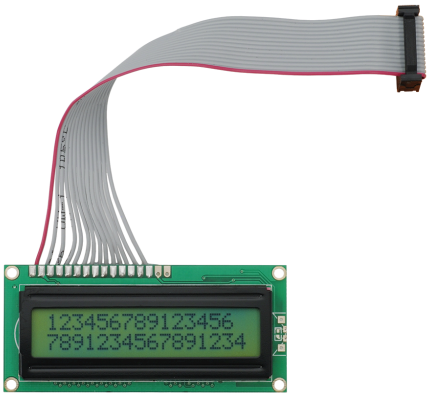
INTRODUÇÃO

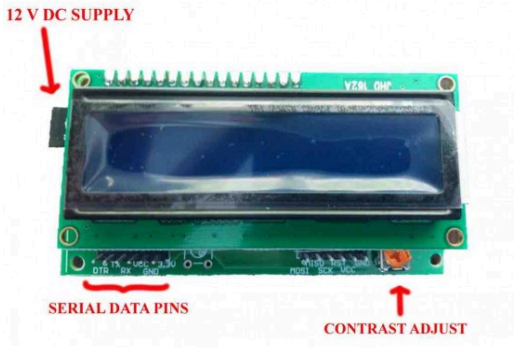
Vimos no Roteiro 7 que a entrada/saída paralela é um método de comunicação que permite a transmissão de múltiplos *bits* de dados simultaneamente, utilizando vários fios ou canais. Essa abordagem contrasta com a comunicação serial, onde os dados são enviados um *bit* por vez, sequencialmente, como ilustra a figura.



Embora a tecnologia serial tenha se tornado predominante em muitos dispositivos modernos, devido à sua eficiência em longas distâncias e menor complexidade de cabeamento, a entrada/saída paralela ainda desempenha um papel importante, especialmente em aplicações de curto alcance e em sistemas embarcados que exigem alta largura de banda. As interfaces paralelas apresentam características únicas que as diferenciam. Primeiramente, a **transmissão simultânea** permite o envio de múltiplos *bits* de dados ao mesmo tempo, resultando em taxas de transferência superiores em comparação à comunicação serial. Essa simultaneidade proporciona **baixa latência**, ideal para aplicações que demandam respostas rápidas. Além disso, a **simplicidade de conexão** é uma vantagem significativa, pois utiliza vários fios – geralmente um para cada *bit* de dados – e pinos para sinais de controle e relógio. Isso facilita a comunicação direta entre dispositivos e torna a implementação de **protocolos mais simples**, especialmente em sistemas embarcados. Por fim, a comunicação paralela é menos **susceptível a erros em distâncias curtas**, tornando-a uma escolha confiável em contextos específicos.

Não à toa, os barramentos internos de processadores, responsáveis por mover dados entre registradores, memória e periféricos, operam de forma paralela. Até mesmo a escrita em registradores de chave, como vimos no Roteiro 5, comuns em aplicações de segurança, costuma ser feita paralelamente, tanto por desempenho quanto por proteção contra estados intermediários indesejados. Além disso, tanto as interfaces seriais quanto as paralelas continuam, por exemplo, a ser comuns em *displays* como LCDs (do inglês, *Liquid Crystal Display*), TFTs (do inglês, *Thin-Film Transistor*) e OLEDs (do inglês, *Organic Light Emitting Diode*). Muitas placas de LCD são projetadas para operar em diferentes modos de comunicação, permitindo que os usuários escolham entre protocolos paralelos ou seriais, embora apenas um protocolo esteja ativo de cada vez. Isso demonstra a flexibilidade dos circuitos integrados controladores, que podem oferecer múltiplas opções de interface selecionáveis pelo usuário. Essa dualidade na comunicação destaca a importância das interfaces paralelas em aplicações onde a velocidade e a eficiência são necessárias, mesmo em um cenário dominado pela tecnologia serial.





Dando sequência ao nosso estudo, este roteiro detalhará como o módulo GPIO (do inglês *General Purpose Input/Output*) estabelece a comunicação entre microcontroladores e o mundo exterior, com ênfase no controle de sinais digitais de propósito geral (entradas e saídas paralelas). Anteriormente, investigamos o GPIO em dois contextos: no Roteiro 3, ao examinar o sistema de interrupção, onde sinais de entrada eram direcionados pelos módulos SYSCFG e EXTI ao controlador NVIC; e no Roteiro 5, ao demonstrar a capacidade de multiplexação dos pinos para diferentes funcionalidades através das alternativas de pinagem. Essa adaptabilidade permitiu a expansão do uso dos pinos, desde o controle fundamental de LEDs e um botão (Roteiros 1-4) até aplicações mais elaboradas nos roteiros subsequentes: geração de sinais digitais via clock de frequência específica (Roteiro 5), captura de eventos digitais, controle de saídas condicionais (Roteiro 6) e comunicação serial assíncrona (Roteiro 7).

Agora, neste roteiro, exploraremos nuances adicionais dos sinais digitais de propósito geral, incluindo previsibilidade, confiabilidade, latências, ruídos, sincronismo e a integridade física dos componentes envolvidos. Analisaremos as distinções entre a alteração simultânea (paralela) e a sequencial (serial) de *bits* em um registrador, além de discutir a implementação de um teclado matricial e os desafios de *bouncing* e *ghosting*. Por fim, apresentaremos informações sobre protocolos de comunicação paralela, cuja relevância persiste em cenários que exigem alta velocidade e interfaces simplificadas, mesmo com a menor prevalência em si.

PROJETOS-EXEMPLO

Nesta seção, apresentamos quatro projetos que exploram a interação entre sistemas digitais e dispositivos externos, como também acessos aos registradores dos microcontroladores, utilizando conceitos de interface paralela e sequencial. Esses projetos destacam aplicações práticas de acesso e controle, essenciais em sistemas embarcados e automação digital.

Programação paralela e sequencial de *bits*

Como vocês imaginam que os sinais gerados nos pinos se comportam ao atribuímos valores aos *bits* de forma sequencial e paralela? Na abordagem sequencial, cada *bit* é alterado individualmente, enquanto a abordagem paralela permite que múltiplos *bits* sejam modificados ao mesmo tempo. Como cada método influencia a relação e a sincronização entre os sinais? Qual abordagem é mais adequada quando o *hardware* requer que todos os *bits* de um registrador sejam alterados simultaneamente, como fazer um acesso de escrita no registrador de chave? Neste projeto, iremos investigar essas questões de maneira prática. Vocês terão a oportunidade de experimentar e compreender as sutilezas entre essas duas estratégias de interface, aprofundando-se em como elas afetam o funcionamento dos sistemas digitais.

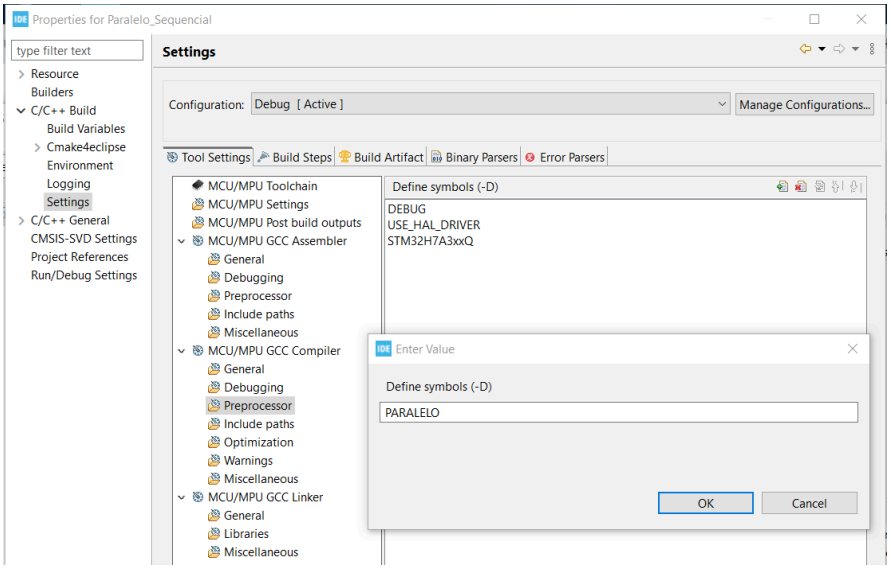
1. Crie um novo projeto, chamado "Paralelo_Sequencial". Ative o *Debug* e mude o *clock* geral para 32MHz

2. Utilizaremos um recurso do C conhecido como **compilação condicional**. Essa técnica permite incluir ou excluir partes do código-fonte durante a fase de compilação, com base em condições específicas. Isso é realizado por meio de diretivas de pré-processador, como `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif` e `#endif`, que foram introduzidas no Roteiro 2. Essas diretivas possibilitam habilitar ou desabilitar trechos de código com base em definições de macros ou constantes. Essa abordagem é especialmente útil para compilar um mesmo programa em diferentes sistemas operacionais ou arquiteturas, incluir ou excluir código para testes, como *logs* de *debug*, sem modificar a lógica principal, e ativar ou desativar funcionalidades conforme as necessidades do projeto.

Neste projeto, vamos usar a compilação condicional para determinar se os níveis lógicos em vários pinos de um mesmo *port* serão alterados de forma paralela ou sequencial. Inicialmente, vamos definir a macro que determina a condição de compilação. No escopo de `/* USER CODE BEGIN PD */`, defina a macro:

```
#define PARALELO
```

Alternativamente, podemos adicionar este símbolo à lista de símbolos pré-definidos para o pré-processador do compilador GCC nas "Properties" do Projeto (C/C++ Build > Settings > GCC Compiler + Preprocessor)..



Se a clareza e a visibilidade imediata são prioritárias, a primeira opção (definição no código) pode ser melhor. Se a organização do projeto e a separação entre configuração e implementação são mais importantes, a segunda opção (uso das propriedades do projeto) pode ser a melhor escolha.

3. Vamos criar uma função para realizar a configuração da GPIO. No escopo de `/* USER CODE BEGIN PFP */`, defina o protótipo:

```
void Config_GPIO(void);
```

e no escopo de `/* USER CODE BEGIN 4 */`, implemente a função:

```
void Config_GPIO(void) {
	// PG9 a PG13 outputs alternando 0s e 1s na ordem
9-10-11-12-13
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOGEN;
	GPIOG->MODER &= ~(GPIO_MODER_MODE9_Msk |
GPIO_MODER_MODE10_Msk |
		GPIO_MODER_MODE11_Msk | GPIO_MODER_MODE12_Msk
		| GPIO_MODER_MODE13_Msk);
	GPIOG->MODER |= GPIO_MODER_MODE9_0 | GPIO_MODER_MODE10_0 |
		GPIO_MODER_MODE11_0 | GPIO_MODER_MODE12_0
		| GPIO_MODER_MODE13_0;
	GPIOG->BSRR = GPIO_BSRR_BR9 | GPIO_BSRR_BS10 |
		GPIO_BSRR_BR11 | GPIO_BSRR_BS12 | GPIO_BSRR_BR13;
}
```

4. Agora, dentro da função ”main” configure o GPIO. No escopo de `/* USER CODE BEGIN 2 */`, chame a função:

```
Config_GPIO();
```

5. Por fim, implementamos a alternância de níveis lógicos dentro do *loop*. Para isso, utilizaremos a compilação condicional com as diretivas `#if`, `#elif`, entre outras. Essas diretivas nos permitirão escolher se queremos compilar o código que realiza a alternância em paralelo ou o que a executa de forma sequencial. Abaixo da linha `/* USER CODE BEGIN 3 */`, escreva o código:

```
#ifdef PARALELO
        // mudanca em paralelo
        GPIOG->BSRR = GPIO_BSRR_BS9 | GPIO_BSRR_BR10 |
                 GPIO_BSRR_BS11 | GPIO_BSRR_BR12 | GPIO_BSRR_BS13;
        HAL_Delay(1);
        GPIOG->BSRR = GPIO_BSRR_BR9 | GPIO_BSRR_BS10 |
                 GPIO_BSRR_BR11 | GPIO_BSRR_BS12 | GPIO_BSRR_BR13;
        HAL_Delay(1);
#else
        // Mudanca sequencial
        GPIOG->BSRR = GPIO_BSRR_BS9;
        GPIOG->BSRR = GPIO_BSRR_BR10;
        GPIOG->BSRR = GPIO_BSRR_BS11;
        GPIOG->BSRR = GPIO_BSRR_BR12;
        GPIOG->BSRR = GPIO_BSRR_BS13;
        HAL_Delay(1);
        GPIOG->BSRR = GPIO_BSRR_BR9;
        GPIOG->BSRR = GPIO_BSRR_BS10;
        GPIOG->BSRR = GPIO_BSRR_BR11;
        GPIOG->BSRR = GPIO_BSRR_BS12;
        GPIOG->BSRR = GPIO_BSRR_BR13;
        HAL_Delay(1);
#endif
```

Note que a parte do código que não atende à condição (após o #else) apresenta um fundo cinza-claro, sinalizando que esta parte não será compilada. Faça o *Build*, compilando a versão que realiza a mudança dos níveis lógicos em paralelo.

6. Conecte o analisador lógico aos pinos 4 a 8 do conector “TECLADO”. O pino 1 está mais à esquerda. Ligue os canais 0 a 4 nos pinos na seguinte ordem: 5, 6, 4, 7, 8 (PG9 a PG13) e o GND no pino mais à direita do conector H11 (usado no roteiro de *timers*). Ajuste a aquisição para a maior velocidade possível (24MS/s), por um tempo de 10ms, e com *trigger* de borda de subida no canal 0.

7. Faça o *Debug* e execute o programa. Realize a coleta dos dados e aplique o máximo de zoom possível em torno de uma transição de níveis lógicos. Observe atentamente a relação temporal entre os cinco sinais. Neste caso, as alternâncias dos *bits* PG9 a PG13 foram programadas simultaneamente ou sequncialmente? É visível a presença de *skew* de dados? É esperado este resultado na comunicação entre o núcleo do processador e o GPIOG? Caso não tenha uma resposta ainda, ela virá mais adiante.

8. Vamos analisar comparativamente os sinais observados com os sinais gerados através de uma abordagem serial. Pare a execução do programa, e comente a linha “#define PARALELO”, para que a compilação condicional use o código que muda o estado de um pino de cada vez. Faça o *Build* e o *Debug* novamente e execute o programa. Execute uma nova captura no analisador lógico e aplique o máximo zoom na região próxima ao trigger. Compare os deslocamentos temporais relativos entre as bordas de subida (dissonância temporal) dos sinais capturados no modo "PARALELO" com aqueles observados neste modo alternativo. Nessa programação, qual abordagem de programação dos *bits* é aplicada? Você consegue explicar as diferenças entre duas abordagens de programação? Reexamine o código e verifique se a causa pode ser identificada ao comparar os instantes em que os *bits* são alterados. Podemos chamar os deslocamentos temporais neste modo alternativo de *skew*? Caso a compreensão ainda não tenha ocorrido, segue adiante e veremos a resposta posteriormente.

Relógio digital: programação paralela de *bits*

Você sabia que existem circuitos que exigem que os *bits* de uma instrução sejam escritos simultaneamente para serem considerados válidos? No Roteiro 5 apresentamos o projeto de um relógio digital usando o módulo RTC. Este módulo utiliza um mecanismo de proteção de escrita nos registradores do RTC para garantir a integridade dos dados. Apenas se pode modificar o conteúdo dos registradores do módulo depois de desbloquear esta proteção com duas chaves. Para habilitar a reconfiguração do RTC, todos os bits das duas chaves devem ser escritos simultaneamente no registrador de chave RTC_WPR.

1.Crie um projeto novo usando o *Cube*, com o nome “Paralelo_RTC” e **desative** a opção **“Initialize all peripherals with their default!”**. Ative o módulo *Debug* como “Serial Wire”.

2. Verifique na seção “Pinout & Configuration” que o módulo “RTC” da categoria "Timers", está desativado. Em vez de ativá-lo e configurá-lo através do editor gráfico, optamos por fazê-lo diretamente através das instruções que escreveremos, baseadas na interface CMSIS, como no Roteiro 5. Se expandir o módulo “GPIO” da categoria “System Core”, verá que os pinos PB0, PE1 e PB14, onde os LEDs LD1, LD2 e LD3 do NUCLEO estão, respectivamente, conectados, já estão habilitados. E se expandir o módulo “NVIC” da mesma categoria, verá que a interrupção está ativada e 4 *bits* são atribuídos para configurar o grupo de prioridade.

3. Gere o código de inicialização pelo *Cube* e abra o arquivo “main.c”.

4. Vamos implementar a nossa função de inicialização do módulo RTC, `RTC_PInit` cuja declaração deve ser incluída no escopo /* USER CODE BEGIN PFP */

```
void RTC_PInit(void);
```

O módulo RTC é inicializado com uma data, um horário e um alarme que dispara eventos periódicos em cada segundo para atualização automática das variáveis de horário que organizamos em duas `struct` declaradas no escopo /* USER CODE BEGIN 1 */

```
struct {
        uint8_t HH;
        uint8_t MM;
        uint8_t SS;
} horario;
struct {
        uint8_t ano;
        uint8_t mes;
        uint8_t dia;
} dia;
```

Incluímos em `RTC_PInit` a inicialização do RTC recomendada pelo fabricante. Para esclarecer as funções das instruções na configuração e inicialização do módulo RTC, organizamos as instruções em grupos, utilizando pares de chaves {} para separar cada função.

```
void RTC_PInit (void) {
  {
   // Habilitar o sinal de ativacao de RTC (eh ativado no reset - default)
   RCC->APB4ENR |= RCC_APB4ENR_RTCAPBEN_Msk;
  }
  {
   // Habilitar acesso de escrita a BDCR e configurar a fonte do RTC
   PWR->CR1 |= PWR_CR1_DBP;
   while (!(PWR->CR1 & PWR_CR1_DBP)); // Aguarda a habilitação
   // Resetar o domínio de Switching de Tensão (VSW) - consultar o manual
   RCC->BDCR |= RCC_BDCR_VSWRST;
   // Restaurar o conteúdo anterior com habilitação de RTC
   RCC->BDCR &= ~(RCC_BDCR_RTCSEL | RCC_BDCR_VSWRST);
   RCC->BDCR |= (RCC_BDCR_RTCEN_Msk |
   RCC_BDCR_RTCSEL_1);
   // Habilita LSI
   RCC->CSR |= RCC_CSR_LSION;
   while (!(RCC->CSR & RCC_CSR_LSIRDY)); // Aguarda estabilização do LSI
  }
  {
    // Desbloquear a proteção de escrita
   RTC->WPR = 0xCAU;  // Desbloquear a proteção de escrita
   RTC->WPR = 0x53U;
  }
  {
    // Modo de Inicialização do RTC
    RTC->ICSR |= RTC_ICSR_INIT; // Entra no modo de inicialização
    while (!(RTC->ICSR & RTC_ICSR_INITF)); // Aguarda a inicialização do
    RTC
    // Configura o formato de hora para 24 horas
```

```
    RTC->CR &= ~RTC_CR_FMT;
    // Configura o pre-scaler do RTC
    RTC->PRER = (127 << RTC_PRER_PREDIV_A_Pos) | (255 <<
    RTC_PRER_PREDIV_S_Pos);
    // Configurar a hora
    RTC->TR = ((2 << RTC_TR_HT_Pos) | (3 << RTC_TR_HU_Pos) |  // Horas
    (23)
     (4 << RTC_TR_MNT_Pos) | (5 << RTC_TR_MNU_Pos) |  // Minutos (45)
     (1 << RTC_TR_ST_Pos) | (2 << RTC_TR_SU_Pos));  // Segundos (12)
    // Configurar a data
    RTC->DR = ((2 << RTC_DR_YT_Pos) | (4 << RTC_DR_YU_Pos) | // Ano (2024
    - 2000 = 24)
    (0 << RTC_DR_MT_Pos) | (8 << RTC_DR_MU_Pos) |  // Mês (Agosto = 8)
     (3 << RTC_DR_DT_Pos) | (1 << RTC_DR_DU_Pos));  // Dia (31)
     RTC->ICSR &= ~RTC_ICSR_INIT; // Sair do modo de inicialização
   }
}
```

Em seguida, implementamos o procedimento de configuração do alarme também recomendado pelo fabricante. Embora essa configuração possa ser realizada no modo de inicialização, optamos por executá-la fora desse modo para demonstrar que é possível alterar as configurações do alarme mesmo sem estar no modo de inicialização, precisando apenas ter a proteção desbloqueada. Insira o seguinte bloco de código no final da versão anterior da função `RTC_PInit`.

```
{
  // Configura o alarme
  // Desabilitar o alarme
  RTC->CR &= ~RTC_CR_ALRAE; // Desabilita o Alarme A
  // Desativar a interrupcao do alarme
  RTC->CR &= ~RTC_CR_ALRAIE; // Desabilitar a interrup do alarme A
  // Limpar a flag de interrupção
  RTC->SCR |= RTC_SCR_CALRAF_Msk;
  while (!(RTC->ICSR & RTC_ICSR_ALRAWF)); // Aguarda config
  RTC->ALRMAR &= ~(RTC_ALRMAR_PM);  //fomato 24 horas
  // Programa o evento alarme para cada segundo
  RTC->ALRMAR |= (RTC_ALRMAR_MSK4 | RTC_ALRMAR_MSK3 |
  RTC_ALRMAR_MSK2 | RTC_ALRMAR_MSK1);
  // Habilita o Alarme A
  RTC->CR |= RTC_CR_ALRAE;
  // Habilitar a interrupcao do alarme
  RTC->CR |= RTC_CR_ALRAIE;
}
```

STM32H7A3 permite configurar a periodicidade de um alarme através das quatro máscaras `Mskx` do registrador RTC_ALRMAR, conforme detalha a tabela 5 da Nota Técnica AN 3371. Para geração de eventos de alarme por segundo, os *bits* das quatro máscaras são setado em "1" pela instrução que deve ser adicionada no final da versão anterior de `RTC_PInit`.

```
{
```

```
  // Eventos de alarme por segundo
  RTC->ALRMAR     |=     (RTC_ALRMAR_MSK4     |     RTC_ALRMAR_MSK3     |
  RTC_ALRMAR_MSK2 | RTC_ALRMAR_MSK1);
}
```

Para garantir a integridade e a consistência dos dados, é necessário bloquear acessos de escrita do RTC após as configurações nos registradores com a seguinte instrução, que deve ser inserida no final da versão anterior de `RTC_PInit`.

```
{
  RTC->WPR = 0xFFU;    //Bloquear acesso de escrita
}
```

Para finalizar, precisamos habilitar a linha IRQ41 do NVIC na qual os eventos de interrupção do Alarme são mapeados.

exti_rtc_al	48	41	RTC_ALARM	RTC alarms (A and B) through EXTI Line interrupts	0x0000 00E4

Vimos no Roteiro 3 que o STM32H7A3 diferencia o tratamento de eventos internos e externos. Embora o RTC seja um componente integrado, ele funciona de forma independente do núcleo do processador, permitindo que continue contando e monitorando eventos mesmo quando o processador está inativo. Como resultado, algumas interrupções do RTC são tratadas de maneira semelhante às interrupções externas, pois têm a capacidade de "despertar" o sistema a partir de modos de baixo consumo, como *Sleep* ou *Stop*. Essas interrupções, assim como aquelas provenientes de módulos GPIO, são enviadas para o NVIC pelo módulo EXTI. Os eventos de entrada do EXTI em que os eventos de interrupção do RTC são mapeados podem ser consultados no Manual de Referência. Os eventos de alarme são associados aos eventos de entrada 17 do módulo EXTI.

Table 126. EXTI Event input mapping

Event input	Source	Event input type	Wakeup target(s)	Connection to NVIC
0 - 15	EXTI[15:0]	Configurable	Any	Yes
16	PVD and AVD(1)	Configurable	CPU only	Yes
17	RTC alarms	Configurable	CPU only	Yes
18	RTC tamper, RTC timestamp, RCC LSECSS(2)	Configurable	CPU only	Yes
19	RTC wakeup timer	Configurable	Any	Yes

Em termos práticos, isso significa que, além de habilitarmos a IRQ 41 correspondente à fonte de requisição ALARME no NVIC, precisamos habilitar o evento de entrada 17 do módulo EXTI para que os sinais do ALARME sejam amostrados. Portanto, vamos agregar o seguinte bloco de instruções no final da versão anterior de `RTC_PInit`.

```
{
  //NVIC: Configuracao de Atendimento de Interrupcoes
  // Configurar NVIC para habilitar a interrupção ALARM do RTC
  NVIC_SetPriority(RTC_Alarm_IRQn, 1); // Configura a prioridade da
  interrupção
  NVIC_EnableIRQ(RTC_Alarm_IRQn); // Habilita a interrupção no NVIC
  //Configuracao no EXTI
```

```
  EXTI->IMR1 |= EXTI_IMR1_IM17_Msk;     // habilita a interrupcao do
  evento de entrada 17
  EXTI->RTSR1 |= EXTI_RTSR1_TR17_Msk; // captura na borda de subida
}
```

5. Para finalizar, inserimos a chamada da função de inicialização no escopo /* USER CODE BEGIN 2 */

```
  // Configuração do RTC
  RTC_PInit();
```

6. Implementamos agora a ISR, `RTC_Alarm_IRQHandler`, no arquivo "stm32h7xx_it.c" que seta a variável `estado` em 1 em cada interrupção de 1s. Insira no escopo /* USER CODE BEGIN 1 */

```
static uint8_t estado=0;

/**
* @brief ISR para tratar o evento Alarme A e B
*/
void RTC_Alarm_IRQHandler (void) {
  //if (RTC->SR & RTC_SR_ALRAF) {
  if (RTC->SR & RTC_MISR_ALRAMF) {
     RTC->SCR |= RTC_SCR_CALRAF; // Limpar a flag
     EXTI->PR1 = EXTI_PR1_PR17; // Limpar a flag de pend. no EXTI
     // Seta estado em 1
     estado = 1;
  }
}
```

7. A variável `estado` é usada no fluxo de controle principal da função `main` para atualização dos membros das duas `structs` declaradas anteriormente no arquivo "main.c". Por **modularidade**, encapsulamos acessos a esta varivel por duas funções que são definidas antes da rotina de serviço e após a declaração de

```
static uint8_t estado=0;
/**
* @brief Le o estado de um novo segundo
*/
uint8_t RTC_IT_segundo () {
     return estado;
}
/**
* @brief Reseta o estado do segundo
*/
void RTC_reseta_IT_segundo () {
     estado = 0;
```

```
}
```

8. Voltamos agora para o arquivo “main.c” para completarmos a lista dos protótipos

```
/* USER CODE BEGIN PFP */
void RTC_PInit(void);
uint8_t RTC_IT_segundo ();
void RTC_reseta_IT_segundo ();
/* USER CODE END PFP */
```

e adicionarmos as duas variáveis locais no mesmo escopo das structs, `/* USER CODE BEGIN 1 */`

```
uint32_t tmpreg;
uint8_t ledon=0;
```

9. Finalmente, concluimos a nossa implementação com a inserção dos seguintes códigos no escopo `/* USER CODE BEGIN 3 */`

```
if (RTC_IT_segundo()) {
  RTC_reseta_IT_segundo();
  // Ler a hora
  tmpreg = RTC->TR;
  // Extrair componentes de hora
  horario.HH = ((tmpreg & RTC_TR_HT) >> RTC_TR_HT_Pos) * 10
  + ((tmpreg & RTC_TR_HU) >> RTC_TR_HU_Pos);
  horario.MM = ((tmpreg & RTC_TR_MNT) >> RTC_TR_MNT_Pos) * 10
      + ((tmpreg & RTC_TR_MNU) >> RTC_TR_MNU_Pos);
  horario.SS = ((tmpreg & RTC_TR_ST) >> RTC_TR_ST_Pos) * 10
      + ((tmpreg & RTC_TR_SU) >> RTC_TR_SU_Pos);
  // Ler a data
  tmpreg = RTC->DR;
  // Extrair componentes de data
  dia.ano = ((tmpreg & RTC_DR_YT) >> RTC_DR_YT_Pos) * 10
      + ((tmpreg & RTC_DR_YU) >> RTC_DR_YU_Pos) ;   // Ajuste para
  anos a partir de 2000
  dia.mes = ((tmpreg & RTC_DR_MT) >> RTC_DR_MT_Pos) * 10
      + ((tmpreg & RTC_DR_MU) >> RTC_DR_MU_Pos);
  dia.dia = ((tmpreg & RTC_DR_DT) >> RTC_DR_DT_Pos) * 10
      + ((tmpreg & RTC_DR_DU) >> RTC_DR_DU_Pos);
  if(ledon) {
  GPIOE->BSRR = GPIO_BSRR_BR1;
      ledon = 0;
  } else {
      GPIOE->BSRR = GPIO_BSRR_BS1;
      ledon = 1;
  }
}
```

Este trecho de código atualiza os membros das duas `structs` e controla a alternância do estado do LED LD2 do NUCLEO.

10. Realize o *Build* e transfira o código executável para o microcontrolador no modo *Debug*.

11. Continue (“Resume”) a execução do programa e observe o comportamento do sistema. Qual a frequência das alternâncias do estado do LED LD2? Se não souber, analise a configuração do registrador `RTC->ALRMAR` e consulte a Tabela 5 do *datasheet* do periférico RTC para responder.

12. Pause o programa e adicione um *breakpoint* na linha “`if(ledon) {`”. Em seguida, abra a aba “Variables” e expanda as duas `structs`. Continue a execução do programa (“Resume”) e observe os valores dos membros de `horario` a cada pausa. Os valores de hora e data correspondem à configuração inicial? Qual periférico do microcontrolador STM32H7A3 atualiza os registradores `RTC->TR` e `RTC->DR`, de onde você extrai os dados de horário e data?

13. Vamos analisar agora a importância de programação paralela dos *bits* do registrador de proteção de escrita `RTC_WPR`. Substitua as duas linhas de instruções para desbloquear a proteção de escrita por quatro linhas equivalentes em termos de resultados numéricos. Construa (“Build”) e transfira o código executável para o microcontrolador no modo *Debug*. Em seguida, continue (“Resume”) a execução. Quais foram suas observações? ? Qual fator você considera ter causado a alteração no comportamento do sistema? Uma dica é

```
//    RTC->WPR = 0xCAU;  // Desbloquear a proteção de escrita
//    RTC->WPR = 0x53U;
      RTC->WPR &= ~0xFF;  // Desbloquear a proteção de escrita
      RTC->WPR |= 0xCA;
      RTC->WPR &= ~0xFF;
      RTC->WPR |= 0x53;
```

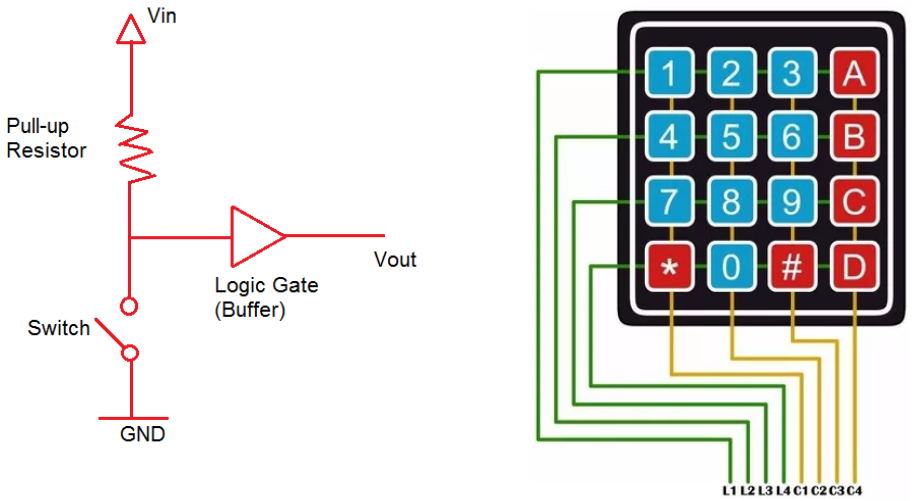
14. Restaure o estado original do programa. Reconstrua o código executável e execute o código para verificar se ele recuperou o seu comportamento anterior.

Projeto de um testador visual de *bouncing*

Como você imagina que os LEDs se comportarão ao programar um microcontrolador para que, a cada pressionamento de uma tecla, o LED aceso se desloque ciclicamente para o próximo na sequência LD1-LD2-LD3 no NUCLEO? É possível que os pressionamentos não sejam detectáveis por *software*? Neste projeto, implementaremos um testador visual de *bouncing* que inclui o controle de ativação do elemento de teste (tecla/botão/chave), permitindo verificar se o deslocamento sequencial e cíclico realmente ocorre ao pressionar o

elemento de teste. Para a prova de conceito, usaremos a tecla “0” de um *keypad* de membrana conectado no *header* H10 da placa de expansão alinhando L1 com o pino “1” de H10. Observe que o *keypad* disponível no almoxarifado não tem a coluna C4.

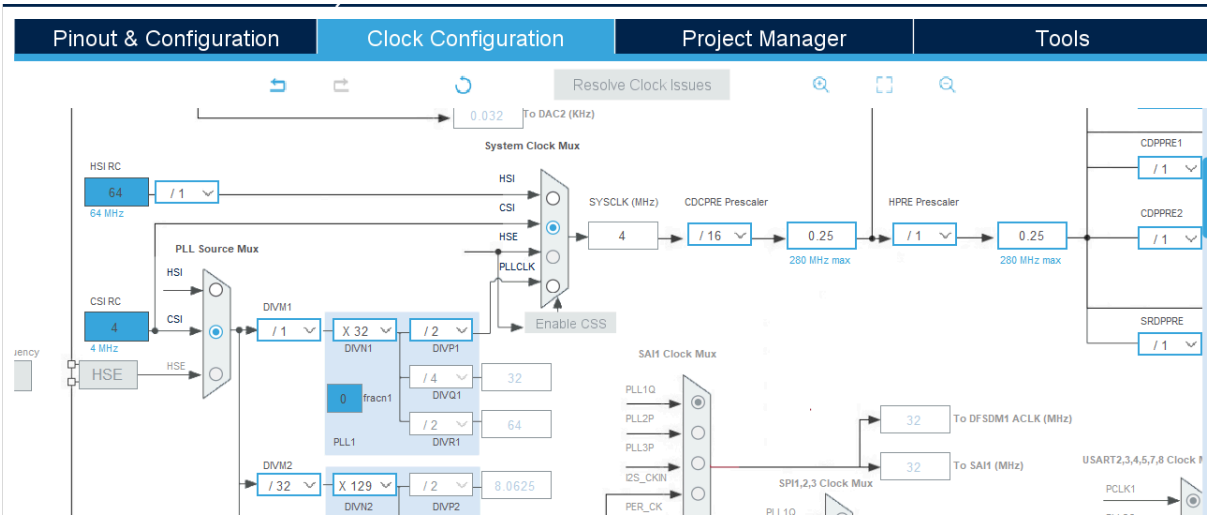
Consideramos que uma tecla, chave ou botão passa no teste de "sem *bouncing*" se o deslocamento do estado aceso do LED ocorre de forma sequencial. Caso contrário, afirmamos que o dispositivo apresenta *bouncing*.



O controle adicional de ativação do elemento de teste consiste em substituir o GND no esquemático à esquerda pela saída de um pino digital. Assim, quando essa saída estiver em nível lógico “0”, o comportamento do circuito será equivalente ao aterramento do ponto. Por outro lado, quando a saída estiver em nível lógico “1”, o nível em Vout será sempre “1”.

1. Crie um projeto novo usando o *Cube*, com o nome “Paralelo_Bouncing”, com a opção **“Initialize all peripherals with their default!” desabilitada**. Ative o módulo *Debug* como “Serial Wire”.

2. Entre na aba “Clock Configuration” e modifique a frequência do *clock* do sistema para a fonte CSI na frequência **250kHz**.



3. Verifique na seção “Pinout & Configuration” que os pinos PB0, PE1 e PB14, onde os LEDs LD1, LD2 e LD3 do NUCLEO estão, respectivamente, conectados, já estão habilitados. E se expandir o módulo “NVIC” da mesma categoria “System Core”, verá que a interrupção está ativada e 4 *bits* são atribuídos para configurar o grupo de prioridade.

4. Gere o código de inicialização pelo *Cube* e abra o arquivo “main.c”.

5. Vamos implementar a função `GPIO_KEY_PInit` para inicializar os pinos PG11, correspondente à linha L4, e PG10, correspondente à coluna C2, garantindo que a configuração seja compatível com o esquemático: PG11 deve corresponder ao Terra, PG10 a Vout, e um resistor *pull-up* deve ser aplicado no pino PG10. Além disso, habilitaremos a interrupção da PG10 (entrada) para detectar a borda de descida. Não se esqueça de adicionar o protótipo no escopo `/* USER CODE BEGIN PFP */`.

```
void GPIO_KEY_PInit(void);
```

antes de implementar a sua definição no escopo /* USER CODE BEGIN 4 */

```
/**
* @brief Private initialization of the membrane keypad
*/
void GPIO_KEY_PInit(void) {
  {
    // Pinos usados: PG11 (L4) e PG10 (C2)
    // Habilita sinais de relogio
    RCC->AHB4ENR |= RCC_AHB4ENR_GPIOGEN;
    // Port G: PG11; GPIO output OPEN DRAIN
    GPIOG->MODER &= ~(GPIO_MODER_MODE11_Msk);
    GPIOG->MODER |= (GPIO_MODER_MODE11_0); // MODER11 = [01]
    GPIOG->OTYPER |= (GPIO_OTYPER_OT11); // Open drain
    // Port G:   PG10; GPIO Input
    GPIOG->MODER &= ~GPIO_MODER_MODE10_Msk;// MODER = [00]
    GPIOG->PUPDR &= ~GPIO_PUPDR_PUPD10_Msk; // PUPDRx = [01] pull-up
    GPIOG->PUPDR |= GPIO_PUPDR_PUPD10_0;
    // Coloca a saida L1 em nivel baixo
    GPIOG->BSRR = GPIO_BSRR_BR11; // PG11 RESET
  }
  {
    //Habilita interrup de PG10: bits [11:8] de SYSCFG_EXTICR3
    SYSCFG->EXTICR[2] &= ~SYSCFG_EXTICR3_EXTI10_Msk;
    SYSCFG->EXTICR[2] |= SYSCFG_EXTICR3_EXTI10_PG;
    // Ativar a borda de descida e desativar borda de subida
    EXTI->RTSR1 &= ~EXTI_RTSR1_TR10_Msk;
    EXTI->FTSR1 |= EXTI_FTSR1_TR10_Msk;
    // Mascaras de interrupcao
    EXTI->IMR1 |= EXTI_IMR1_IM10;
  }
  {
    //Habilita IRQ9 no NVIC:  EXTI15_10 (vetor 40), prioridade 1
     NVIC_SetPriority(EXTI15_10_IRQn, 1); // Configura a prioridade
  da interrupção
```

```
    NVIC_EnableIRQ(EXTI15_10_IRQn);
  }
```

6. Insira a chamada da função no escopo /* USER CODE BEGIN 2 */

```
GPIO_KEY_PInit();
```

7. Abra agora o arquivo "stm32h7xx_it.c" para implementarmos a ISR `EXTI9_5_IRQHandler` que processará os eventos de interrupção do pino PG10 no escopo /* USER CODE BEGIN 1 */

```
void EXTI15_10_IRQHandler(void) {
      static uint8_t ledon;
      if(EXTI->PR1 & EXTI_PR1_PR10_Msk) {
            EXTI->PR1 |= EXTI_PR1_PR10_Msk; // Limpa o flag
            switch (ledon) {
            case 0:
                  GPIOB->BSRR = GPIO_BSRR_BR14;
                  GPIOB->BSRR = GPIO_BSRR_BS0;
                  ledon = 1;
                  break;
            case 1:
                  GPIOB->BSRR = GPIO_BSRR_BR0;
                  GPIOE->BSRR = GPIO_BSRR_BS1;
                  ledon = 2;
                  break;
            case 2:
                  GPIOE->BSRR = GPIO_BSRR_BR1;
                  GPIOB->BSRR = GPIO_BSRR_BS14;
                  ledon = 0;
                  break;
            default:
            }
      }
}
```

8. Construa ("Build") o código executável e transfira o código para o microcontrolador no modo *Debug*. Em seguida, continue ("Resume") a execução e pressione a tecla "0" do *keypad* várias vezes. Observe como a cor do LED aceso muda, o que você pode concluir sobre o *bouncing* da tecla "0"?

9. Agora, vamos analisar o impacto da substituição da conexão ao terra por uma saída em "dreno aberto" no comportamento do sistema. Interrompa a execução do programa e altere o nível lógico de PG11 de "0" para '1'. Recompile ("`Terminate and Relaunch`") o projeto. Em seguida, pressione a tecla '0' repetidamente. Qual a alteração observada na cor do LED?

```
      //GPIOG->BSRR = GPIO_BSRR_BR11; // PG11 RESET
```

```
GPIOG->BSRR = GPIO_BSRR_BS11; // PG11 SET
```

10. Pause a execução e coloque um *breakpoint* na ISR `EXTI15_10_IRQHandler`. Observe se o fluxo de controle é interrompido dentro da rotina. Qual foi a sua constatação? Com a nova configuração, a detecção de variações nos estados da tecla através de eventos de interrupção ainda é possível? Você saberia descrever o efeito da pequena alteração que realizamos, a ponto de cessar a geração de eventos de interrupção e a detecção dos pressionamentos da tecla? Caso a resposta ainda não esteja clara, não se preocupe, pois a explicação será apresentada posteriormente.

11. Restaure o seu código original. Para realizar um teste visual de *bouncing* em qualquer botão ou tecla de dois estados, conecte os pinos do componente aos pinos 4 e 6 de H10. Em seguida, pressione o botão várias vezes e observe a sequência em que os LEDs acendem e apagam. Podemos considerar que o elemento de teste **pode estar** livre de *bouncing* para a frequência de operação de teste, se os LEDs acendem e apagam de forma ordenada; caso contrário, podemos afirmar que ele apresenta *bouncing*. Além disso, é importante complementar este teste com uma análise do sinal de transição utilizando um osciloscópio[1].

Projeto de Teclado Matricial

Imagine que você está projetando um sistema de segurança para uma porta que utiliza um *keypad* numérico. O desafio é garantir que cada tecla pressionada seja reconhecida de forma precisa e sem erros. Como você utilizaria um microcontrolador para identificar qual tecla foi acionada? Pense em estratégias que poderiam ser empregadas para acionar simultaneamente as colunas e linhas do *keypad*, evitando interpretações equivocadas causadas por latências. Além disso, como você enfrentaria o problema do *bouncing* (“repique” mecânico) das teclas, assegurando que cada pressionamento seja registrado apenas uma vez? Quais dificuldades você antecipa e como poderia superá-las?

Neste projeto, exploraremos como, por meio de programação, podemos completar a matriz do *keypad* para que, ao pressionar uma tecla, sinais únicos sejam gerados para sua identificação. Vamos também aprender a identificar essas teclas pelos sinais nos pinos de entrada e a implementar soluções para evitar o fenômeno do *bouncing*.

1. Crie um novo projeto chamado “Teclado”, usando o *Cube* mas sem inicializar os periféricos no *default*. Ative o *Debug* e ajuste o *clock* do sistema para 64MHz.

2. Para uma melhor organização do código, vamos criar funções para a configuração de cada módulo utilizado no projeto. No escopo de `/* USER CODE BEGIN PFP */`, crie os protótipos das funções a serem implementadas:

---

[1] Em uma escala de dezenas de microsegundos, não observamos nenhum *bouncing* na bancada do SATE.

```
void Config_GPIO(void);
void Config_IRQs(void);
void Config_NVIC(void);
void Config_Timer(void);
void Config_UART(void);
void SendChar(char);
uint8_t LeTecla(void);
```

As seis primeiras funções serão implementadas no arquivo “main.c”. A última função será implementada no arquivo “stm32h7xx_it.c”, para poder passar o valor da tecla às funções em “main.c” sem necessidade de variável global.

3. No escopo de `/* USER CODE BEGIN 4 */`, implemente as funções de inicialização, seguindo modelos de código semelhantes aos usados na detecção do pressionamento da tecla "0" e na detecção de interrupção em uma coluna no projeto anterior. A única diferença é que agora estamos expandindo para 12 teclas e 3 colunas.

```
void Config_GPIO(void) {
	// PORTs usados: D, E, G
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIODEN | RCC_AHB4ENR_GPIOEEN |
RCC_AHB4ENR_GPIOGEN;
	// Port D: PD0, PD1; GPIO outputs OPEN DRAIN
	GPIOD->MODER &= ~(GPIO_MODER_MODE0_Msk |
GPIO_MODER_MODE1_Msk);
	GPIOD->MODER |= (GPIO_MODER_MODE0_0 | GPIO_MODER_MODE1_0); //
MODER = [01]
	GPIOD->OTYPER |= (GPIO_OTYPER_OT0 | GPIO_OTYPER_OT1); // Open
drain
	// Port E: PE4; GPIO output OPEN DRAIN
	GPIOE->MODER &= ~(GPIO_MODER_MODE4_Msk);
	GPIOE->MODER |= (GPIO_MODER_MODE4_0); // MODER = [01]
	GPIOE->OTYPER |= (GPIO_OTYPER_OT4); // Open drain
	// Port G: PG11; GPIO output OPEN DRAIN
	//		PG9, PG10, PG12; GPIO Input
	GPIOG->MODER &= ~(GPIO_MODER_MODE9_Msk |
GPIO_MODER_MODE10_Msk | GPIO_MODER_MODE11_Msk |
GPIO_MODER_MODE12_Msk); // MODER = [00]
	GPIOG->MODER |= (GPIO_MODER_MODE11_0); // MODER = [01]
	GPIOG->OTYPER |= (GPIO_OTYPER_OT11); // Open drain
	// PG9, PG10, PG12: com Pull-up
	GPIOG->PUPDR &= ~(GPIO_PUPDR_PUPD9_Msk |
GPIO_PUPDR_PUPD10_Msk | GPIO_PUPDR_PUPD12_Msk);
	GPIOG->PUPDR |= (GPIO_PUPDR_PUPD9_0 | GPIO_PUPDR_PUPD10_0 |
GPIO_PUPDR_PUPD12_0); // PUPDRx = [01]
	// Coloca as 4 saidas em nivel baixo
	GPIOD->BSRR = GPIO_BSRR_BR0 | GPIO_BSRR_BR1; // PD0 RESET,
PD1 RESET
	GPIOE->BSRR = GPIO_BSRR_BR4; // PE4 RESET
```

```
    GPIOG->BSRR = GPIO_BSRR_BR11; // PG11 RESET
}
void Config_IRQs(void) {
    RCC->APB4ENR |= RCC_APB4ENR_SYSCFGEN; // clock gating de
SYSCFG
    // PG9, PG10, PG12: ligados nas IRQs
    // IRQ9, IRQ10 e IRQ12; Conectar no PORT G
    // 9: bits [7:4] de SYSCFG_EXTICR3
    // 10: bits [11:8] de SYSCFG_EXTICR3
    // 12: bits [3:0] de SYSCFG_EXTICR4
    // PORTG: bits "0110"
    SYSCFG->EXTICR[2]    &=    ~(SYSCFG_EXTICR3_EXTI9_Msk    |
SYSCFG_EXTICR3_EXTI10_Msk);
    SYSCFG->EXTICR[2]    |=    (SYSCFG_EXTICR3_EXTI9_PG    |
SYSCFG_EXTICR3_EXTI10_PG);
    SYSCFG->EXTICR[3] &= ~(SYSCFG_EXTICR4_EXTI12_Msk);
    SYSCFG->EXTICR[3] |= (SYSCFG_EXTICR4_EXTI12_PG);
    // Todos em borda de descida
    EXTI->RTSR1   &=   ~(EXTI_RTSR1_TR9   |   EXTI_RTSR1_TR10   |
EXTI_RTSR1_TR12); // Destiva borda de subida
    EXTI->FTSR1   |=   (EXTI_FTSR1_TR9   |   EXTI_FTSR1_TR10   |
EXTI_FTSR1_TR12); // Ativa borda de descida
    // Mascaras de interrupcao
    EXTI->IMR1   |=   (EXTI_IMR1_IM9   |   EXTI_IMR1_IM10   |
EXTI_IMR1_IM12);
}
void Config_NVIC(void){
    // IRQs 9, 10 e 12
    // Temos EXTI9_5 (vetor 23) e EXTI15_10 (vetor 40), ambos
prioridade 1
   NVIC_SetPriority(EXTI9_5_IRQn, 1); // Configura a prioridade da
  interrupção
    NVIC_EnableIRQ(EXTI9_5_IRQn);

   NVIC_SetPriority(EXTI15_10_IRQn, 1); // Configura a prioridade
  da interrupção
    NVIC_EnableIRQ(EXTI15_10_IRQn);
}
void Config_Timer(void) {
    //Inicializa o TIM6
    RCC->APB1LENR |= RCC_APB1LENR_TIM6EN; // clock gating
    TIM6->EGR |= TIM_EGR_UG_Msk;     // atualizacao manual
    while (TIM6->EGR & TIM_EGR_UG);
    TIM6->PSC = 64 - 1; // prescaler, f = 1MHz
    TIM6->SR &= ~(TIM_SR_UIF); // Limpa flag de update
}
```

4. Para enviar a tecla digitada para o Terminal Serial, é necessário inicializar o módulo USART3 de maneira semelhante ao que foi apresentado no Roteiro 7.

```
void Config_UART(void) {
	RCC->APB1LENR |= RCC_APB1LENR_USART3EN;   // Ativa o clock para USART3
	USART3->CR1 &= ~(USART_CR1_PCE_Msk |   // Desativa o controle de paridade (sem paridade)
		USART_CR1_OVER8_Msk |     // Configura oversampling para 16x (OVER8 = 0)
		USART_CR1_M0_Msk |            // Configura tamanho de caractere (word length = 8 bits)
		USART_CR1_M1_Msk);
	USART3->BRR = 0x1A0A;   // Configura o baud rate para 9600 (considerando um clock de 64 MHz)
	USART3->CR2 &= ~USART_CR2_STOP_Msk; // Configura bits de parada (STOP = 1 bit)
	USART3->PRESC = ~USART_PRESC_PRESCALER_Msk;     //Configurar prescaler (DIV = 1)
	USART3->CR1 |= USART_CR1_UE; // Habilita o USART3
	USART3->CR1 |= USART_CR1_TE | USART_CR1_RE; // Habilita o transmissor e o receptor
	while (!(USART3->ISR & USART_ISR_REACK));  //aguarda a efetivacao de RX
	while (!(USART3->ISR & USART_ISR_TEACK));  //aguarda a efetivacao de TX
}
void SendChar(char c) {
	while(!(USART3->ISR & USART_ISR_TXE_TXFNF)) {}
	USART3->TDR = c;
}
```

5. Abra o arquivo “stm32h7xx_it.c”. Inicialmente, no escopo de `/* USER CODE BEGIN PV */`, vamos implementar uma variável para guardar o valor da tecla, com escopo dentro de todas as funções deste arquivo:

```
static uint8_t tecla = 255;
```

Esta variável irá guardar um valor numérico para cada tecla pressionada. O valor 255 significa que não há nenhuma tecla que não tenha sido processada ainda. Valores de 0 a 9 significam as teclas “0” até “9”. O valor 10 corresponde à tecla “*” e o valor 11, à tecla “#”.

6. Neste arquivo, além das ISRs, também teremos algumas funções auxiliares: definição da linha que permanece em nível lógico baixo, “*delay*” em microssegundos, e a função que passa o valor da tecla às funções no arquivo “main.c”. No escopo de `/* USER CODE BEGIN PFP */`, declare os protótipos das funções:

```
void Linha(uint8_t);
void DelayUs(uint16_t);
uint8_t LeTecla(void);
```

7. No escopo de `/* USER CODE BEGIN 1 */`, implemente as funções de ativação seletiva de linhas:

```
// Ativa uma das linhas (nível lógico baixo) e desativa as demais
// 0: Ativa todas as linhas
// 1 a 4: Ativa a linha correspondente
void Linha(uint8_t l) {
	switch(l) {
	case 0:
		GPIOD->BSRR = GPIO_BSRR_BR0 | GPIO_BSRR_BR1; // PD0 RESET, PD1 RESET
		GPIOE->BSRR = GPIO_BSRR_BR4; // PE4 RESET
		GPIOG->BSRR = GPIO_BSRR_BR11; // PG11 RESET
		break;
	case 1: // L1, PE4
		GPIOD->BSRR = GPIO_BSRR_BS0 | GPIO_BSRR_BS1; // PD0 SET, PD1 SET
		GPIOE->BSRR = GPIO_BSRR_BR4; // PE4 RESET
		GPIOG->BSRR = GPIO_BSRR_BS11; // PG11 SET
		break;
	case 2: // L2, PD1
		GPIOD->BSRR = GPIO_BSRR_BS0 | GPIO_BSRR_BR1; // PD0 SET, PD1 RESET
		GPIOE->BSRR = GPIO_BSRR_BS4; // PE4 SET
		GPIOG->BSRR = GPIO_BSRR_BS11; // PG11 SET
		break;
	case 3: // L3, PD0
		GPIOD->BSRR = GPIO_BSRR_BR0 | GPIO_BSRR_BS1; // PD0 RESET, PD1 SET
		GPIOE->BSRR = GPIO_BSRR_BS4; // PE4 SET
		GPIOG->BSRR = GPIO_BSRR_BS11; // PG11 SET
		break;
	case 4: // L4, PG11
		GPIOD->BSRR = GPIO_BSRR_BS0 | GPIO_BSRR_BS1; // PD0 SET, PD1 SET
		GPIOE->BSRR = GPIO_BSRR_BS4; // PE4 SET
		GPIOG->BSRR = GPIO_BSRR_BR11; // PG11 RESET
		break;
	}
	DelayUs(500); // Atraso para "firmar" os valores nos pinos
}

// Delay em microssegundos usando TIM6 a 1MHz
void DelayUs(uint16_t d) {
```

```
	TIM6->ARR = d - 1;
	TIM6->CNT = 0;
	TIM6->CR1 |= TIM_CR1_CEN; // Inicia a contagem
	while(!(TIM6->SR & TIM_SR_UIF)) {} //Espera flag de update
	TIM6->CR1 &= ~(TIM_CR1_CEN); // Paralisa a contagem
	TIM6->SR &= ~(TIM_SR_UIF); // Limpa flag de update
}
// Passa o valor atual da variavel tecla para outras funcoes,
// voltando o valor da tecla para tecla nao pressionada
uint8_t LeTecla(void) {
	uint8_t tmp;
	tmp = tecla;
	tecla = 255;
	return tmp;
}
```

Note que na função que define a linha em nível baixo, há um "*delay*" com o propósito de garantir que os níveis lógicos se estabilizem nos pinos das linhas antes que o fluxo do programa continue.

8. Agora faltam as ISRs que tratam as interrupções das colunas. Logo abaixo das funções implementadas no item anterior, implemente as ISRs:

```
// ISRs
void EXTI9_5_IRQHandler(void) {
	// Apenas EXTI9, coluna C1 (teclas 1, 4, 7 e *)
	if(EXTI->PR1 & EXTI_PR1_PR9_Msk) {
		DelayUs(20000); // Debounce
		Linha(1); // Testa L1 (tecla 1)
		if(!(GPIOG->IDR & GPIO_IDR_ID9)) {
			tecla = 1;
		} else {
			Linha(2); // Testa L2 (tecla 4)
			if(!(GPIOG->IDR & GPIO_IDR_ID9)) {
				tecla = 4;
			} else {
				Linha(3); // Testa linha 3 (tecla 7)
				if(!(GPIOG->IDR & GPIO_IDR_ID9)) {
					tecla = 7;
				} else {
					Linha(4); // testa linha 4 (tecla *)
					if(!(GPIOG->IDR & GPIO_IDR_ID9)) {
						tecla = 10; // codigo usado para *
					}
				}
			}
		}
		EXTI->PR1 |= EXTI_PR1_PR9_Msk; // Limpa o flag
```

```
            Linha(0); // Coloca todas as linhas em Low
      }
}
void EXTI15_10_IRQHandler(void) {
      // Primeiro EXTI10, coluna C2 (teclas 2, 5, 8 e 0)
      if(EXTI->PR1 & EXTI_PR1_PR10_Msk) {
            DelayUs(20000); // Debounce
            Linha(1); // Testa L1 (tecla 2)
            if(!(GPIOG->IDR & GPIO_IDR_ID10)) {
                  tecla = 2;
            } else {
                  Linha(2); // Testa L2 (tecla 5)
                  if(!(GPIOG->IDR & GPIO_IDR_ID10)) {
                        tecla = 5;
                  } else {
                        Linha(3); // Testa linha 3 (tecla 8)
                        if(!(GPIOG->IDR & GPIO_IDR_ID10)) {
                              tecla = 8;
                        } else {
                              Linha(4); // testa linha 4 (tecla 0)
                              if(!(GPIOG->IDR & GPIO_IDR_ID10)) {
                                    tecla = 0;
                              }
                        }
                  }
            }
            EXTI->PR1 |= EXTI_PR1_PR10_Msk; // Limpa o flag
            Linha(0); // Coloca todas as linhas em Low
      }
      if(EXTI->PR1 & EXTI_PR1_PR12_Msk) { // Se nao for EXTI10,
testa EXTI12, coluna C3 (teclas 3, 6, 9, e #)
            DelayUs(20000); // Debounce
            Linha(1); // Testa L1 (tecla 3)
            if(!(GPIOG->IDR & GPIO_IDR_ID12)) {
                  tecla = 3;
            } else {
                  Linha(2); // Testa L2 (tecla 6)
                  if(!(GPIOG->IDR & GPIO_IDR_ID12)) {
                        tecla = 6;
                  } else {
                        Linha(3); // Testa linha 3 (tecla 9)
                        if(!(GPIOG->IDR & GPIO_IDR_ID12)) {
                              tecla = 9;
                        } else {
                              Linha(4); // testa linha 4 (tecla #)
                              if(!(GPIOG->IDR & GPIO_IDR_ID12)) {
                                    tecla = 11; // Codigo para a tecla
#
```

```
                        }
                    }
                }
            }
            EXTI->PR1 |= EXTI_PR1_PR12_Msk; // Limpa o flag
            Linha(0); // Coloca todas as linhas em Low
        }
}
```

Note o funcionamento das ISRs, de acordo com o algoritmo apresentado na seção Teclado Matricial. Note também que temos um vetor para tratar as interrupções externas de 5 a 9 e outra para as interrupções de 10 a 15. Por isso, é importante verificar a "*flag*" de interrupção para saber qual a interrupção externa que gerou a chamada à ISR. A primeira ISR abrange a primeira coluna, enquanto que a segunda abrange C2 e C3.

Por fim, note os "*delays*" de cerca de 20ms, para evitar erros de leitura por conta do *bounce*. Erros podem acontecer quando a borda de descida inicial é reconhecida, mas durante a varredura ocorre um repique que desfaz momentaneamente a conexão entre linha e coluna. Neste caso, a tecla não será reconhecida durante a varredura. Ou seja, o usuário pressiona a tecla e nada acontece, sendo este um evento de ocorrência aleatória. Com o atraso, a varredura só inicia após a conexão elétrica entre linha e coluna ter se estabilizado.Para evitar que o repique seja interpretado como múltiplas ocorrências da mesma interrupção externa, a *flag* de interrupção é limpa ao final de cada varredura.

9. Voltando agora ao arquivo "main.c", vamos finalmente implementar o programa principal, com a configuração dos periféricos e, no *loop*, a leitura da tecla. Inicialmente, vamos declarar variáveis para guardar o valor da tecla no escopo da função e o valor do caractere a ser transmitido pela USART3. No escopo de `/* USER CODE BEGIN 1 */`, declare as variáveis:

```
uint8_t t = 255;
char c;
```

10. Agora no escopo de `/* USER CODE BEGIN 2 */`, execute a configuração:

```
Config_GPIO();
Config_IRQs();
Config_NVIC();
Config_Timer();
Config_UART();
```

11. Por fim, vamos implementar o *loop*. Abaixo da linha `/* USER CODE BEGIN 3 */`, escreva o código:

```
while(t == 255) {
      t = LeTecla();
}
```

```
if(t == 10) {
      c = '*';
} else if(t == 11) {
      c = '#';
} else {
      c = t + 0x30;
}
t = 255;
SendChar(c);
```

Note que usamos a função “LeTecla” para obter o último valor de tecla pressionada, ou 255 caso nenhuma tecla tenha sido pressionada desde a última chamada da função. Note também a conversão dos valores numéricos das teclas de 0 a 9 nos caracteres ASCII correspondentes, com a soma do valor 0x30, que corresponde ao código ASCII do algarismo “0”.

12. Conecte o teclado matricial no conector correspondente na placa auxiliar (o conector tem o texto “TECLADO” junto ao mesmo). Mantenha o teclado de frente para você e plugue o conector nos 7 pinos mais à esquerda (o pino mais a direita permite a inclusão de uma quarta coluna, para teclados 4x4).

13. Faça o *Build* e o *Debug*. Abra o terminal serial com 9600 *baud*, sem *bit* de paridade e 1 *stop bit*, e execute o programa, pressionando algumas teclas do *keypad* de membrana.

14. Antes de processar um evento de interrupção para varrer as linhas do teclado em busca de uma tecla pressionada, a função `DelayUs(20000)` é chamada, introduzindo um atraso de 20000 microssegundos implementado por *hardware*. Qual temporizador é responsável pela geração desse atraso? Explique a relação entre os valores de *auto-reload* e *prescaler* configurados para esse temporizador, a unidade de tempo do seu contador e o atraso de 20000 microssegundos desejado. Temporizadores são amplamente utilizados para gerar atrasos com alta precisão.

15. Vamos verificar o comportamento do controle subjacente. Experimente pressionar duas ou mais teclas ao mesmo tempo, e veja o que acontece. Como você classifica o efeito observado? Caso você não tem ideia do que está acontecendo, você terá uma resposta ao concluirmos este Roteiro.

16. Pause o programa e remova as linhas que contêm as chamadas “`DelayUs(20000); // Debounce`”. Em seguida, selecione "`Terminate and Relaunch`" para reiniciar o código executável. Continue ("Resume") a execução e verifique se as teclas estão livres de *bouncing*. É uma boa prática implementar técnicas de *debouncing* para garantir que as leituras sejam limpas e precisas em *keypads* de membrana?

FUNDAMENTOS TEÓRICOS

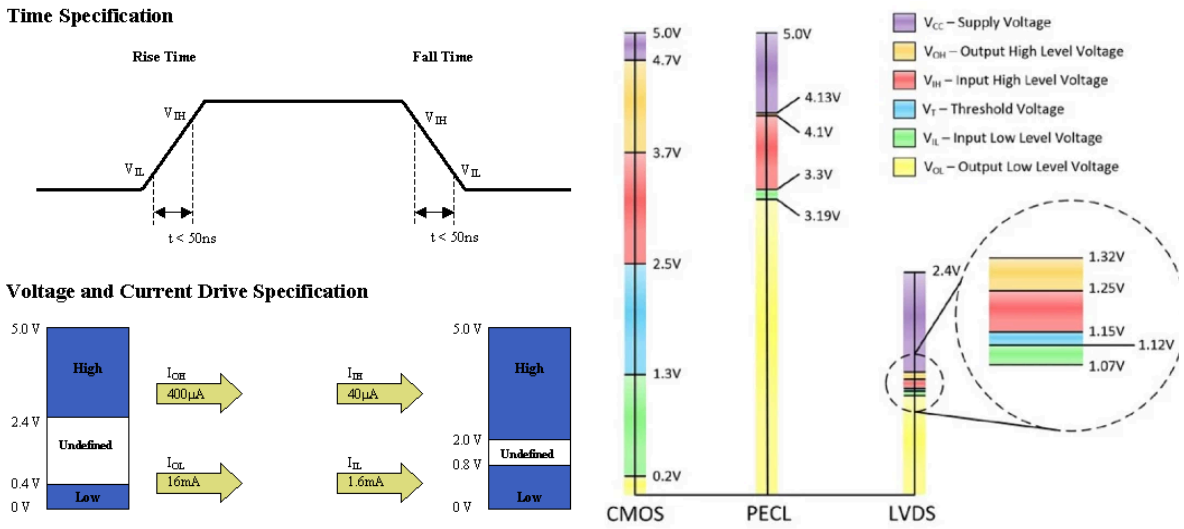
Interfaces paralelas possibilitam a transmissão simultânea de múltiplos *bits* de dados através de diversos canais (pinos), promovendo uma troca de informações rápida e eficiente. Compreender as **características dos sinais digitais** e os **modos de operação dos pinos GPIO** é fundamental para mitigar desafios inerentes ao trabalho com esses sinais. A **programação paralela**, por sua vez, capacita o microcontrolador a controlar múltiplos pinos de forma coordenada, facilitando a comunicação com dispositivos externos. Um exemplo clássico é o protocolo LPT (do inglês *Line Printer Terminal*), historicamente empregado na conexão de impressoras a computadores, que paraleliza a transmissão de dados para alcançar maior velocidade em comparação com interfaces seriais.

No domínio dos sistemas embarcados, diversas **interfaces paralelas personalizadas** são implementadas para o controle de periféricos específicos, como *displays* de 7 segmentos, LCDs e teclados matriciais. Embora operem com sinais digitais, esses dispositivos demandam um controle particularizado através da manipulação dos pinos GPIO, utilizando algoritmos que asseguram a conversão correta entre dados e sinais físicos.

A seguir, exploraremos os detalhes desses tópicos.

CARACTERÍSTICAS DOS SINAIS DIGITAIS

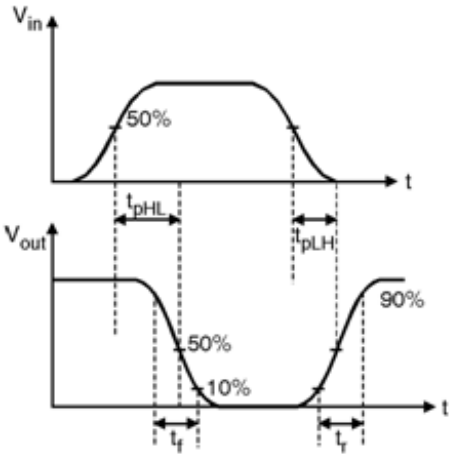
Em sistemas eletrônicos digitais, a comunicação e o controle são baseados em sinais digitais, que operam com dois níveis de tensão distintos representando os estados lógicos “0" e “1”. Idealmente, um sinal digital realiza transições instantâneas entre esses níveis bem definidos, embora na prática essas transições ocorram em um intervalo de tempo finito, conhecidos por **tempo de subida** (em inglês, *rise time*) e **tempo de descida** (em inglês, *fall time*), e os valores lógicos sejam amostrados em momentos específicos.



(Fonte: NI). (Fonte: NI).

Para assegurar a operação correta e a confiabilidade desses sistemas, é imprescindível compreender as características intrínsecas dos sinais digitais e os principais desafios que podem afetá-los. Um problema comum é o **estado flutuante**, que ocorre quando um pino de entrada de um circuito digital não está conectado a um nível lógico definido, resultando em uma tensão indeterminada susceptível a ruídos e capacitâncias parasitas, o que pode levar a leituras imprevisíveis e comportamento errático do sistema. É importante notar que a faixa de tensão correspondente a este estado indefinido não é universal, variando significativamente dependendo da tecnologia de fabricação dos dispositivos digitais utilizados.

As **latências**, que se referem ao atraso no processamento e propagação dos sinais digitais, denotados por t_{pHL} e t_{pLH} na figura que se segue, também representam um desafio, podendo resultar em interpretações equivocadas do estado de eventos, afetando a previsibilidade do sistema.



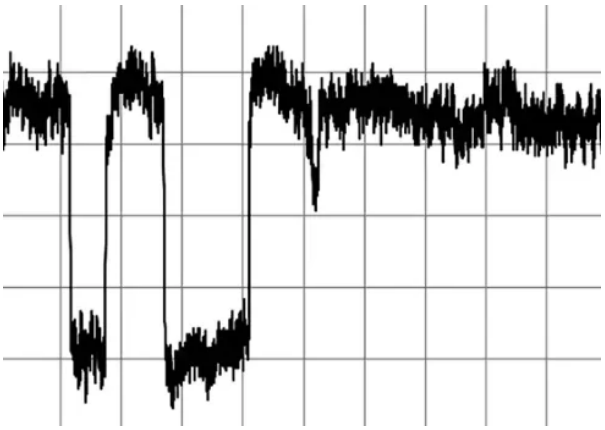
Propagation delay and rise and fall times

(Fonte: Basic Electronics Tutorial).

É importante considerar que, frequentemente, a **potência** (corrente e tensão) que os pinos dos microcontroladores podem fornecer ou suportar é **limitada**. Essa restrição muitas vezes exige a utilização de circuitos complementares, como *drivers* de corrente ou conversores de nível, para garantir a compatibilidade com a carga a ser controlada ou com a fonte de alimentação utilizada.

O **ruído elétrico** pode afetar o desempenho de sistemas eletrônicos, especialmente em ambientes com fortes campos eletromagnéticos ou com aterramento inadequado. Esse tipo de ruído pode se acoplar de forma indesejada às trilhas da placa de circuito impresso e aos pinos dos componentes por meio de **acoplamento eletromagnético**, gerando tensões parasitas. Essas tensões indesejadas somam-se aos sinais legítimos do circuito, podendo, em casos mais

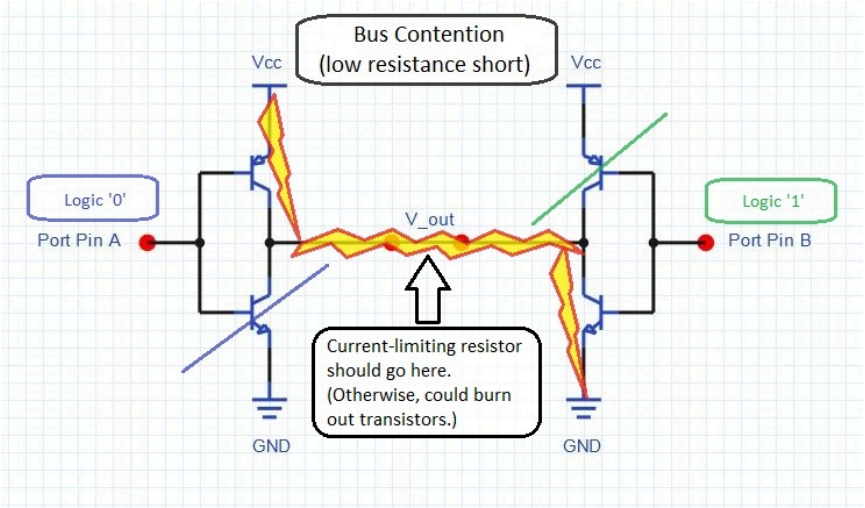
severos, provocar leituras incorretas nos níveis lógicos das entradas digitais, ou seja, o sistema pode interpretar um sinal “0” como “1”, e vice-versa, como ilustra a seguinte figura.



(Fonte: All About Circuits.)

Outro tipo comum de interferência é o **ruído de chaveamento**, que consiste em distúrbios transitórios gerados pelas rápidas variações de corrente durante a comutação de dispositivos semicondutores (como transistores). Essas variações rápidas (altas dv/dt e di/dt) podem provocar flutuações nas tensões de referência e distorcer os níveis lógicos dos sinais, comprometendo a integridade do sistema. Adicionalmente, o **ruído térmico**, inerente à agitação aleatória dos elétrons nos condutores e componentes resistivos, também contribui para a degradação da qualidade do sinal, especialmente em circuitos analógicos de alta sensibilidade. Para mitigar esses efeitos, a adoção de boas práticas de projeto é fundamental, englobando um *layout* otimizado da placa de circuito impresso com roteamento cuidadoso das trilhas, a separação adequada de sinais digitais e analógicos, como veremos no Roteiro 9, um aterramento eficiente com planos de terra bem distribuídos e baixa impedância, e o uso de desacoplamento capacitivo próximo aos componentes sensíveis.

Outro ponto crítico é a **contenção de barramento** (em inglês, *bus contention*), que se manifesta quando dois ou mais dispositivos tentam controlar os mesmos fios dentro de um barramento compartilhado simultaneamente, impondo níveis lógicos opostos. Essa condição pode gerar curtos-circuitos e causar danos físicos permanentes aos componentes, comprometendo a integridade física dos circuitos. Ademais, a integridade dos sinais nos barramentos também pode ser severamente afetada por **reflexões** causadas por terminações inadequadas. Quando os sinais elétricos trafegam pelas trilhas do barramento e encontram uma impedância diferente da sua impedância característica no final da linha (devido à ausência ou incorreção de resistores de terminação), parte da energia do sinal é refletida de volta para a fonte. Essas reflexões podem se sobrepor ao sinal original, causando distorções, *overshoot*, *undershoot* e ruído, levando a erros na interpretação dos níveis lógicos e, consequentemente, a falhas na comunicação entre os dispositivos conectados ao barramento.



(Fonte: Rhea).

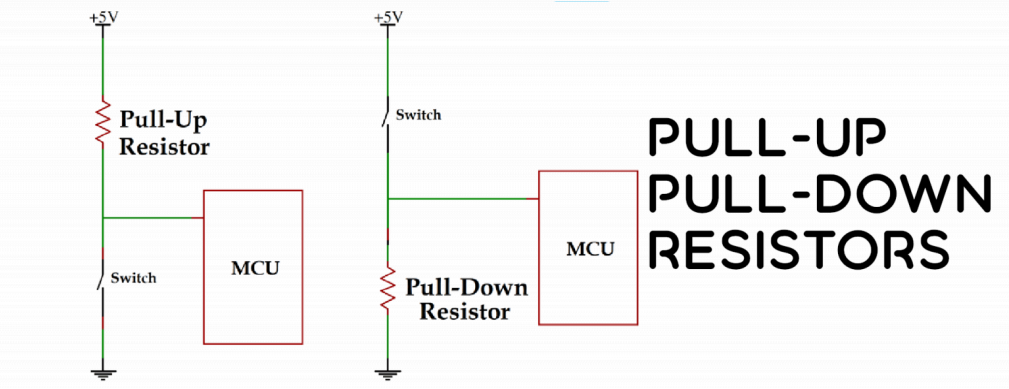
MODOS DE OPERAÇÃO DOS PINOS GPIO

Para assegurar o funcionamento previsível e a integridade física dos componentes em sistemas digitais, é fundamental o tratamento adequado dos sinais digitais, prevenindo problemas como indefinição de estado, contenção de barramento, latências, potências e ruídos elétricos. Um projeto eletrônico bem elaborado, que considera cuidadosamente as conexões, o isolamento contra interferências e o controle de acesso aos barramentos, estabelece a base para a construção de sistemas digitais robustos e confiáveis.

Antes de explorarmos as especificidades da comunicação digital paralela, vamos apresentar os diversos modos de operação dos pinos GPIO em microcontroladores. Conforme abordado em roteiros anteriores, esses pinos de propósito geral oferecem grande flexibilidade, podendo ser configurados como entradas ou saídas digitais. Nesta seção, nosso foco será o entendimento desses modos de operação, bem como características elétricas e funcionais relevantes, demonstrando como essa gestão contribui para mitigar, de forma eficaz, problemas levantados. Entre as características importantes desses pinos, destacam-se:

- ***Slew Rate***: Refere-se à taxa de variação do sinal nas saídas digitais, indicando quão rapidamente um sinal pode mudar de um nível lógico para outro. Essa característica influencia diretamente o tempo de subida e descida dos pulsos digitais. Ajustar o *slew rate* é importante para equilibrar desempenho e integridade do sinal. Um *slew rate* mais lento ajuda a reduzir a geração de ruídos de chaveamento e interferência eletromagnética (EMI), sendo ideal para aplicações sensíveis a ruídos ou com fiação longa. Por outro lado, um *slew rate* mais rápido pode ser necessário em sistemas de alta velocidade, onde a precisão temporal e a resposta rápida são críticas. Muitos microcontroladores permitem configurar o *slew rate* nos pinos GPIO, oferecendo flexibilidade para otimizar o comportamento elétrico conforme os requisitos do projeto.

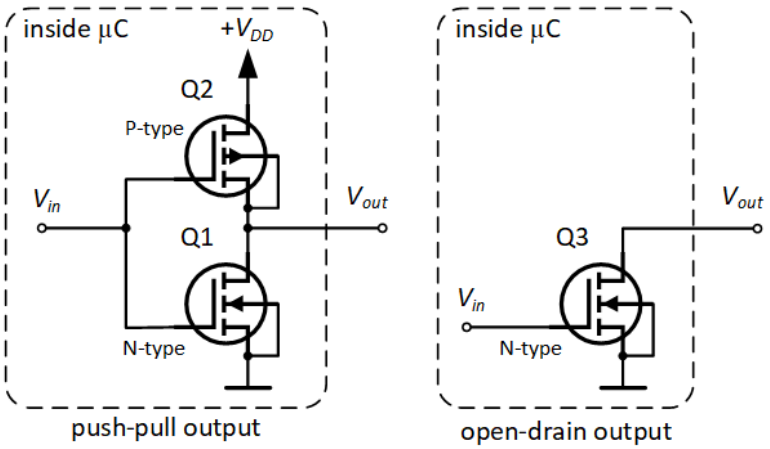
- ***Drive Strength***: Indica a capacidade do pino de fornecer ou absorver corrente. Isso afeta a capacidade de carregar cargas externas, como LEDs ou outros circuitos. A configuração do *drive strength* permite ao programador ajustar a corrente máxima que o pino pode fornecer ou drenar. Essa flexibilidade é importante para compatibilizar o microcontrolador com dispositivos conectados que possuem diferentes requisitos de potência. Um *drive strength* mais alto pode ser necessário para acionar cargas com maior demanda de corrente, enquanto um *drive strength* mais baixo pode ser suficiente para cargas leves e ajudar a reduzir o consumo de energia e interferências eletromagnéticas. Algumas vezes, o programador usa os recursos de limitação de corrente no pino digital para proteção do sistema.
- **Filtro de Entrada** (em inglês, *Input Filter*): **:** Refere-se à capacidade de aplicar filtros digitais nos pinos de entrada, ajudando a eliminar ruídos indesejados e estabilizar a leitura dos sinais. A configuração desses filtros é uma medida importante para amenizar os efeitos de ruídos elétricos que podem se sobrepor ao sinal digital de interesse, prevenindo leituras incorretas e garantindo uma interpretação mais confiável dos dados de entrada.
- ***Pull-up/Pull-down* Resistors**: Estas configurações são empregadas nos pinos de entrada de microcontroladores e outros circuitos digitais para garantir um estado lógico bem definido quando o pino não está diretamente conectado a uma fonte de sinal ativa. Imagine um pino ligado à saída de um dispositivo com saída tri-state ou dreno-aberto que, em determinado momento, pode não estar fornecendo um nível lógico “alto” ou “baixo” explícito. Nessas situações, sem um resistor de *pull-up* ou *pull-down*, a tensão no pino ficaria flutuante, sujeita a ruídos e com um estado lógico incerto.



(Fonte: Robu.in).

- **Modos de operação de saída:** Os modos de operação de saída podem ser classificados principalmente como *open-drain* e *push-pull*. No modo ***open-drain***, o pino se conecta ao GND quando ativado, mas não tem uma conexão direta a VCC, sendo necessário um resistor pull-up para gerar um nível lógico alto. Esse modo é ideal para aplicações que precisam de múltiplos dispositivos em um mesmo

barramento. Embora permita a combinação de sinais de diferentes fontes, a velocidade de resposta pode ser mais lenta devido ao uso do resistor.
Por outro lado, no modo ***push-pull***, o pino pode conduzir tanto para VCC (nível alto) quanto para GND (nível baixo), permitindo transições rápidas entre os estados. Esse modo é comum em saídas digitais que não requerem conexão a múltiplos dispositivos simultaneamente.



(Fonte: Circuit Digest).

A flexibilidade na escolha entre os modos *open-drain* e *push-pull* é crucial para mitigar o problema de contenção de barramento. Quando múltiplos dispositivos estão conectados a um mesmo barramento, é essencial garantir que não haja conflito entre os sinais de saída de diferentes fontes, o que poderia resultar em curto-circuito ou comportamentos imprevisíveis. No modo *open-drain*, cada dispositivo no barramento pode ser configurado para apenas "puxar" para o Terra (GND), sem a possibilidade de forçar um nível lógico alto, o que evita a sobrecarga de sinais conflitantes. Já no modo *push-pull*, como o pino pode gerar tanto nível alto quanto baixo, é importante garantir que apenas um dispositivo controle o barramento em determinado momento, prevenindo a contenção e garantindo transições de estado rápidas e seguras. A capacidade de selecionar o modo adequado permite um controle mais eficiente e seguro de barramentos compartilhados, promovendo a integridade dos sinais e aumentando a confiabilidade do sistema.

As configurações mencionadas, *Slew Rate*, *Drive Strength*, *Input Filter*, *Pull-up/Pull-down* e *Open-Drain/Push-Pull*, são mecanismos que o programador pode ajustar por meio de *software* para influenciar o comportamento dos sinais de entrada e saída, ajudando a mitigar formas específicas de interferência. No entanto, existem características intrínsecas ao projeto do *hardware* dos pinos GPIO que dependem diretamente da física dos dispositivos semicondutores e das estruturas de circuito implementadas durante a fabricação. Essas características não podem ser configuradas via *software*, devido a limitações de velocidade, confiabilidade e acesso direto ao nível físico do silício. Exemplos dessas características incluem:

- **Impedância de Saída**: A resistência que o pino apresenta em sua saída pode afetar a interação com circuitos externos. Em microcontroladores modernos, a impedância de saída dos pinos é geralmente mantida baixa quando o pino está ativamente fornecendo um nível lógico (alto ou baixo) para garantir uma transferência de sinal eficiente e forte para os circuitos conectados. Isso permite que o microcontrolador controle outros componentes com corrente suficiente e com uma queda de tensão mínima. No entanto, em diversas situações, especialmente quando configurado como entrada ou inativo como saída, um pino de microcontrolador apresenta alta impedância. Essa característica ajuda na prevenção de contenção de barramento quando o pino não está controlando ativamente a saída, na minimização da perturbação do circuito que está sendo lido quando configurado como entrada, garantindo leituras precisas, e na redução de interferências em circuitos multiplexados, permitindo que um mesmo pino seja usado para diferentes funções sem influenciar outros circuitos inativos.
- **Proteção de Descarga Eletrostática** (em inglês, *Electrostatic Discharge* - ESD): Refere-se a mecanismos de proteção integrados nos pinos do microcontrolador para atenuar os danos causados por descargas eletrostáticas, aumentando a onfiança e a longevidade dos microcontroladores em diversas aplicações. A ESD ocorre quando há uma transferência repentina de eletricidade estática entre objetos com diferentes potenciais elétricos, podendo gerar picos de tensão extremamente altos e correntes capazes de danificar permanentemente componentes semicondutores sensíveis, como os microcontroladores.

Assim, módulos GPIO são a escolha preferencial para implementar E/S em diversas aplicações, especialmente com sinais não padronizados, incluindo as entradas e saídas paralelas, **pois** oferecem versatilidade (configuráveis como entrada ou saída), simplicidade de configuração (sem necessidade de hardware complexo adicional), custo-benefício (ampla disponibilidade de pinos), baixa latência (transferência paralela veloz dentro do mesmo PORT) e suporte de bibliotecas de desenvolvimento.

COMUNICAÇÃO PARALELA VIA GPIO

Em sistemas embarcados e microcontroladores, os pinos GPIO são frequentemente usados para comunicação paralela, permitindo o envio e recebimento simultâneo de múltiplos *bits* de dados por meio de vários canais. Cada pino pode ser configurado para representar um único ***bit* de dados**, e quando agrupamos vários pinos, eles formam um "barramento de dados de entrada/saída**"**. Os registradores do GPIO são mapeados no espaço de memória do microcontrolador e podem ser acessados com uma única instrução. Ao escrever nesses registradores, o microcontrolador envia os dados diretamente para o barramento, que conecta o processador aos registradores do GPIO, e esses, por sua vez, transmitem os *bits* para os pinos, tudo dentro de um único ciclo de instrução. Por exemplo, se configurarmos 8 pinos e escrevermos um *byte* de dados no registrador correspondente, esses 8 *bits* serão transmitidos simultaneamente, aproveitando a comunicação paralela. Além de transmitir dados, os pinos GPIO também podem ser usados para **controle de sinal**. Isso significa que podemos

configurar os pinos para indicar quando os dados estão prontos para ser enviados ou lidos, ou para sinalizar quando o dispositivo está pronto para receber mais dados.

Embora essa abordagem aumente a velocidade da comunicação, ela também traz desafios importantes. A gestão dos pinos não envolve apenas a configuração do *hardware*, mas também exige uma programação cuidadosa para garantir que todos os sinais de dados e controle sejam enviados e recebidos corretamente. Apesar de os pinos GPIO serem relativamente fáceis de configurar e controlar, a implementação de uma comunicação paralela via GPIO traz consigo alguns desafios, principalmente relacionados à sincronização e ao controle dos sinais de dados.

Em uma comunicação paralela, todos os *bits* devem ser transmitidos simultaneamente e lidos de forma sincronizada. Isso exige uma **sincronização precisa entre os pinos**. Um pequeno atraso entre os sinais pode resultar em dados corrompidos ou lidos de forma incorreta. O controle preciso do tempo de ativação e desativação dos pinos é fundamental. Uma abordagem de *software* que não seja suficientemente eficiente pode causar erros de leitura, já que o processador pode não conseguir atualizar todos os pinos simultaneamente. O uso de interrupções ou temporizadores de *hardware* (caso o microcontrolador os suporte) pode ajudar a sincronizar a mudança dos estados dos pinos com maior precisão. Além disso, *buffers* de *hardware* podem ser usadas para minimizar a latência entre os sinais.

A comunicação paralela tende a ser mais suscetível a **interferências e ruídos**, especialmente em longas distâncias. Isso ocorre porque múltiplos sinais estão sendo transmitidos simultaneamente, o que pode criar conflitos e problemas de *crosstalk* entre os fios. Manter a integridade dos dados é um desafio importante quando se trabalha com comunicação paralela, já que sinais diferentes podem interagir e interferir entre si. O uso de linhas curtas para as conexões de dados, a implementação de cabos blindados ou o uso de técnicas de codificação de erros podem ajudar a reduzir o impacto da interferência. Além disso, sistemas de comunicação paralela de alta velocidade podem utilizar *buffers* ou registradores para armazenar temporariamente os dados, minimizando a chance de perda.

Outro desafio significativo está na **latência do sistema**, especialmente em sistemas de baixo desempenho ou quando muitos pinos GPIO estão sendo utilizados para a comunicação. A manipulação dos pinos via *software* pode levar a atrasos que afetam a eficiência da transmissão, principalmente quando se comunica com dispositivos que exigem alta taxa de transferência de dados. A programação direta dos pinos GPIO pode não ser suficientemente rápida para suportar altas taxas de transmissão, especialmente se o processador principal estiver realizando outras tarefas simultaneamente. Utilizar pinos dedicados para funções específicas de comunicação (como pinos de controle) e liberar o processador principal para outras tarefas pode melhorar o desempenho. O uso de periféricos de comuicação serial, como SPI (Serial Peripheral Interface), I2C ou UART, também pode ser uma alternativa para contornar as limitações da comunicação paralela direta.

Como a comunicação paralela exige muitos pinos de I/O, ela pode esgotar rapidamente os recursos de um microcontrolador, especialmente em sistemas com poucos pinos disponíveis.

Isso limita a escalabilidade do sistema. O **número de pinos GPIO disponíveis** para a comunicação paralela pode ser insuficiente em sistemas com recursos limitados. Usar expansores de I/O, como os *chips* 74HC595 (para saída) ou 74HC165 (para entrada), pode permitir a ampliação do número de pinos disponíveis para a comunicação sem sobrecarregar o microcontrolador. Outra solução seria usar barramentos de dados seriais (como SPI ou I2C) para reduzir a quantidade de pinos necessários.

COMUNICAÇÃO PARALELA INTERMODULAR

Nos microcontroladores modernos, como o STM32H7, a comunicação entre os diferentes módulos do sistema pode ocorrer de duas formas principais: paralela ou serial. O tipo de comunicação depende do tipo de interface usada. Muitos microcontroladores, como o STM32H7, utilizam uma combinação de ambas as abordagens, mas a comunicação paralela é mais comum em alguns barramentos importantes, como o AHB (do inglês *Advanced High-performance Bus*), APB1 (do inglês *Advanced Peripheral Bus 1*), e APB2 (do inglês *Advanced Peripheral Bus 2*), para conectar o núcleo do processador a memória e diferentes dispositivos. Esses barramentos possuem várias linhas que carregam sinais de endereço, dados e controle, permitindo uma **comunicação paralela intermodular** eficiente.

Embora a comunicação paralela proporcione transferências rápidas de dados, ela enfrenta alguns desafios práticos. Um dos maiores problemas é o *skew*. O ***skew*** ocorre quando os sinais dos diferentes *bits* chegam ao receptor em momentos ligeiramente diferentes, causando **dissonância temporal**. Isso pode resultar em erros de leitura ou falhas na comunicação. Para resolver esse problema, é preciso garantir que todos os *bits* cheguem no momento certo. Isso é feito por meio de um bom **g**erenciamento dos sinais, como o uso de mascaramento e escrita atômica[2], que ajudam a minimizar o impacto do *skew* e tornam a comunicação mais confiável. Além disso, técnicas de projeto de *hardware* (como roteamento cuidadoso da PCB e controle de impedância) também ajudam a reduzir o efeito do *skew*, tornando a comunicação mais precisa.

Os barramentos de alto desempenho do STM32H7 são projetados com o objetivo de minimizar o *skew*. No entanto, devido a fatores inerentes à física dos materiais, variações de fabricação e as condições de operação, é extremamente difícil garantir uma ausência total de *skew* em todas as situações. Os projetistas de sistemas que utilizam esses microcontroladores e seus barramentos de alta velocidade ainda precisam estar cientes da possibilidade de *skew* e considerar técnicas de projeto adequadas para mitigar seus efeitos, especialmente em interfaces paralelas críticas.

A arquitetura interna de um microcontrolador moderno, como o STM32H7, é projetada de forma que cada periférico tenha um projeto de *hardware* específico, otimizado para sua

---

[2] Uma **escrita atômica** é uma operação de escrita em memória (ou em um recurso compartilhado) que é percebida como **indivisível.**

função, e seus registradores dedicados. A comunicação entre o processador e esses registradores é facilitada pelo mapeamento dos registradores na memória (*Memory-Mapped I/O*) e pelo uso de barramentos internos projetados para serem compatíveis com a largura desses registradores, permitindo o acesso e a configuração eficientes dos periféricos através de operações de leitura e escrita na memória.

Especificamente, os módulos GPIO do STM32H7A possuem barramentos internos paralelos e registradores de 32 *bits* para controlar os pinos. Isso garante que a comunicação entre o processador e o GPIO seja rápida e eficiente. Em termos práticos, quando se escreve um valor em um endereço de memória em que o registrador de dados de saída de um GPIO está mapeado, todos os *bits* desse valor são transferidos em paralelo pelos barramentos internos. Se os pinos GPIO estiverem conectados a um dispositivo físico externo, os sinais são propagados paralelamente para o dispositivo, configurando uma comunicação paralela onde múltiplos sinais correspondentes aos *bits* são transmitidos simultaneamente entre o microcontrolador e o dispositivo no mundo exterior. Dessa forma, os pinos GPIO naturalmente se apresentam como a escolha ideal para implementar interfaces de E/S paralelas nos microcontroladores.

PROGRAMAÇÃO PARALELA DE BITS

Quando falamos de programação paralela de *bits* (ou programação simultânea de *bits*), estamos nos referindo a uma técnica em que todos os valores de *bits* são atualizados ao mesmo tempo, ou seja, durante o mesmo ciclo de relógio. Para que isso aconteça, é essencial que todos os *bits* que estão sendo alterados possam ser transferidos e manipulados por uma única instrução. Ou seja, todos os *bits* precisam ser lidos e modificados simultaneamente em um único ciclo de processamento. Na programação sequencial, por sua vez, a modificação dos *bits* ocorre um a um, ou seja, os *bits* são alterados em etapas, com uma instrução por vez. Para isso, uma técnica comum é o **mascaramento** (em inglês, *masking*), onde se usa máscaras de *bits* para modificar seletivamente os *bits* de um registrador ou memória através da estratégia leitura-modificação-escrita (em inglês *read-modify-write*). A operação lógica AND, por exemplo, com uma máscara de *bits*, força os *bits* correspondentes a “0”, enquanto a operação OR com a máscara força os *bits* correspondentes a “1”.

No entanto, quando se precisa alterar *bits* de forma mista, ou seja, atualizar tanto para “0” quanto para “1” ao mesmo tempo, o uso de máscaras simples com as operações AND e OR pode ser problemático. Isso pode gerar uma situação conhecida como *skew* — uma variação no tempo de atualização dos *bits*. O *skew* ocorre quando, em vez de todos os *bits* serem atualizados simultaneamente, alguns *bits* são modificados um pouco antes ou depois de outros, o que pode levar a erros de sincronização. Esse problema é especialmente crítico em sistemas onde a sincronia precisa ser perfeita, como no caso de registradores de controle, que exigem modificações simultâneas de múltiplos *bits* para desbloquear mecanismos de proteção ou para habilitar/invalidar funções específicas (como os registradores de chave `RTC_WPR` e

SBC_AIRCR). Se os *bits* de controle não forem atualizados de maneira sincronizada, o sistema pode falhar em executar corretamente essas funções, como explicado no Roteiro 5.

Uma solução para garantir que a alteração de múltiplos *bits* aconteça simultaneamente em um único ciclo de instrução é utilizar uma abordagem de leitura-modificação-escrita "estendida". A ideia é realizar uma modificação atômica do registrador ou valor de memória depois de montar uma palavra com os *bits* atualizados:

1. **Leitura do Estado Atual:** O valor atual do registrador é lido para obter o estado dos *bits* antes da modificação.
2. **Modificação do Valor:** As operações necessárias são aplicadas para alterar apenas os *bits* específicos que devem ser atualizados, utilizando uma sequência de operadores lógicos como AND, OR e XOR.
3. **Escrita do Novo Valor:** O valor modificado é então escrito de volta no registrador em uma única operação de atribuição, garantindo que a modificação de todos os *bits* do registrador ocorra simultaneamente.

Em microcontroladores modernos, alguns registradores operam como **registradores de comando**. Cada escrita neles dispara uma ação específica, a exemplo dos registradores `GPIOx_BSRR` e `RTC_WPR` do microcontrolador STM32H7A. Os valores escritos nesses registradores geralmente não são legíveis (em inglês, *readable*), pois não armazenam dados de forma persistente como um registrador de dados. Tipicamente, programa-se em paralelo o valor do comando correspondente no campo de *bits* reservado para que o *hardware* consiga decodificar o código em ações predefinidas.

PROTOCOLO DE COMUNICAÇÃO PARALELA

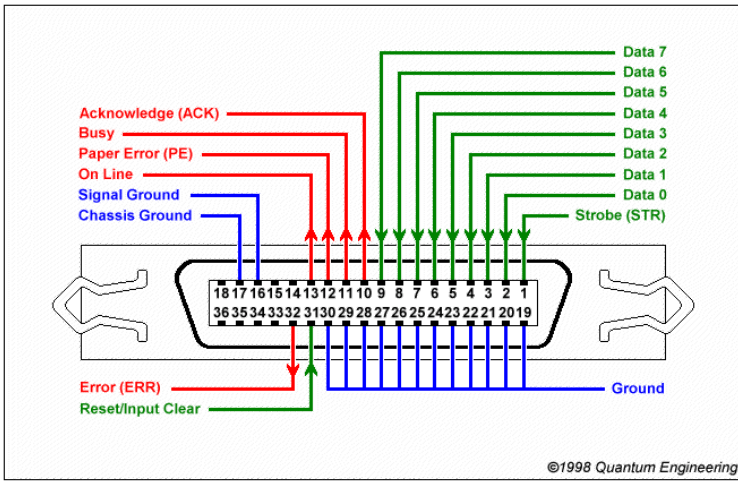
Assim como em comunicações seriais assíncronas, a troca bem-sucedida de dados entre transmissor e receptor em sistemas paralelos depende da adesão a um protocolo. Este protocolo deve definir a organização dos dados e as regras de sincronização, garantindo a compreensão mútua das informações transmitidas. Adicionalmente, o controle preciso das operações dos pinos GPIO é crucial, pois eles servem como os canais físicos para o envio e recebimento de sinais digitais na comunicação paralela.

Na comunicação serial, os *bits* são enviados um após o outro, em sequência, utilizando um único canal (ou alguns fios para controle). Vimos no Roteiro 7 que a sincronização se refere principalmente a garantir que o receptor saiba quando um *bit* começa e termina, o que é geralmente feito através de *bits* de início e parada (em comunicação assíncrona) ou um sinal de *clock* compartilhado (em comunicação síncrona). Na comunicação paralela, por sua vez, múltiplos *bits* são enviados ao mesmo tempo por diferentes canais. Temos o desafio de sincronizar a chegada simultânea de múltiplos *bits*, garantindo que todos esses *bits* enviados simultaneamente pelo transmissor sejam amostrados pelo receptor no **mesmo instante**. Diferenças no comprimento dos fios, nas capacitâncias parasitas, nas características dos

drivers e receptores podem causar variações nos tempos de propagação dos sinais nos diferentes canais. Isso pode levar a "*skew*" de *bits*. Se o receptor amostrar os *bits* em momentos ligeiramente diferentes, o dado recebido pode ser incorreto.

Protocolos de comunicação paralela empregam sinais de controle (*handshaking*) para mitigar problemas de sincronização decorrentes da transmissão simultânea, indicando a validade dos dados e a prontidão do receptor. Um dos protocolos paralelos mais tradicionais e amplamente reconhecidos é o LPT (do inglês *Line Printer Terminal*), historicamente utilizado na conexão de impressoras aos primeiros computadores pessoais. A porta paralela LPT exemplifica a estruturação da comunicação paralela e o estabelecimento da sincronização entre os dispositivos para a troca de informações.

O protocolo LPT, baseado no padrão **Centronics**, permite a transferência simultânea de múltiplos *bits* de dados (tipicamente 8 *bits*) através de várias linhas físicas, resultando em taxas de transferência superiores às das conexões seriais tradicionais. O conector padrão utilizado é o **DB-25**, que possui 25 pinos dedicados a dados, controle, *status* e aterramento. Além disso, o LPT incorpora sinais de controle essenciais para a sincronização da transferência de dados entre o computador e a impressora, como os sinais de ***strobe*** (para indicar a validade dos dados) e ***acknowledge*** (para confirmar o recebimento dos dados).



O sinal *strobe* e o sinal *acknowledge* são fundamentais na comunicação entre um computador e uma impressora, especialmente no contexto do ***handshaking,*** que é o processo de sincronização de comunicação entre dois dispositivos como vimos no Roteiro 7. O **sinal *strobe*** é um sinal de controle enviado pelo computador à impressora para indicar que um novo conjunto de dados está disponível para leitura. Quando o computador coloca os dados nos pinos de dados, ele pulsa o sinal strobe por um breve período. Esse pulso informa à impressora o momento exato em que os dados no barramento devem ser lidos e armazenados, facilitando a sincronização da leitura.

Em resposta, a impressora envia o **sinal *acknowledge*** de volta ao computador para confirmar o recebimento e o processamento dos dados. Ao receber o sinal *acknowledge*, o computador

sabe que pode enviar o próximo conjunto de dados com segurança. Essa confirmação evita a perda de informações, pois o computador aguarda a confirmação da impressora antes de prosseguir com a transmissão. A troca sincronizada dos sinais *strobe* e *acknowledge* minimiza erros de comunicação, garantindo que a impressora processe os dados na ordem correta e dentro do tempo esperado.

O **sinal *busy*** complementa o *handshaking* do protocolo LPT, atuando como um indicador do estado de prontidão do receptor (tipicamente a impressora). Enquanto o sinal *strobe* notifica a impressora que dados válidos estão presentes nas linhas de dados, e o sinal ACK sinaliza que a impressora recebeu o *byte* de dados, o sinal *busy* serve como um "freio". Quando a impressora está ocupada processando os dados recebidos, imprimindo ou realizando alguma outra operação interna, ela mantém a linha *busy* em um estado ativo (geralmente nível lógico alto). Isso informa ao transmissor que ele deve aguardar e não enviar o próximo *byte* de dados. Somente quando a impressora estiver pronta para receber mais informações e puder processá-las, ela desativará o sinal *busy* (levando-o ao nível lógico baixo), sinalizando ao computador que a transmissão pode prosseguir com o próximo *byte*, reiniciando o ciclo com a ativação do sinal *strobe*.

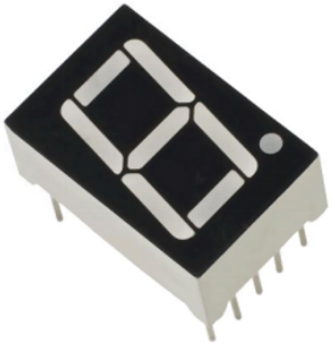
Embora a popularidade da porta paralela **LPT** tenha diminuído consideravelmente com a ascensão das conexões seriais e USB, ela ainda é utilizada em alguns contextos industriais e sistemas legados que precisam operar com equipamentos mais antigos. A relação com o padrão Centronics é fundamental, pois ele estabeleceu as bases para a comunicação LPT, servindo como um modelo de referência para a conexão de impressoras paralelas.

INTERFACES PARALELAS PERSONALIZADAS

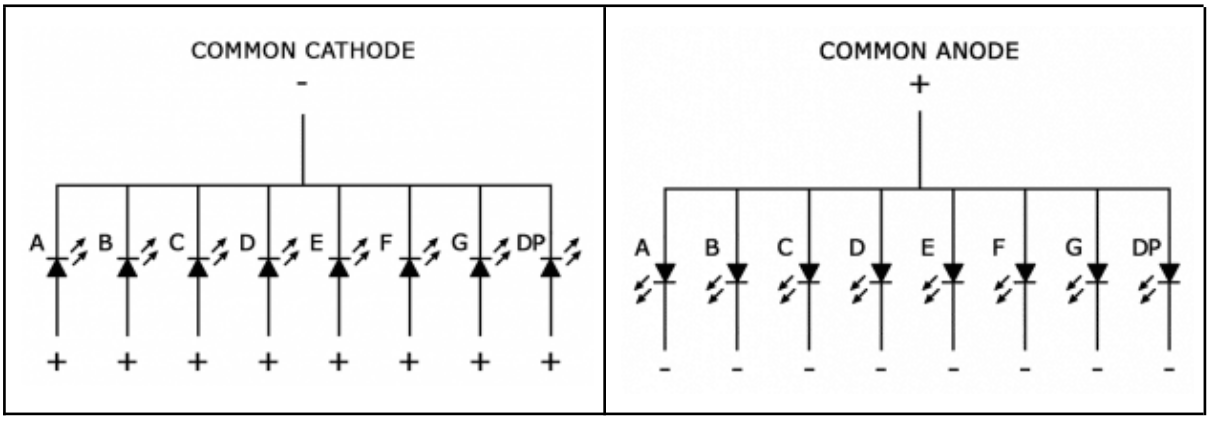
Diversos dispositivos, como *displays* de 7 segmentos, LCDs 16x2 e teclados matriciais, utilizam interfaces paralelas que não aderem a um padrão de comunicação específico. Como os microcontroladores geralmente não dispõem de um periférico dedicado para controlar esses dispositivos em paralelo, o desenvolvedor precisa configurar e gerenciar manualmente os pinos GPIO aos quais eles estão conectados, implementando por *software* a geração dos sinais físicos requeridos.

DISPLAY DE 7 SEGMENTOS

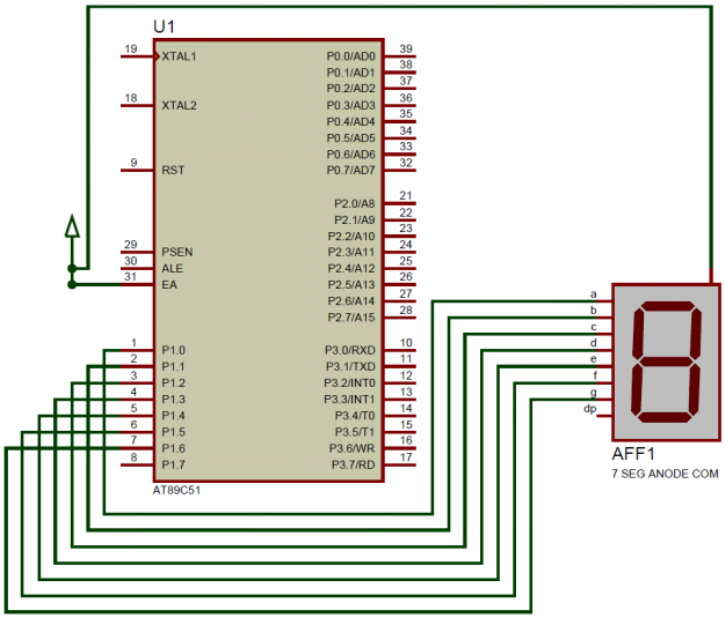
Um *display* de 7 segmentos é um dispositivo de exibição composto por 7 LEDs dispostos de forma a formar números decimais (0-9) e alguns caracteres alfanuméricos básicos. Ele é amplamente utilizado em dispositivos como relógios digitais, painéis de instrumentos, calculadoras e outros dispositivos de exibição simples. Cada segmento do *display* é representado por um LED que pode ser aceso ou apagado individualmente, dependendo da combinação de sinais enviados. A disposição dos LEDs forma 7 segmentos (A a G), e dependendo da ativação de cada um deles, diferentes números ou caracteres podem ser exibidos.



Os *displays* de 7 segmentos podem ser classificados em duas categorias principais, dependendo da maneira como os segmentos são acionados: catodo comum (CC) e anodo comum (AC). No *display* de **catodo comum**, todos os catodos dos LEDs (os terminais negativos) estão conectados a um pino comum. Para acender um segmento, é necessário aplicar um nível baixo (0V) no pino correspondente ao segmento. Ou seja, a corrente flui do pino do segmento para o catodo comum. Enquanto no *display* de **anodo comum**, todos os anodos dos LEDs (os terminais positivos) estão conectados a um pino comum. Para acender um segmento, é necessário aplicar um nível alto (Vcc) no pino correspondente ao segmento. Ou seja, a corrente flui do anodo comum para o pino do segmento.



A seguinte figura ilustra a conexão de um *display* de 7 segmentos com um microcontrolador 8051 através de 7 pinos além de um terra comum.



(Fonte: Spectra One).

O controle de um *display* de 7 segmentos pode ser feito de forma simples utilizando pinos GPIO de um microcontrolador. Cada segmento (A, B, C, D, E, F, G) é controlado por um pino de GPIO, configurado como saída digital. O microcontrolador envia sinais de nível alto (1) ou nível baixo (0) aos pinos correspondentes aos segmentos, dependendo se o *display* é de catodo ou anodo comum.

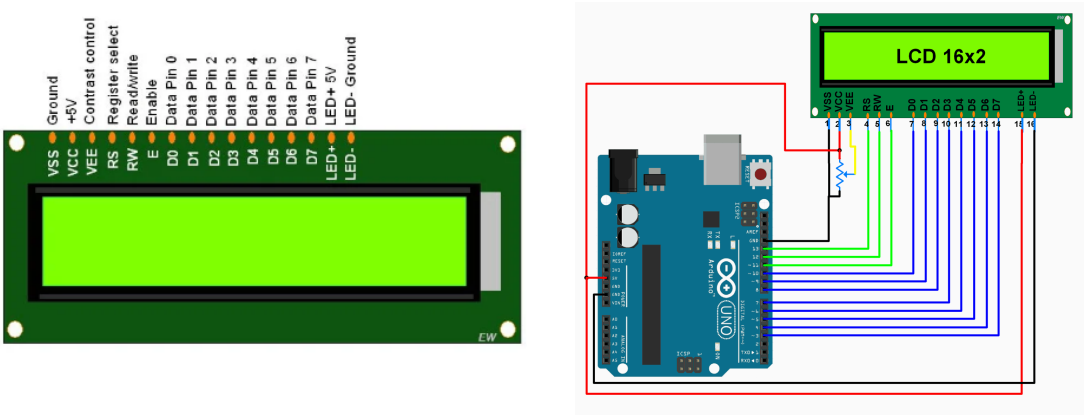
A lógica para acender os números em um display de 7 segmentos é simples. Por exemplo, para exibir o número **'8'** (que acende todos os segmentos) num *display* de catodo comum, os pinos correspondentes a todos os segmentos (A a G) devem ser configurados simultaneamente com nível alto (Vcc). Para exibir outros números, uma combinação diferente de segmentos deverá ser acionada em paralelo.

LCD 16x2

Um **LCD 16x2** é um *display* de cristal líquido que possui 16 colunas e 2 linhas de caracteres, sendo capaz de exibir até 32 caracteres. Cada caractere é formado por uma matriz de 5x8 *pixels*, onde os segmentos são acesos ou apagados para formar as letras, números ou símbolos. É amplamente utilizado em sistemas embarcados, como controladores, microcontroladores e sistemas de *display* simples.

O LCD 16x2 contém duas áreas de memória importantes para o controle da exibição: DDRAM (do inglês *Display Data RAM*) e CGRAM (do inglês *Character Generator RAM*). A **DDRAM** é a memória que armazena os caracteres a serem exibidos no *display*. Cada posição na memória DDRAM corresponde a uma posição no *display* físico, ou seja, o conteúdo armazenado nessa memória será exibido nas 16 colunas de cada uma das 2 linhas do *display*. O microcontrolador escreve os dados na DDRAM para determinar o que será mostrado nas posições específicas do *display*. A **CGRAM** é uma área de memória usada para

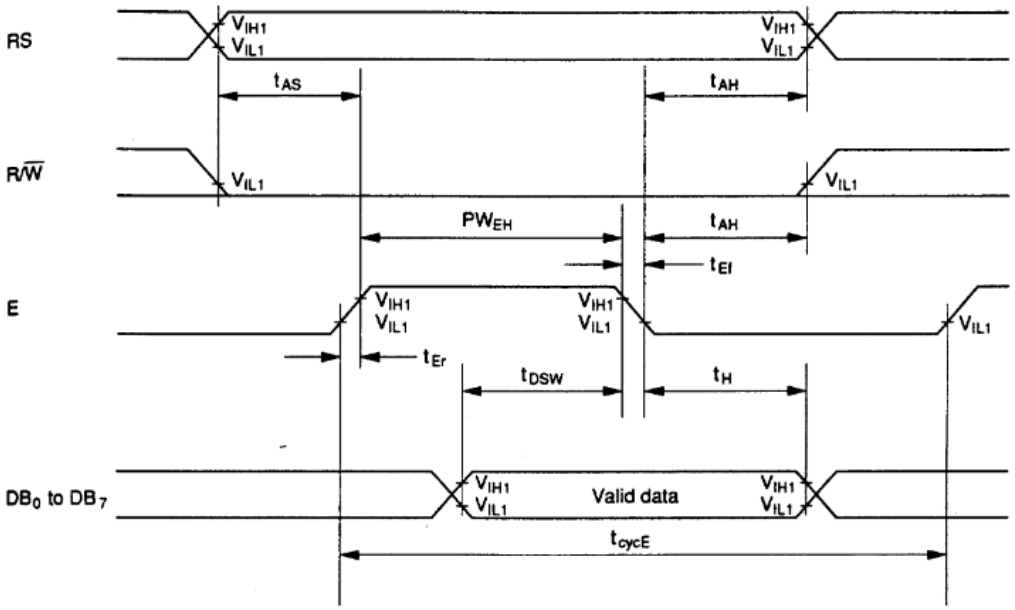
armazenar fontes de caracteres personalizadas. Ela contém até 8 caracteres personalizados que podem ser definidos pelo usuário. Cada caractere na CGRAM é formado por uma matriz de 5x8 *pixels*, e a memória pode armazenar até 8 padrões de caracteres personalizados (geralmente de 0x00 a 0x07). Os caracteres personalizados criados na CGRAM podem ser exibidos no LCD quando o endereço correspondente for escrito na DDRAM.



(Fonte: ALLELCO)

O controle do LCD 16x2 é feito através de pinos GPIO do microcontrolador, que se comunicam com o *display* para enviar comandos e dados. O *display* pode operar no modo paralelo ou serial (via adaptador I2C). No modo paralelo, os pinos principais envolvdios no controle do LCD são pinos de dados (D0 a D7) e pinos de controle. Os **pinos de dados** transmitem dados e comandos (como limpar a tela, mover o cursor etc.) para o LCD. Em uma configuração de 8 *bits* (modo de 8 *bits*), 8 pinos GPIO são usados. E os **pinos de controle,** RS, R/W e E**,** envolvidos diretamente na transmissão de dados ou comandos para o LCD. As relações temporais dos sinais físicos nesses pinos são mostradas na figura que se segue.

Write Operation



Item	Symbol	VDD=5V		VDD=3.3V		Unit
		Min	Max	Min	Max	
Enable cycle time	tcycE	500	--	1000	--	ns
Enable pulse width	PWEH	230	--	450	--	
Enable rise/fall time	tEr,tEf	--	20	--	25	
Address set-up time (RS, R/W to E)	tAS	40	--	60	--	
Address hold time	tAH	10	--	20	--	
Data set-up time	tDSW	80	--	195	--	
Data hold time	tH	10	--	10	--	

Fonte: Datasheet.

O sinal **RS** (do inglês *Register Select*) define a natureza do dado transmitido: comando (RS = 0) ou dado de exibição (RS = 1). Dados de exibição são armazenados na DDRAM, enquanto comandos são interpretados como sinais de controle. O *datasheet* do LCD especifica o tempo de processamento necessário para cada tipo de operação, para garantir o processamento completo antes do envio subsequente. O sinal **R/W** (do inglês *Read/Write*) controla a direção da operação: leitura do LCD (R/W = 1) ou escrita no LCD (R/W = 0). Em aplicações de escrita, este pino geralmente é mantido em nível baixo. Por fim, o sinal **E** (do inglês *Enable*) sinaliza ao LCD que um dado ou comando válido está presente nos pinos de dados e pode ser processado. O processamento é tipicamente disparado por uma transição de nível alto para baixo neste pino (pulso de descida).

Portanto, para enviar um dado ou comando ao LCD com o pino R/W mantido em nível baixo, o procedimento envolve configurar o nível do pino RS (para indicar dado ou comando), apresentar o dado ou comando nos pinos de dados e gerar um pulso de descida no pino E para habilitar a leitura e execução pelo *display*.

Rickey disponibiliza um tutorial minucioso sobre interfaceamento de um LCD para microcontroladores.

TECLADO MATRICIAL

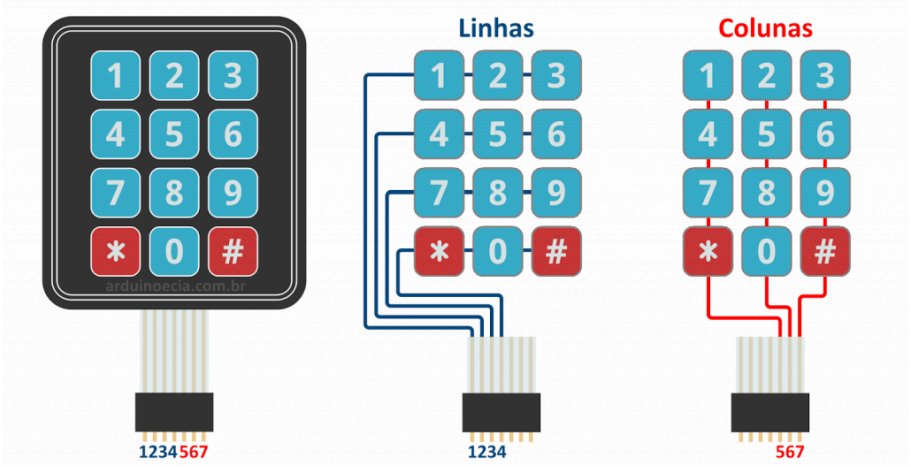
Os teclados matriciais são amplamente usados em sistemas eletrônicos devido à sua eficiência na redução do número de pinos necessários para ler um grande número de teclas. Este tipo de teclado organiza as teclas em uma matriz de linhas e colunas, o que permite escanear várias teclas com uma quantidade reduzida de pinos de entrada e saída para detectar quais teclas estão sendo pressionadas.

Vamos considerar um teclado numérico simples com 12 teclas, dispostas em 3 colunas e 4 linhas. Este tipo de configuração requer apenas 7 pinos no microcontrolador: 3 pinos de entrada para as colunas e 4 pinos de saída para as linhas, em vez de 12 pinos se cada tecla fosse monitorada individualmente. A grande vantagem dos teclados matriciais é que eles permitem monitorar um número maior de teclas sem a necessidade de muitos pinos do microcontrolador. Em vez de utilizar 12 pinos para 12 teclas, usamos 3 pinos para as colunas e 4 pinos para as linhas, totalizando 7 pinos. A fórmula geral para um teclado matricial é a seguinte:

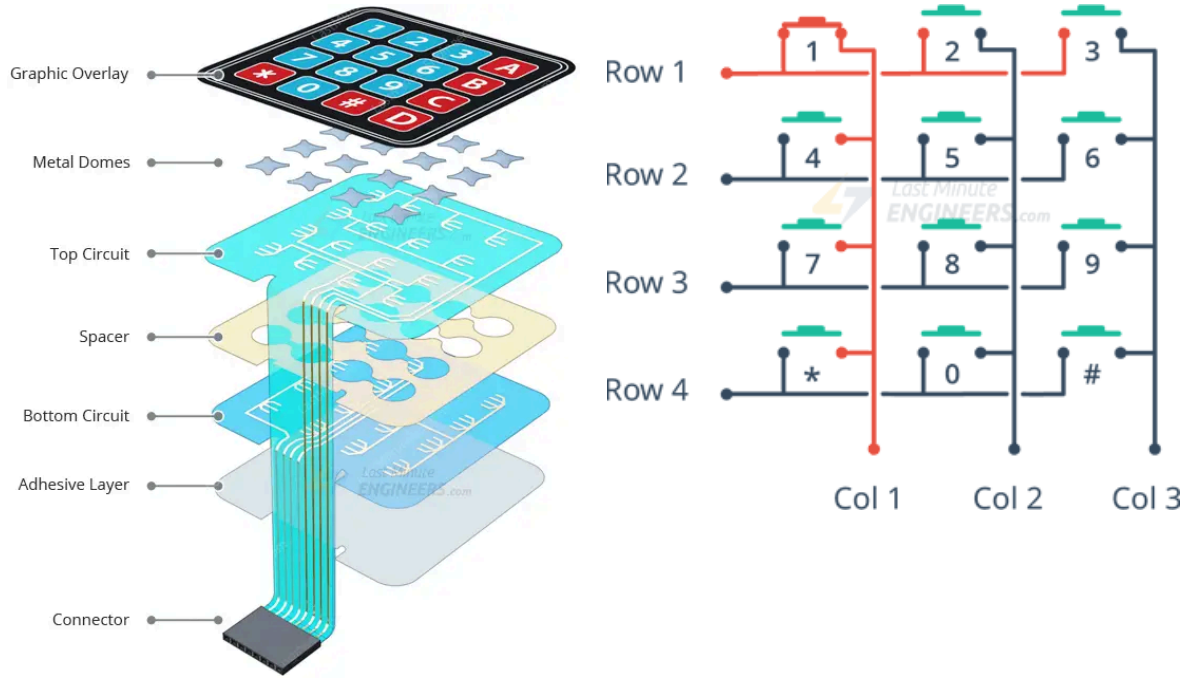
$$\text{Número de Pinos} = \text{Linhas} + \text{Colunas}$$

Imagine um teclado de computador, com aproximadamente 100 teclas. Seriam necessárias 100 GPIOs em um microcontrolador para tratar todas as teclas e enviar o código correspondente à placa-mãe do computador. Se for usado um teclado matricial, podemos organizar estas 100 teclas eletricamente (a disposição mecânica pode ser diferente) em 10 linhas e 10 colunas, totalizando 20 pinos de GPIO.

Cada tecla é um "*push button*" que conecta eletricamente uma linha a uma coluna, sendo que cada combinação "linha-coluna" só pode ser conectada por uma tecla específica, permitindo assim sua identificação. As figuras abaixo mostram um teclado de membrana com suas conexões:



Estrutura interna



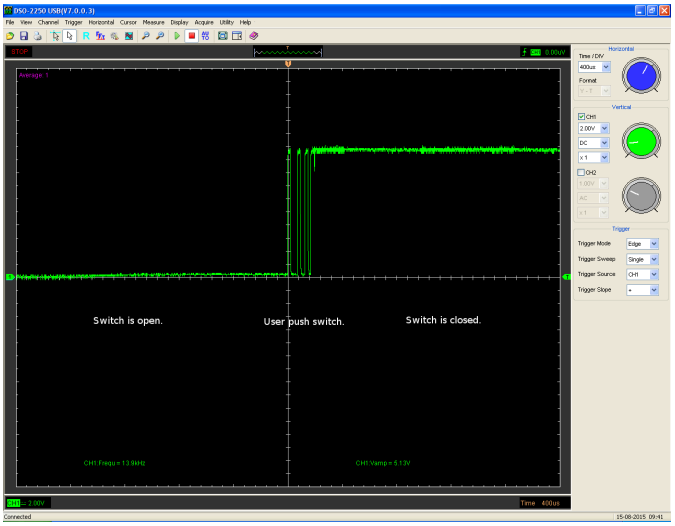
Se por um lado a estrutura matricial simplifica as conexões físicas, por outro exige um algoritmo mais elaborado para a leitura da tecla pressionada. O algoritmo usado é o de **varredura** (*scan*), e envolve ativar uma linha por vez, enviando um sinal lógico baixo (0) e verificando, por meio das colunas, se alguma tecla foi pressionada. Isso é feito repetidamente para todas as linhas, de forma cíclica. Pode-se realizar a varredura por *polling* ou precedendo a mesma por uma etapa de sinalização de "alguma" tecla pressionada, através de interrupção, que é a forma mais utilizada.

O processo de detecção de tecla e posterior varredura segue os seguintes passos:

1. Inicialmente, configure as linhas como saídas digitais **em estado lógico baixo** (0) e as colunas como entradas de interrupção externa sensíveis à **borda de descida**.
2. No momento que uma tecla é pressionada, a linha e a coluna correspondentes são conectadas. Como todas as linhas estão em nível baixo, a coluna correspondente à tecla vai do nível alto para o baixo (borda de descida), gerando a chamada à ISR. **As próximas etapas ocorrem dentro da ISR da coluna correspondente**.
3. Mantenha a primeira linha em nível baixo e coloque as demais em nível alto.
4. Leia o nível lógico na coluna correspondente à ISR. Se o valor lido é "0", isso indica que a tecla pressionada está na primeira linha, e assim a ISR deve guardar o valor correspondente à tecla que liga a linha 1 com a coluna correspondente à ISR e a ISR vai a seu final, limpando sua "flag".
5. Caso o valor lido na coluna seja "1", isso significa que a tecla pressionada está em outra linha; assim, coloque a linha 1 em nível alto e a linha 2 em nível baixo.
6. Leia novamente o nível lógico na coluna, verificando se a tecla está na linha 2.
7. Repita o processo até localizar a tecla correspondente, ou até que todas as linhas sejam verificadas e nenhuma delas corresponda à tecla pressionada (isto pode acontecer, como será visto adiante).

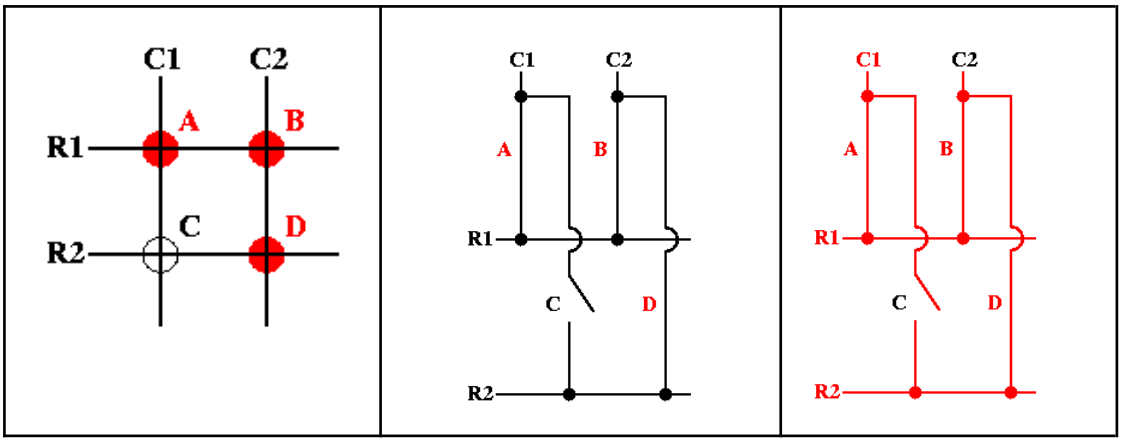
As **colunas** devem ter resistores "***pull-up***" conectados para garantir que, quando nenhuma tecla estiver pressionada, o sinal lido nas colunas seja logicamente alto (1). Isso garante que, ao pressionar uma tecla, a leitura de um estado baixo (0) seja detectada corretamente. No caso de microcontroladores que possuem *pull-ups* internos, esses podem ser habilitados para simplificar o circuito.

Embora os teclados matriciais sejam simples de implementar, dois fenômenos podem causar problemas na leitura de teclas: *bouncing* e *ghosting*. O ***bouncing*** ocorre quando uma tecla é pressionada ou liberada, mas o sinal gerado nos pinos de entrada não é limpo ou estável. Em vez de um único sinal de "pressionamento" ou "liberação" de tecla, o que ocorre é uma série de transições rápidas e erráticas entre nível alto ('1') e nível baixo ('0') antes de um estado estável ser alcançado, como ilustra na figura que se segue. Isso pode ser interpretado como vários pressionamentos de tecla, quando na realidade houve apenas uma única pressão..



(Fonte: All About Circuits).

O ***ghosting*** é um fenômeno que ocorre quando o pressionamento de várias teclas ao mesmo tempo (principalmente em teclados matriciais) faz com que teclas não pressionadas sejam erroneamente registradas como pressionadas. Vimos que, em um teclado matricial, as teclas são organizadas em linhas e colunas. Quando se pressiona uma tecla, o circuito entre uma linha e uma coluna é fechado. Contudo, ao pressionar múltiplas teclas simultaneamente (como A, B e D na figura, onde duas teclas estão na linha R1 e outra na linha R2), as linhas e colunas podem se combinar de maneira incorreta, resultando em uma "tecla fantasma" (em inglês, *ghost key*). Nesse cenário, o sistema interpreta erroneamente que uma tecla adicional, como C, foi pressionada. Isso ocorre porque, ao colocar a linha R2 em nível baixo para verificar as teclas pressionadas nessa linha, a coluna C1 também é levada ao nível baixo devido às conexões das teclas A e B em R1 e B e D em C2, levando à detecção incorreta da tecla 'C' como pressionada. Além disso, o *ghosting* pode provocar curtos-circuitos entre saídas digitais em determinadas combinações de teclas, resultando em danos físicos aos componentes do sistema.



(Fonte: Dribin).

Para evitar danos aos pinos das linhas em *ghosting*, as **linhas** de saída devem ser configuradas como saídas "***open drain***". Isso significa que as linhas só podem levar ativamente o sinal para o nível lógico baixo (0), enquanto o estado alto (1) é garantido pelos resistores *pull-up* nas colunas. Imagine que o usuário pressione duas teclas da mesma coluna nas linhas 1 e 2. A ISR correspondente à coluna será iniciada, e R1 (L1) permanecerá em nível lógico baixo enquanto R2 (L2) muda para nível alto. Se as duas teclas estão pressionadas, ambas as linhas estão conectadas à mesma coluna, e, portanto, conectadas entre si. Teremos um curto-circuito entre uma saída em nível baixo e uma em nível alto, o que pode provocar danos. Se as saídas são *open drain*, não há nível lógico alto "ativo", e assim a coluna permanece em nível baixo. Por outro lado, a estratégia proposta para identificação da tecla pressionada apenas reconhece a primeira tecla pressionada na varredura, enquanto outras teclas pressionadas simultaneamente são ignoradas. Este efeito é chamado de ***key blocking*** ou *key rollover*.

Uma solução simples para lidar com *bouncing* é implementar um ***debounce***, em *software* que consiste em adicionar um pequeno *delay* após a detecção de uma pressão de tecla, permitindo que o *bounce* termine antes de o microcontrolador registrar a tecla. Por exemplo, após detectar a pressão de uma tecla, o sistema espera cerca de 20 ms antes de verificar novamente se a tecla ainda está pressionada, garantindo uma leitura estável:

```
if (coluna == 0) {
   delay(20); // Aguardar o fim do bouncing
   if (coluna == 0) { // Verificar novamente a tecla
      // Registrar a tecla pressionada
   }
}
```

Essa técnica é eficaz para sistemas mais simples, embora existam outras abordagens mais avançadas, como filtros de *hardware*, componentes específicos ou algoritmos de *software* mais sofisticados.

Embora os *keypads* de membrana utilizam uma tecnologia baseada em camadas de material condutor (geralmente carbono ou outro material metálico) e não têm os contatos mecânicos típicos dos botões físicos, a natureza elétrica e a mecanização de pressão de uma membrana pode criar flutuações elétricas rápidas quando a chave é acionada, algo que se comporta de forma semelhante ao *bouncing* de uma chave de contato mecânica. Esses *bouncings* tendem a ser menos pronunciado do que em botões mecânicos tradicionais. Isso não significa que o *bouncing* não seja uma preocupação em *keypads* de membrana. Em sistemas sensíveis, como sistemas embarcados, onde a precisão das leituras é crítica, *bouncing* pode levar a leituras incorretas, fazendo com que o sistema registre uma tecla pressionada várias vezes ou não reconheça uma pressão de tecla.

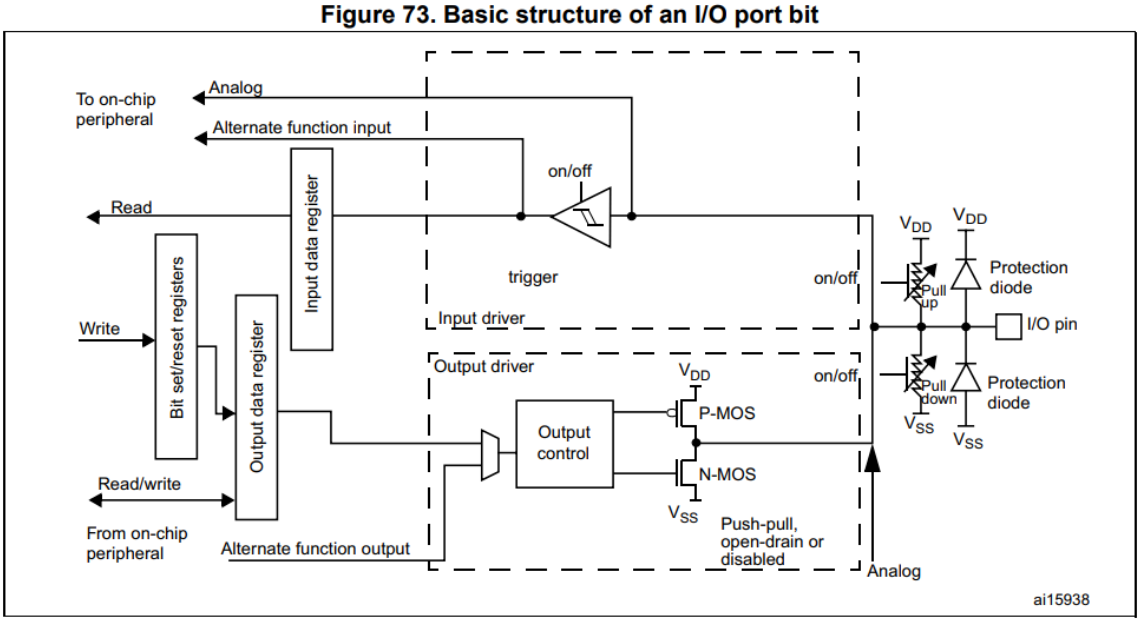
Ambos os fenômenos, ***bouncing*** e ghosting, podem comprometer a funcionalidade de um teclado, especialmente quando um sistema depende da entrada precisa de dados. O *bouncing* causa falsos registros de pressionamento, enquanto o *ghosting* gera entradas erradas quando múltiplas teclas são pressionadas simultaneamente. Portanto, para garantir um funcionamento correto e confiável, é importante que tanto o *hardware* quanto o *software* do sistema de controle do teclado lidem adequadamente com esses problemas.

Por fim, é importante ressaltar que a leitura do teclado matricial **NÃO É** uma aplicação de porta paralela, pois a varredura analisa uma linha de cada vez. A varredura funciona porque a velocidade com que a varredura é executada é muito maior do que a velocidade com que apontamos a tecla. Assim, a tecla permanece pressionada pelo dedo do usuário durante todo o tempo de varredura. Apesar da varredura ser sequencial, o teclado matricial é classificado como uma interface paralela devido ao fato de que os pinos de linha e coluna são lidos de forma independente e simultânea no nível de *hardware*, o que caracteriza a comunicação paralela.

STM32H7A3: MÓDULO GPIO

Os módulos GPIO integrados no STM32H7A3 permitem configurar os pinos em diversos modos para atender a diferentes funcionalidades. A figura apresenta uma estrutura básica, dos pinos de entrada e saída do microcontrolador.

Figure 73. Basic structure of an I/O port bit



Entre os modos de operação, temos a entrada, que pode ser configurada como ***Input Floating***, onde o pino é uma entrada flutuante ou uma entrada que não está conectada a um nível lógico definido (alto ou baixo); ***Input Pull-Down***, com um resistor de *pull-down* (figura à esquerda); ***Input Pull-Up***, que utiliza um resistor de *pull-up* (figura à direita); e ***Analog***, que permite uma entrada analógica. Para a saída, as opções incluem ***Output Open-Drain***, que

é configurada como uma saída de dreno aberto e permite o uso de um resistor de *pull-up* externo, e ***Output Push-Pull***, que fornece uma saída de alta corrente. Além disso, os pinos também podem ser configurados para funções alternativas, como ***Alternate Function Push-Pull***, que atua como uma saída *push-pull* para um periférico, e ***Alternate Function Open-Drain***, que opera como uma saída de dreno aberto para funções alternativas. Todos os pinos dedignados para módulos GPIO possuem diodos de proteção, fornecendo um caminho para descargas elétricas.

O modo de operação de um pino de STM32H7A3 é configurado através do registrador `GPIOx_MODER`, e cada pino possui um conjunto de registros de controle associados que permitem configurar diversos aspectos. Nos roteiros anteriores, vimos que o `GPIOx_MODER` define o modo do pino, enquanto o `GPIOx_OTYPER` especifica o tipo de saída (*push-pull* ou *open-drain*), o `GPIOx_OSPEEDR` define a velocidade de saída, e o `GPIOx_PUPDR` ajusta o resistor de *pull-up* ou *pull-down*, independentemente da direção do pino. Para a leitura dos dados, o `GPIOx_IDR` é um registrador de leitura-somente que contém os dados de entrada, enquanto o `GPIOx_ODR` é um registrador de leitura-escrita que contém os dados de saída. O `GPIOx_BSRR` permite definir e *resetar bits* individualmente no registrador `GPIOx_ODR`, e o `GPIOx_LCKR` bloqueia a configuração do pino para evitar alterações até o próximo *reset* do MCU ou do periférico. Por fim, os registradores `GPIOx_AFRL` e `GPIOx_AFRH` são utilizados para selecionar a função alternativa a ser aplicada ao pino. Além das funções principais, os pinos GPIO têm a capacidade de gerar interrupções externas e contar com mecanismos de bloqueio.

Vale destacar aqui uma descrição sumária desses registradores no Manual de Referência, em que esses registradores são classificados em registradores de configuração/seleção de função/bloqueio de acesso aos *bits* (`GPIOx_MODER`, `GPIOx_OTYPER`, `GPIOx_OSPEEDR`, `GPIOx_PUPDR`, `GPIOx_AFRH`, `GPIOx_AFRL` e `GPIOx_LCKR`), registradores de dados (`GPIOx_IDR` e `GPIOx_ODR`) e um registrador de "*set*"/"*reset*" (`GPIOx_BSRR`).

11.1 Introduction

Each general-purpose I/O port has four 32-bit configuration registers (GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR and GPIOx_PUPDR), two 32-bit data registers (GPIOx_IDR and GPIOx_ODR) and a 32-bit set/reset register (GPIOx_BSRR). In addition all GPIOs have a 32-bit locking register (GPIOx_LCKR) and two 32-bit alternate function selection registers (GPIOx_AFRH and GPIOx_AFRL).

Ao examinarmos a descrição de cada um desses registradores, podemos observar que todos eles permitem acesso tanto para leitura quanto para escrita, exceto os registradores `GPIOx_BSRR`, que são exclusivos para acesso de escrita. Como ilustrado no excerto do Manual de Referência a seguir, esses registradores servem apenas para setar (`BSn`) e resetar (`BRn`) os valores dos *bits* `n` dos registradores de dados de saída `GPIOx_ODR`. Denominamos esses registradores de **registradores de controle**, pois eles possibilitam a manipulação dos *bits* em um registrador de dados ou de configuração através dos sinais de controle "1", sem a

a necessidade de ler e escrever dados dos registradores manipulados. Isso é especialmente útil para operações que requerem mudanças rápidas e frequentes, como setar ou resetar saídas digitais. Em muitos casos, como neste, a leitura de GPIOx_BSRR retornará o valor 0x0000.

11.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A to K)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BR[15:0]:** Port x reset I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BS[15:0]:** Port x set I/O pin y (y = 15 to 0)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

Esses registradores de controle são comuns em microcontroladores, sendo utilizados em uma ampla gama de aplicações. Eles não são limitados a casos de uso específicos; pelo contrário, são uma parte fundamental do *design* de sistemas embarcados e contribuem para a eficiência e a funcionalidade dos microcontroladores.

De acordo com *Datasheet*, os pinos GPIO são projetados para serem de alta capacidade de corrente, permitindo que possam fornecer ou absorver quantidades significativas de energia em aplicações que exigem o controle de cargas externas, como LEDs, relés e outros dispositivos que demandam maior corrente. Além disso, a capacidade de seleção de velocidade oferecida pelo registrador `GPIOx_OSPEEDR` é especialmente importante para gerenciar o ruído interno, pois sinais que operam a altas velocidades podem gerar interferências eletromagnéticas e ruído indesejado, afetando o desempenho do sistema como um todo. Ao otimizar a velocidade dos GPIOs, é possível minimizar essas emissões, resultando em um funcionamento mais estável e eficiente. Essa gestão adequada da velocidade também contribui para a redução do consumo de energia, uma vez que modos de operação mais lentos podem consumir menos corrente, prolongando a vida útil de dispositivos alimentados por bateria.

PROGRAMAÇÃO EM C: COMPILAÇÃO CONDICIONAL

Imagine que você está desenvolvendo um firmware para um dispositivo embarcado, mas esse dispositivo pode rodar em diferentes microcontroladores, com variações de memória, periféricos e até sistemas operacionais. Ao invés de criar versões separadas para cada uma dessas configurações, você pode usar a compilação condicional para customizar o comportamento do programa para cada cenário, simplesmente alterando as definições de macros no momento da compilação. Isso não só economiza tempo de desenvolvimento, como também melhora a manutenibilidade e a escabilidade do projeto.

A **compilação condicional** é um recurso presente em muitas linguagens de programação, incluindo C, que permite incluir ou excluir partes do código **durante o processo de compilação** dependendo de determinadas condições, como a plataforma de *hardware*, o tipo de compilador ou até mesmo configurações do projeto. Em vez de escrever várias versões do código, podemos usar diretivas de pré-processador para criar versões alternativas de código em um único arquivo, facilitando a manutenção e a personalização do programa.

As diretivas de pré-processador são instruções que começam com o caractere '`#`'. Elas são processadas antes da compilação real do código e permitem incluir ou excluir trechos de código com base em condições definidas pelo programador. A estrutura condicional básica envolve a diretiva "`#if`", que avalia uma expressão condicional (`condição`) e, com base no resultado, inclua ou exclua trechos de código, e as diretivas `#elif` e `#else` que permitem adicionar alternativas à condicional:

```
#if condição
      // Código a ser incluído se a condição for verdadeira
#elif outra_condição
      // Código a ser incluído se a outra_condição for verdadeira
#else
      // Código a ser incluído se nenhuma das condições anteriores
for verdadeira
#endif
```

As diretivas "`#if`" e "`#elif`" avaliam expressões condicionais construídas com operadores aritméticos, relacionais ou lógicos, envolvendo macros ou verificam a definição de macros através da diretiva "`defined`".

Antes de usar a compilação condicional, é necessário definir as macros que servirão como indicadores. A diretiva "`#define`" define uma macro, que pode ser uma constante ou uma *flag*, enquanto a diretiva "`#undef`" remove uma macro previamente definida. Essas macros podem ser usadas nas condições do "`#if`" e "`#elif`".

```
#define ARM 1
#if ARM == 1
      // Código específico para a arquitetura ARM
```

```
#elif ARM == 2
      // Código específico para a arquitetura ARM Thumb
#else
      // Código para outras arquiteturas
#endif
```

No lugar de "`#if defined(NOME_MACRO)`" e "`#if !defined (NOME_MACRO)`" podemos usar as diretivas "`#ifdef`" e "`#ifndef`", respectivamente, para verificar se NOME_MACRO foi definida ou não.