Capítulo 6

(GPIO) Portas de entrada e saída no STM32F407

6.1. Introdução

Embora o fato de os microcontroladores serem produtos de alta tecnologia, eles acabam ficando sem uso prático se não puderem ser conectados a outros dispositivos. O aparecimento de tensão nos pinos de um microcontrolador não é útil se não for usado para ligar/desligar dispositivos externos como *LEDs*, relés e *displays*.

Um pino do microcontrolador que pode ser configurado pelo software em tempo de execução para executar várias funções é chamado de pino de entrada/saída de propósito geral (*GPIO – General Purpose Input/Output*). O GPIO fornece alta flexibilidade de uso no projeto de sistemas embarcados. Ele permite a um microcontrolador se adequar às necessidades de uma grande quantidade de sistemas.

O software pode programar um pino no STM32F407 para desempenhar uma das seguintes funções:

1. Entrada digital, que pode detectar quando uma tensão no pino está no nível lógico alto ou baixo;
2. Saída digital, que controla a tensão de saída no pino, enviando um nível lógico alto ou baixo;
3. Função analógica, que pode executar uma conversão analógica para digital ou digital para analógica;
4. Outras funções complexas como saídas de sinal PWM, driver para LCD, entrada de captura de eventos, interrupções externas, interfaces de comunicação como USART, SPI, I2C, USB, Ethernet, etc. Chamamos essa última categoria de função alternativa (*Alternative Function – AF*).

Uma das características mais importantes dos microcontroladores é o número de pinos de propósito geral que permitem ao projetista conectar dispositivos sensores ou atuadores. O STM32F407 possui 80 pinos de *I/O* que podem ser individualmente configurados para funcionarem como entradas e/ou saídas. Os pinos de *I/O* são agrupados em estruturas físicas conhecidas como "portas", que compartilham os mesmos registradores de controle e de dados.

O STM32F407 possui 5 portas GPIO denominadas GPIO PORT A (GPIOA), GPIOB, GPIOC, GPIOD e GPIOE. Cada porta possui 16 pinos de *I/O*, nomeados de Px0 (menos significativo) a Px15 (mais significativo), onde x é a letra correspondente à porta. Assim, quando nos referimos ao pino PC4, por exemplo, estamos nos referindo ao pino 4 da porta C (atente para o fato de que a indexação dos pinos começa em zero).

Por razões práticas, além da maioria dos pinos de *I/O* poderem ser usados como entradas/saídas de uso geral, eles compartilham de várias funções especiais, que serão exploradas em capítulos posteriores. Se um pino é usado para qualquer uma das funções, ele não pode ser usado para outra. Na verdade, a função de um pino de *I/O* pode ser alterada enquanto o microcontrolador estiver em funcionamento (*runtime*), mas o hardware externo ao microcontrolador deve estar preparado para se adequar à mudança para nova funcionalidade.

Apesar do STM32F407 ser alimentado tipicamente com uma tensão de 3,3V, quando os seus pinos estão operando como entradas digitais eles suportam uma faixa de tensão que vai de 0 V a 5 V. Dizemos que os pinos digitais são tolerantes à tensão de 5V, mas o mesmo não ocorre quando os pinos estão sendo usadas como entrada analógica, cujo valor máximo é de 3,3V. Deve-se assegurar que os valores de nível lógico alto e baixo sejam obedecidos. Assim, para o nível lógico baixo, um valor tão próximo de 0 V quanto possível deve ser aplicado. Já para o nível alto, um valor tão próximo a, no mínimo, 3,3V deve ser aplicado. Sempre deve ser evitada a aplicação de valores intermediários de tensão em pinos operando como entradas digitais, uma vez que os valores podem não ser distinguidos pelo hardware do microcontrolador e causar mau funcionamento da aplicação.

As principais características das portas de GPIO são:

• Cada porta tem até 16 pinos sob controle simultâneo;
• Modos de entrada: flutuante ou com resistores de *pull-up/down*;
• Modos de saída possíveis: *push-pull* ou dreno aberto, com ou sem resistores de *pull-up/down*;
• Seleção de velocidade para cada pino para minimizar a geração de ruídos e emissões eletromagnéticas;
• Funções analógicas (entrada e saída);
• Multiplexação flexível que permite o uso de pinos como GPIO ou como uma das várias funções periféricas alternativas.

Normalmente, um periférico, como o GPIO, requer um processo de inicialização antes de poder ser usado. Isso pode incluir algumas das seguintes etapas:

• Programar o circuito de controle de clock para habilitar a conexão do sinal de clock ao periférico. Muitos microcontroladores modernos, como é o caso do STM32, permitem o ajuste fino da distribuição do sinal de clock, como habilitar/desabilitar a conexão do clock para cada periférico individual para otimizar o uso de energia. Normalmente, os sinais de clock dos periféricos são desligados por padrão e é necessário habilitá-los antes de programar o periférico.
• Configurar o modo de operação dos pinos de I/O para corresponder ao uso (por exemplo, direção de entrada/saída, função, etc.). Além disso, você também pode precisar programar configurações adicionais para definir as características elétricas esperadas, como tipo de saída, nível de tensão.
• Configuração periférica. A maioria dos periféricos contém vários registradores programáveis que precisam ser configurados antes de poder usá-los. Em alguns casos, a sequência de programação pode parecer um pouco mais complexa do que a de um microcontrolador de 8 bits porque os periféricos em microcontroladores de 32 bits geralmente são muito mais sofisticados do que os periféricos em sistemas de 8/16 bits.
• Configuração de interrupção. Se um periférico for usado com operações de interrupção, você precisará programar o controlador de interrupção (NVIC) para habilitar a interrupção e configurar o seu nível de prioridade.

A configuração de um pino é feita por meio dos registradores de controle da porta GPIO a qual o pino pertence. Cada porta GPIO possui quatro registradores de controle de 32 bits que permitem configurar o modo de operação de até 16 pinos simultaneamente:

• MODER (*Mode Register*)
• OTYPER (*Output Type Register*)
• OSPEEDR (*Output Speed Register*)
• PUPDR (*Pull-Up/Pull-Down Register*)

Esses, e todos os registradores de controle de todos os periféricos, são acessados pelo processador como se fossem posições de memória (*I/O* mapeado em memória). O processador acessa os registradores de todas as portas GPIO pode meio do barramento de alta velocidade AHB1.

Na Figura 1 é mostrada a estrutura básica do hardware de cada pino de I/O.

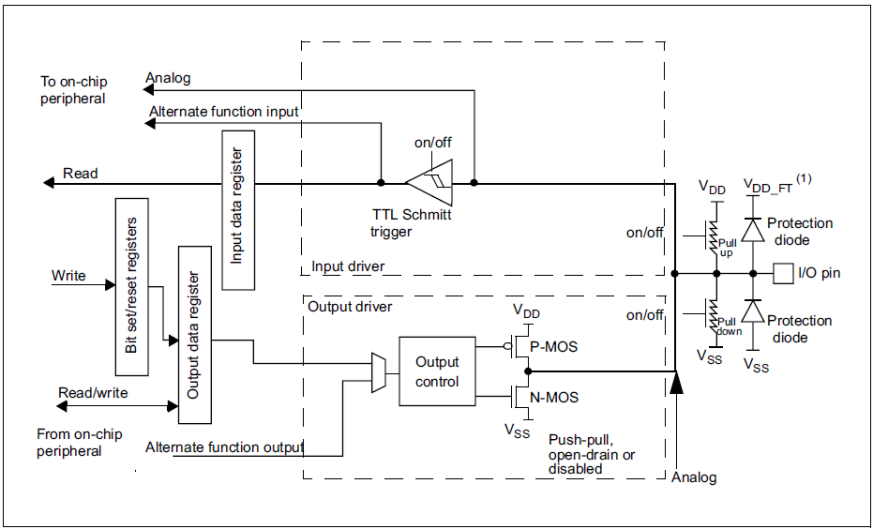


Figura 1 – Estrutura básica do hardware de um pino de uma porta de I/O tolerante a 5 V.

O registrador MODER, cuja organização é mostrada na Figura 2, é usado para selecionar o modo de operação dos pinos de I/O (entrada digital, saída digital, função alternativa ou função analógica) de uma porta. Dependendo do modo selecionado, algumas partes da estrutura mostrada na Figura 1 são habilitadas enquanto outras não, conforme veremos mais adiante.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 2 – Registrador MODER.

Para definir o modo de operação de um determinado pino *y* (*y* = 0, 1, ..., 15) de um GPIO, devemos configurar 2 bits contíguos (bits *2y:2y+1*), chamados de MODERy[1:0] no registrador MODER. Os valores binários possíveis de configuração são os seguintes:

0b00: Entrada digital (modo default)
0b01: Saída digital
0b10: Função alternativa
0b11: Função analógica

Por exemplo, suponha que desejamos configurar o pino 0 (*y=0*) como uma saída digital. Nesse caso, devemos escrever o valor 0b01 nos bits *2y:2y+1*, isto é, nos bits 0 e 1, que são os bits MODER0[1:0] no registrador MODER. Outro exemplo, agora queremos configurar o pino 5 (*y=5*) no modo analógico. Nesse caso, devemos escrever o valor 0b11 nos bits 10 e 11, que são os bits MODER5[1:0] do registrador MODER.

Toda inicialização de periférico, como as portas de GPIO, é realizada por meio da programação dos registradores. Conforme mencionado, os registradores são mapeados na memória e, portanto, podem ser acessados por meio de ponteiros, na linguagem C. Por exemplo, para implementar o modo analógico no pino 5 de uma porta, poderíamos escrever algo como:

```
MODER |= 0b11 << 10;       //seleciona o modo analógico no pino 5
```

Com o uso do operador bit a bit OR (|) na linha de programa mostrada acima, estamos, por meio da técnica de mascaramento de bits, configurando o pino 5 sem alterar a configuração dos demais pinos da porta. O mascaramento com lógica OR permite setar bits em um registrador, enquanto o mascaramento com lógica AND permite resetar bits em um registrado.

Os registradores OTYPER e OSPEEDR são usados para selecionar o tipo de saída (*push-pull* ou *open-drain*) e a velocidade (*low speed*, *medium speed*, *high speed* e *very high speed*), respectivamente. É possível usar o driver de saída no modo *push-pull* ou no modo dreno aberto (somente o transistor N-MOS é ativado quando um 0 é escrito).

Todos os pinos do GPIO possuem resistores internos de *pull-up* e *pull-down* que podem ser ativados ou desativados por meio do registrador PUPDR.

6.2. GPIO no modo de entrada (modo default)

Durante e logo após o reset, a maioria dos pinos das portas de I/O iniciam a operação no modo de entrada flutuante.

Excepcionalmente, os pinos de depuração (PA15: JTDI, PA14: JTCK/SWCLK, PA13: JTMS/SWDAT, PB4: NJTRST e PB3: JTDO), iniciam nas suas respectivas funções alternativas para permitir o debug do sistema.

Quando o pino de I/O é programado como entrada:

- o driver de saída é desativado
- o Schmitt trigger é ativado

Na Figura 3 é mostrado como fica configurado o hardware do pino da porta de I/O no modo de entrada.

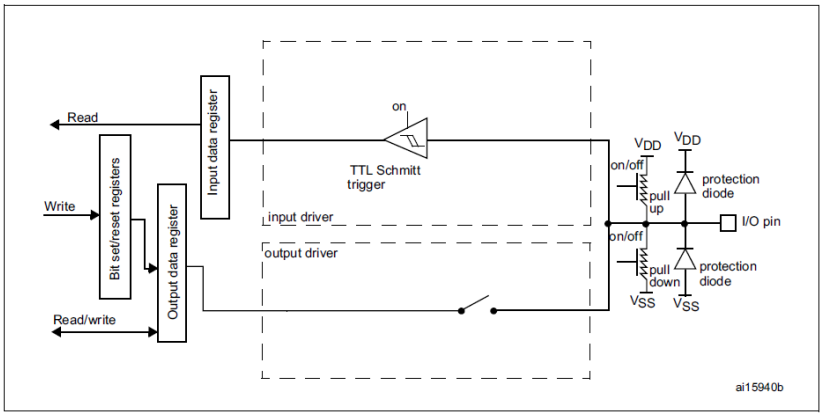


Figura 3 – Configuração de entrada do hardware do pino da porta de I/O.

O bloco Schmitt trigger, também conhecido como disparador de Schmitt, é um circuito comparador. Quando o nível de tensão de entrada no pino é maior que um limiar (pré-estabelecido pelo hardware), a saída do comparador está em nível alto. Quando a entrada está abaixo de outro limiar, a saída está em nível baixo. Quando a entrada se encontra entre os dois limiares, a saída retém o valor anterior até a entrada se alterar suficientemente para mudar o estado do disparador. A ação conjunta dos dois limiares é chamada de histerese. O benefício de um disparador Schmitt sobre um circuito com somente um ponto limiar de entrada é uma maior estabilidade (imunidade ao ruído). Com somente um ponto de limiar de entrada, um sinal ruidoso operando próximo a esse ponto, poderia fazer com que a saída ficasse comutando rapidamente, acima e abaixo do ruído. Um sinal de entrada ruidoso no disparador de Schmitt perto de um ponto limiar poderia causar somente uma mudança no valor de saída, depois do qual teria que ultrapassar o outro limiar para causar uma nova mudança na saída.

Os dados presentes no pino são, então, condicionados pelo disparador de Schmitt e são amostrados pelo registrador de dados de entrada (*Input Data Register* – IDR) a cada ciclo de clock de AHB1. Um acesso de leitura ao registrador IDR fornece o estado de todos os pinos de I/O de uma determinada porta nos seus respectivos bits. A organização do registrador IDR é mostrada na Figura 4. Os 16 bits menos significativos refletem o estado de cada pino da porta. Os outros 16 bits são reservados e são lidos sempre como zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Figura 4 – Registrador de dados de entrada - IDR.

Por exemplo, a seguinte linha de código retorna o estado lógico dos pinos de uma porta:

```
int16_t leitura = IDR;     //lê o estado dos pinos de uma porta
```

Todos os pinos de uma porta podem ser fontes de requisição de interrupção externa. Para usarmos um pino como fonte de interrupção externa, devemos configurar o pino no modo de entrada, além de implementar as configurações específicas da interrupção, o que será abordado em capítulos posteriores.

O registrador de dados de saída (*Output Data Register* – ODR) e o registrador de ativação e desativação de bits (*Bit Set/Reset Register* – BSRR) não tem efeito sobre os pinos quando são configurados no modo de entrada, uma vez que o driver de saída é desabilitado nesse modo.

Os resistores de *pull-up/down* podem ser habilitados ou desabilitados por meio do registrador PUPDR. A organização do registrador PUPDR é mostrada na Figura 5.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 5 – Registrador PUPDR.

Para definir o modo de operação dos resistores de *pull-up/down* de um determinado pino *y* de um GPIO (*y* = 0, 1, ..., 15), devemos configurar 2 bits contíguos (bits *2y:2y+1*), chamados de PUPDRy[1:0] no registrador PUPDR. Os valores binários possíveis de configuração são os seguintes:

0b00: Sem resistor de *pull-up/down*
0b01: Habilita o resistor de *pull-up*
0b10: Habilita o resistor de *pull-down*
0b11: Reservado (sem função)

Por exemplo, suponha que desejamos ligar o resistor de *pull-up* no pino 0 (*y=0*), sem alterar a configuração dos demais pinos. Nesse caso, devemos escrever o valor 0b01 nos bits *2y:2y+1*, isto é, nos bits 0 e 1, que são os bits PUPDR0[1:0] no registrador PUPDR, conforme mostrado no código exemplo abaixo:

```
PUPDR |= 0b01; //habilita pull-up no pino 0 sem alterar o estado dos outros pinos
```

Outro exemplo, agora queremos ligar o resistor de *pull-down* no pino 5 (*y=5*). Nesse caso, devemos escrever o valor 0b10 nos bits 10 e 11, que são os bits PUPDR5[1:0] do registrador PUPDR, como mostrado no exemplo abaixo:

```
PUPDR |= (0b10 << 10); //habilita pull-down no pino 5 sem alterar o estado dos demais pinos
```

6.3. GPIO no modo de saída

Quando o pino de I/O é programado como saída, a configuração de hardware do pino fica como mostrada na Figura 6:

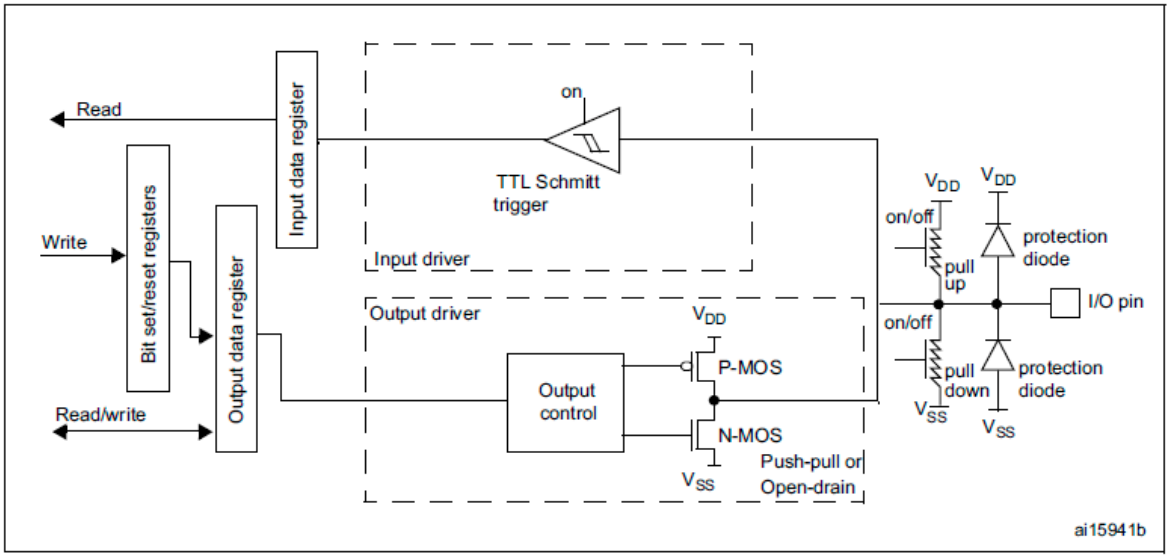


Figura 6 – Configuração de saída do hardware do pino da porta de I/O.

Nesse modo:

- o driver de saída é ativado:
  - No modo *push-pull*, um "0" no registrador de dados de saída (ODR) ativa o transistor N-MOS levando o pino para nível lógico baixo, enquanto um "1" ativa o transistor P-MOS, levando o pino para nível lógico alto;
  - No modo de dreno aberto (open drain), um "0" no registrador de dados de saída (ODR) ativa o transistor N-MOS enquanto um "1" deixa a porta em alta impedância (Hi-Z) (o transistor P-MOS nunca é ativado).
- a entrada Schmitt trigger também é ativada
- os resistores de *pull-up/down* são ativados ou não, dependendo do valor no registrador PUPDR

Quando um pino é configurado como saída, o valor armazenado no bit correspondente do registrador de dados de saída ODR, mostrado na Figura 7, é escrito no pino de I/O. Apenas os 16 bits menos significativos, referentes aos 16 pinos de uma porta, são utilizados. Os demais bits não têm efeito sobre os pinos. Um acesso de leitura ao registrador de dados de saída obtém o último valor escrito nos pinos.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 7 – Registrador de dados de saída - ODR.

A alteração do estado de saída de um pino de uma porta, sem alterar o estado dos demais pinos dessa mesma porta, é uma ação normalmente executada em 2 passos: Primeiro, deve-se ler o estado de todos os pinos por meio de uma leitura a ODR; Segundo, é feito uma operação de mascaramento (com logica AND, se desejarmos resetar um bit, ou com lógica OR, se desejarmos setar um bit) com o valor lido em ODR e o resultado é reescrito no registrador de dados de saída, conforme exemplo mostrado nas linhas de código a seguir:

```
ODR |= 1;     //seta o pino 0 da porta, mantendo o estado dos demais pinos
ODR &= ~1;    //reseta o pino 0 da porta, mantendo o estado dos demais pinos
```

Cada linha de código acima é traduzida em duas instruções: A primeira faz a leitura do registrador ODR; A segunda implementa o mascaramento do conteúdo de ODR com a constante declarada no código e carrega o valor resultante de volta em ODR.

O registrador de ativação e desativação de bits (*Bit Set/Reset Register* – BSRR), mostrado na Figura 8, é um registrador que permite realizar a operação de set e reset em cada bit individual no registrador de dados de saída. Diferentemente de ODR, o registrador BSRR utiliza todos os 32 bits. Cada bit em ODR corresponde a dois bits de controle em BSRR: BSRR(*i*), BS*i* e BSRR(*i+16*), BR*i*. Quando escrito 1, o bit BS(*i*) seta o bit ODR(*i*) correspondente. Quando escrito em 1, o bit BR(*i*) reseta o bit correspondente do ODR(*i*). Escrever 0 em qualquer bit de BSRR não afeta o bit correspondente em ODR. Se houver uma tentativa de setar e resetar simultaneamente um bit no BSRR, a ação de setar terá prioridade.

O uso do registrador BSRR para alterar os valores de bits individuais no registrador ODR é um efeito que ocorre em apenas um passo. O registrador BSRR fornece uma maneira alternativa de executar a manipulação atômica bit a bit, onde é possível modificar um ou mais bits em um único acesso.

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

Figura 8 – Registrador BSRR.

O registrador BSRR é acessado por meio de dois registradores de 16 bits: BSRRL, que agrupa os bits BS[15:0], e BSRRH, que agrupa os bits BR[15:0]. No exemplo mostrado nas linhas de código abaixo, o pino 0 de uma porta é setado enquanto o pino 1 da mesma porta é resetado, mantendo o estado dos demais pinos:

```
BSRRL = 1;          //seta o pino 0
BSRRH = 1 << 1;     //reseta o pino 1
```

Existem dois tipos de saídas digitais: *push-pull* e dreno aberto (*open drain*). Um importante uso de saídas com dreno aberto é conectar diretamente várias saídas juntas a uma linha de modo que qualquer saída possa levar essa linha para nível lógico baixo sem produzir um curto-circuito nas demais saídas. Por exemplo, dispositivos em sistemas de comunicação que operam no mesmo barramento, como o barramento I2C, fazem uso de saídas em dreno aberto.

Comparado ao tipo de saída com dreno aberto, a saída *push-pull* tem a vantagem de ser mais rápida, podendo modificar a tensão de saída do pino de maneira mais rápida se o circuito externo apresentar alguma capacitância. Outra vantagem é que esse tipo de saída pode fornecer corrente e simplificar o circuito. Por exemplo, uma saída em *push-pull* pode alimentar diretamente um LED externo enquanto uma saída com dreno aberto não pode fazê-lo sem uma fonte externa. Usualmente, entretanto, saídas em *push-pull* não podem ser interligadas diretamente pois podem causar potenciais curtos-circuitos.

A seleção do tipo de saída, *push-pull* ou *open-drain,* é feita nos 16 bits menos significativos do registrador OTYPER, mostrado na figura 9. Escrever 0 em um bit *i* faz o pino *i* da porta correspondente operar no modo *push-pull* (modo padrão), enquanto escrever 1 no bit *i* faz o pino *i* operar no modo *open-drain*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 9 – Registrador OTYPER.

Apesar do pino poder ser configurado como saída, o nível lógico escrito no pino de I/O é amostrado no registrador de dados de entrada IDR a cada ciclo de clock de AHB1. Um acesso de leitura ao registrador de dados de entrada obtém o estado do pino de I/O.

Uma importante característica dos pinos de GPIO é a velocidade (razão) com a qual a sua tensão de saída pode variar por unidade de tempo. Essa propriedade é conhecida como *slew rate*, normalmente medida em V/µs. Por exemplo, se a tensão de saída de um pino varia de 0 a 3V em 3µs, temos que o *slew rate* é de:

$$slew\ rate = \Delta v/\Delta t = 3V/3\mu s = 1V/\mu s$$

Quanto maior o *slew rate*, maior é a velocidade na qual a tensão do pino pode variar. Sinais que variam rapidamente normalmente têm grandes amplitudes e apresentam componentes harmônicas de alta frequência que causam interferências eletromagnéticas por indução, radiação ou condução, podendo produzir mal funcionamento em circuitos próximos. Um *slew rate* menor é preferível para minimizar possíveis distúrbios eletromagnéticos.

O *slew rate* de um pino de I/O é programado por meio do registrador OSPEEDR, mostrado na Figura 10.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 10 – Registrador OSPEEDR.

Para definir o *slew rate* de um pino *y* de um GPIO (*y* = 0, 1, ..., 15), devemos configurar 2 bits contíguos (bits *2y:2y+1*), chamados de OSPEEDRy[1:0] no registrador OSPEEDR. Os valores binários possíveis de configuração são os seguintes:

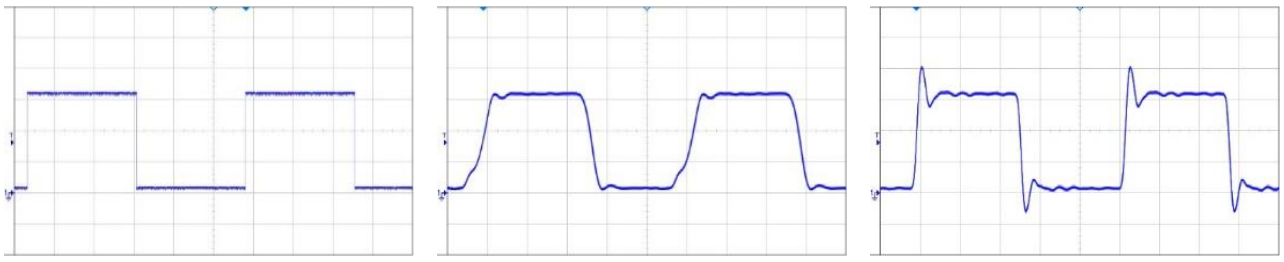
0b00: *Low speed*
0b01: *Medium speed*
0b10: *High speed*
0b11: *Very high speed*

Por exemplo, suponha que desejamos configurar o *slew rate* do pino 5 (*y=5*) como Very *High speed,* sem alterar o *slew rate* dos demais pinos. Nesse caso, devemos escrever o valor 0b11 nos bits *2y:2y+1*, isto é, nos bits 10 e 11, que são os bits OSPEEDR5[1:0] no registrador OSPEEDR, conforme mostrado na linha de código abaixo:

```
OSPEEDR |= 0b11 << 10; //High speed no pino 5 sem alterar o slew rate dos outros pinos
```

Nas figuras a seguir, podemos observar o efeito do *slew rate* sobre um pino de I/O no microcontrolador STM32F407.

Na Figura 11(a), é observada a forma de onda ideal do sinal de saída em um pino que fica alternando periodicamente o seu estado lógico: A transição do nível alto para baixo, ou vice-versa, deveria acontecer instantaneamente, como sugere a figura. Na prática, entretanto, a transição entre os níveis não é instantânea, mas demora um certo tempo (tempo de subida ou tempo de descida), que pode ser observado na Figura 11(b), onde o *slew rate* do pino foi configurado como *Low speed*. Os tempos de subida e descida podem ser reduzidos configurando o *slew rate* do pino como *Very high speed*, mostrado na Figura 11(c). Nesse caso, como consequência do alto valor do *slew rate*, a forma de onda apresenta sobretensões (*overshoots*) de alta frequência, o que pode produzir interferência eletromagnética em circuitos próximos.



(a) *(b)* *(c)*

Figura 11 – Efeito do slew rate na forma de onda do sinal de saída de um pino de I/O: (a) sinal ideal. (b) sinal com slew rate em Low Speed. (c) sinal com slew rate em Very High Speed.

6.4. GPIO no modo de função alternativa

Quando um pino de I/O é programado para operar no modo de função alternativa:

- o driver de saída pode ser configurado como dreno aberto ou *push-pull*
- o driver de saída é acionado pelo sinal proveniente do periférico, e não mais pelo registrador ODR ou BSRR
- a entrada Schmitt trigger é ativada e o seu sinal de saída é roteado para a entrada de algum periférico
- os dados presentes no pino de I/O são amostrados no registrador IDR a cada ciclo de clock de AHB1

Na Figura 12, é mostrada a configuração de hardware do pino no modo de função alternativa:

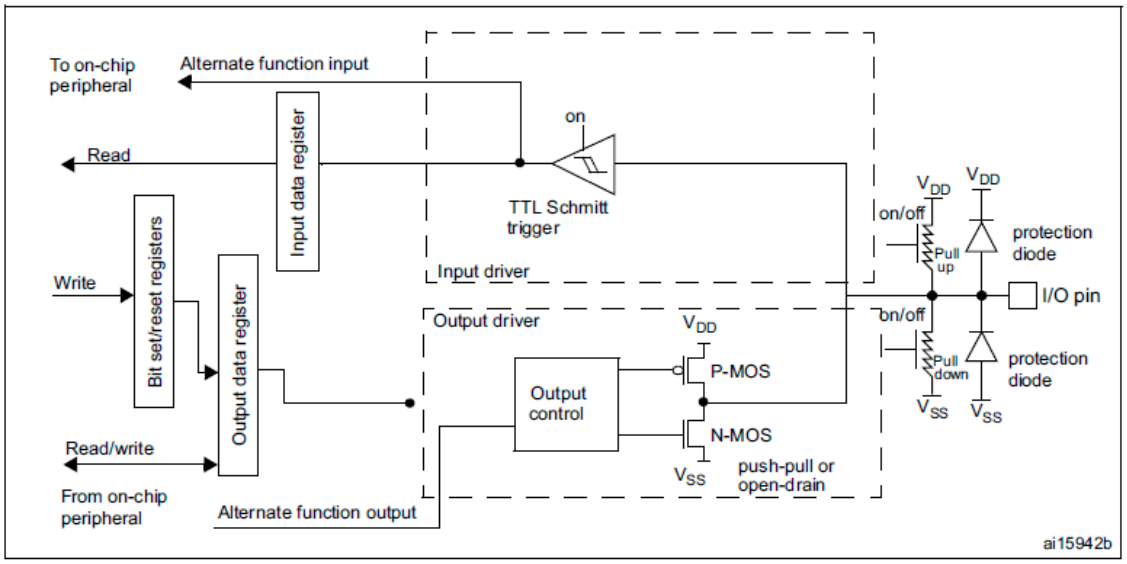


Figura 12 – Configuração de função alternativa do hardware do bit da porta de I/O.

Cada pino de I/O do STM32F407 pode, alternativamente, se conectar a até 16 periféricos/módulos internos de entrada e/ou saída por meio de um multiplexador que permite que o pino desempenhe apenas uma função alternativa (AF) por vez. Dessa maneira, não pode haver conflito entre periféricos compartilhando o mesmo pino de I/O.

Cada pino possui um multiplexador com dezesseis entradas de funções alternativas (Figura 13), nomeadas de AF0 até AF15, onde a função alternativa AF0 é o *default* após o reset. A função alternativa desejada para uma determinada aplicação pode ser configurada por meio dos registradores de função alternativa *Alternate Function Register Low – AFRL,* para os pinos de 0 a 7 de uma porta, e *Alternate Function Register High – AFRH,* para os pinos de 8 a 15.

Além dessa arquitetura flexível de multiplexação, cada periférico é mapeado em diferentes pinos de I/O para otimizar o número de periféricos disponíveis em uma aplicação.

Para configurar um pino com uma função diferente daquela de entrada e saída digital regular, devemos proceder da seguinte forma:

- Para os conversores AD e DA, configuramos o pino desejado no modo analógico no registrador MODER;

• Para os demais periféricos, configuramos o pino desejado no modo de função alternativa no registrador MODER, selecionamos o tipo de saída, habilitamos ou não os resistores de *pull-up/down* e selecionamos a velocidade nos registradores OTYPER, PUPDR e OSPEEDR, respectivamente. Em seguida, conectamos o pino de I/O à função alternativa desejada por meio dos registradores AFRL ou AFRH, dependendo do pino.

Para definir a função alternativa de um pino *y* de um GPIO (*y* = 0, 1, ..., 7), devemos configurar 4 bits contíguos (bits *4y: 4y+1: 4y+2: 4y+3*), chamados de AFRL*y*[3:0] no registrador AFRL. Os valores binários possíveis de configuração são os seguintes:

0b0000: AF0	0b0100: AF4	0b1000: AF8	0b1100: AF12
0b0001: AF1	0b0101: AF5	0b1001: AF9	0b1101: AF13
0b0010: AF2	0b0110: AF6	0b1010: AF10	0b1110: AF14
0b0011: AF3	0b0111: AF7	0b1011: AF11	0b1111: AF15

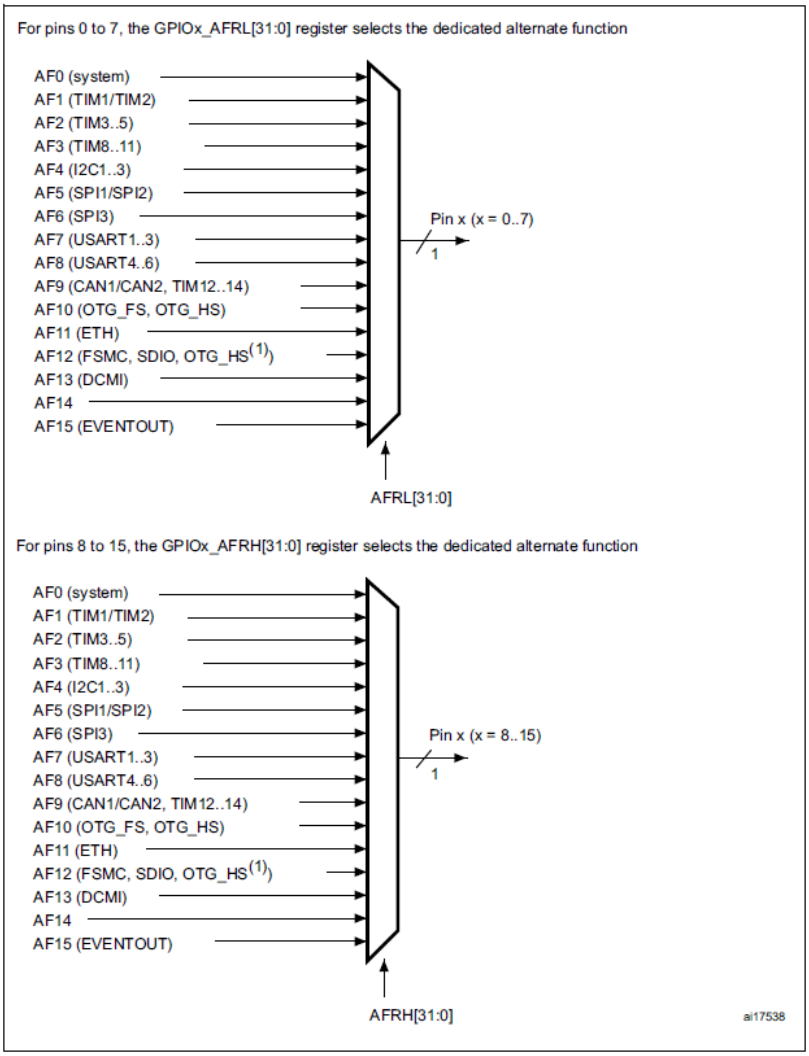


Figura 13 – Multiplexação das funções alternativas nos pinos do STM32F407.

Na Figura 14 é mostrada a organização do registrador AFRL e na Figura 15 a do registrador AFRH.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 14 – Registrador AFRL.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 15 – Registrador AFRH.

Nos arquivos de cabeçalho fornecidos pela ST Microelectronics com as definições dos registradores, os registradores AFRL e AFRH são tratados como um array de dois registradores AFR, sendo o AFRL nomeado como AFR[0] e AFRH como AFR[1].

Como exemplo, suponha que desejamos configurar o pino 3 (*y=3*) de uma porta para operar como um pino da interface serial USART (função alternativa AF7), sem alterar a configuração dos demais pinos. Nesse caso, devemos escrever o valor 0b0111 nos bits *4y: 4y+1: 4y+2: 4y+3*, isto é, nos bits 12, 13, 14 e 15, que são os bits AFRL3[3:0] no registrador AFRL, já que o pino é o de número 3, conforme mostrado na linha de código abaixo. Perceba que o registrador AFRL é acessado como AFR[0].

```
AFR[0] |= (0b0111 << 12); //configura a função alternativa AF7 no pino 3
                          //sem alterar a configuração dos demais pinos
```

Se a mesma configuração fosse feita em um pino *y* (*y* = 8, 9, ..., 15), deveríamos configurar 4 bits contíguos (bits *4y-32: 4y-31: 4y-30: 4y-29*), chamados de AFRH*y*[3:0] no registrador AFRH.

Por exemplo, suponha que agora desejamos configurar o pino 9 (*y=9*) para operar como um pino da interface serial USART1 (função alternativa AF7), sem alterar a configuração dos demais pinos. Nesse caso, devemos escrever o valor 0b0111 nos bits *4y-32: 4y-31: 4y-30: 4y-29*, isto é, nos bits 4, 5, 6 e 7, que são os bits AFRH9[3:0] no registrador AFRH, já que o pino é o de número 9, conforme mostrado na linha de código abaixo. Perceba que o registrador AFRH é acessado como AFR[1].

```
AFR[1] |= (0b0111 << 4);  //configura a função alternativa AF7 no pino 9
                          //sem alterar a configuração dos demais pinos
```

6.5. GPIO no modo de função analógica

Quando um pino é programado como função analógica, independentemente de ser entrada ou saída:

- o driver de saída é desativado
- a entrada Schmitt trigger é desativada, fornecendo consumo zero (alta impedância de entrada) para tensões analógicas no pino de I/O. A saída do disparador é forçada a um valor constante (0).
- os resistores de *pull-up/down* são desativados
- um acesso de leitura ao registrador de dados de entrada sempre obtém o valor "0"
- os registradores ODR e BSRR não tem efeito sobre o pino, já que o driver de saída é desabilitado.

Na Figura 16, é mostrada a configuração de hardware do pino no modo de função analógica:

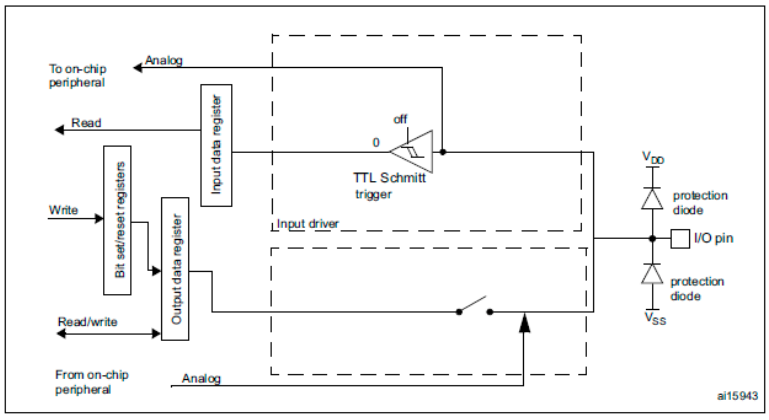


Figura 16 – Configuração de função analógica do hardware do bit da porta de I/O.

Nesse modo, o pino pode funcionar como uma entrada analógica dos conversores AD internos (ADC1, ADC2 ou ADC3), ou pode funcionar como uma saída analógica dos conversores DA internos (DAC1 ou DAC2), sendo o roteamento do pino para o periférico pré-determinado pelo hardware.

6.6. Mecanismo de travamento das configurações do GPIO

É possível travar a configuração dos registradores de controle GPIO aplicando uma sequência de escrita específica ao registrador de travamento (*Lock Register - LCKR*). Os registradores travados são MODER, OTYPER, OSPEEDR, PUPDR, AFRL e AFRH. O travamento das configurações é geralmente realizado pelo sistema operacional em um sistema embarcado para proteger as portas de entrada e saída de acessos indevidos realizados por softwares de terceiros.

Para escrever no registrador LCKR, uma sequência específica de escrita/leitura deve ser aplicada. Quando a sequência correta é aplicada ao bit 16 desse registrador, o valor dos bits LCKR[15:0] é usado para travar a configuração dos pinos nos registradores de controle. Durante a sequência de escrita no bit 16 de LCKR, o valor de LCKR[15:0] não deve ser alterado.

Quando o travamento for aplicado a um pino de uma porta, as configurações do pino não podem mais ser modificadas até o próximo reset do sistema. Cada bit que estiver setado no registrador LCKR congela a configuração dos bits correspondentes nos registradores de controle.

A sequência de bloqueio de configuração dos pinos da porta GPIO só pode ser executada usando um acesso com palavras de 32 bits ao registrador LCKR devido ao fato de que o bit 16 desse registrador deve ser escrito ao mesmo tempo que os bits [15:0].

A sequência de escrita em LCKR para habilitar o travamento das configurações é:

** Escrever '1' no bit 16 juntamente com os bits LCKR[15:0]*
** Escrever '0' no bit 16 juntamente com os bits LCKR[15:0]*
** Escrever '1' no bit 16 juntamente com os bits LCKR[15:0]*
** Ler o registrador LCKR*
** Ler se o bit 16 do registrador LCKR está setado (essa operação de leitura não é obrigatória, mas confirma que o travamento foi ativado)*

6.7. Acesso aos periféricos na linguagem C

Como mencionado, os registradores de controle e utilização dos periféricos são mapeados na memória e, portanto, podem ser acessados usando ponteiros. Por exemplo, podemos definir os registradores do GPIO como ponteiros da seguinte forma:

```
//Definição dos registradores do GPIO
#define MODER       (*((volatile unsigned long *) (0x40020000)))
#define OTYPER      (*((volatile unsigned long *) (0x40020004)))
#define OSPEEDR     (*((volatile unsigned long *) (0x40020008)))
#define PUPDR       (*((volatile unsigned long *) (0x4002000C)))
#define IDR         (*((volatile unsigned long *) (0x40020010)))
#define ODR         (*((volatile unsigned long *) (0x40020014)))
#define BSRRL       (*((volatile unsigned long *) (0x40020018)))
#define BSRRH       (*((volatile unsigned long *) (0x4002001C)))
```

Então, podemos usar a definição diretamente, como já havíamos feito nos exemplos anteriores:

```
MODER |= 0b11 << 10;       //seleciona o modo analógico no pino 5
int16_t leitura = IDR;     //lê o estado dos pinos de uma porta
ODR |= 1;                  //seta o pino 0
OSPEEDR |= 0b11 << 10;     //High speed no pino 5
```

Este método funciona para um pequeno número de registradores. No entanto, à medida que a quantidade aumenta, esse estilo de codificação pode ser problemático porque para cada definição de endereço de registrador, o programa precisa armazenar a constante de endereço de 32 bits, resultando no aumento do tamanho do código. Quando há múltiplas instanciações do mesmo periférico, por exemplo, o microcontrolador STM32F407 possui cinco periféricos GPIO (GPIOA, GPIOB, GPIOC, GPIOD e GPIOE), e a mesma definição deve ser repetida para cada uma das instanciações. Isso não é escalável e dificulta a manutenção do software. Não é fácil criar uma função que

possa ser compartilhada entre múltiplas instanciações do mesmo periférico. Por exemplo, pode ser necessário criar a mesma função para cada uma das portas GPIO, resultando em um tamanho de código maior.

Para resolver esses problemas, a prática comum é definir os registradores dos periféricos como estruturas de dados. Por exemplo, no pacote de software de driver de dispositivo dos fornecedores de microcontroladores, podemos encontrar:

```
typedef struct
{
  __IO uint32_t MODER;    //GPIO mode register. Address offset: 0x00
  __IO uint32_t OTYPER;   //GPIO output type register. Address offset: 0x04
  __IO uint32_t OSPEEDR;  //GPIO output speed register. Address offset: 0x08
  __IO uint32_t PUPDR;    //GPIO pull-up/pull-down register. Address offset: 0x0C
  __IO uint32_t IDR;      //GPIO input data register. Address offset: 0x10
  __IO uint32_t ODR;      //GPIO output data register. Address offset: 0x14
  __IO uint16_t BSRRL;    //GPIO bit set/reset low register. Address offset: 0x18
  __IO uint16_t BSRRH;    //GPIO bit set/reset high register. Address offset: 0x1A
  __IO uint32_t LCKR;     //GPIO configuration lock register. Address offset: 0x1C
  __IO uint32_t AFR[2];   //GPIO alternate function registers. Address offset: 0x20-0x24
} GPIO_TypeDef;
```

Então, os endereços bases de cada periférico (GPIOA até o GPIOE) são definidos como ponteiros para esse tipo de estrutura de dados:

```
//Endereço base dos periféricos do microcontrolador
#define PERIPH_BASE ((uint32_t)0x40000000)
...
//Endereço base dos periféricos conectados ao barramento AHB1
#define AHB1PERIPH_BASE (PERIPH_BASE + 0x00020000)
...
#define GPIOA_BASE (AHB1PERIPH_BASE + 0x0000)
#define GPIOB_BASE (AHB1PERIPH_BASE + 0x0C00)
#define GPIOC_BASE (AHB1PERIPH_BASE + 0x1000)
#define GPIOD_BASE (AHB1PERIPH_BASE + 0x1400)
#define GPIOE_BASE (AHB1PERIPH_BASE + 0x1800)
...
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE ((GPIO_TypeDef *) GPIOE_BASE)
```

Com isso, para podermos acessar um registrador do periférico, que agora é um elemento de uma struct endereçada por um ponteiro, podemos usar o operador seta (->), indicando o ponteiro da estrutura e o membro que desejamos acessar. Os mesmos exemplos da página anterior, acessando os registradores do GPIOA, por exemplo, seriam escritos da seguinte forma:

```
GPIOA->MODER |= 0b11 << 10;      //seleciona o modo analógico no pino 5 da porta A
int16_t leitura = GPIOA->IDR;    //lê o estado dos pinos da porta A
GPIOA->ODR |= 1;                 //seta o pino 0 da porta A
GPIOA->OSPEEDR |= 0b11 << 10;    //High speed no pino 5 da porta A
```

Para facilitar a compreensão dessa sintaxe, podemos interpretar uma linha como `GPIOA->MODER` dizendo que estamos acessando o registrador MODER do periférico GPIOA.

Nesses trechos de código, há várias coisas novas que não foram abordadas:

O "__IO" é definido em um arquivo de cabeçalho padronizado no CMSIS. Implica um item de dados volátil (por exemplo, um registrador de um periférico), que pode ser lido ou escrito pelo software. Além de "__IO", um registrador de periférico também pode ser definido como "__I" (somente leitura) e "__O" (somente gravação).

```
#ifdef __cplusplus
#define __I volatile             //defines 'read only' permissions
#else
#define __I volatile const       //defines 'read only' permissions
#endif
#define __O volatile             //defines 'write only' permissions
#define __IO volatile            //defines 'read / write' permissions
```

O "uint32_t" (inteiro de 32 bits sem sinal) é um tipo de dados suportado em C99. Isso garante que o tamanho dos dados seja de 32 bits, independentemente da arquitetura do processador, o que pode ajudar o software a ser mais portátil. Para usar este tipo de dado, o projeto precisa incluir o cabeçalho do tipo de dados padrão, o que já é feito ao se utilizar o CMSIS:

```
#include <stdint.h> //Include standard types
/* C99 standard data types:
       uint8_t:  unsigned 8-bit,
       int8_t:   signed 8-bit,
       uint16_t: unsigned 16-bit,
       int16_t : signed 16-bit,
       uint32_t: unsigned 32-bit,
       int32_t : signed 32-bit,
       uint64_t: unsigned 64-bit,
       int64_t : signed 64-bit
*/
```

Quando os periféricos são declarados usando esse método, podemos criar funções que podem ser usadas para cada instância do periférico facilmente. Este método para declarar registradores de periféricos é usado por quase todos os pacotes de driver de dispositivo dos microcontroladores Cortex-M, incluindo o STM32.

6.8. Exemplo de configuração do GPIO

Todos os registradores de controle e de dados das portas de GPIO são alimentados com o clock do barramento de alta velocidade AHB1. Entretanto, apesar do clock dos barramentos AHB estar ligado desde o momento de energização do microcontrolador, esse sinal não chega diretamente aos registradores para minimizar o consumo de energia. Ao invés disso, o clock passa por uma porta AND que é habilitada por bits de controle no periférico responsável pelo controle de clock, isto é, o módulo RCC (*Reset and Clock Control*). É necessário habilitar o clock de cada periférico, incluindo as portas de GPIO, antes de utilizá-lo, inclusive para configurá-lo.

Os sinais de clock das portas de GPIO são individualmente habilitados no registrador AHB1ENR (*AHB1 Enable Register*) do módulo RCC, mostrado na Figura 17, setando os bits dos periféricos que se deseja utilizar. O bit 0 do registrador AHB1ENR (GPIOAEN) habilita o clock da porta GPIOA e, com isso, podemos acessar e configurar todos os registradores dessa porta.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHS EN	ETHM ACPTPEN	ETHM ACRXEN	ETHM ACTXEN	ETHMA CEN	Reserved		DMA2EN	DMA1EN	CCMDAT ARAMEN	Res.	BKPSR AMEN	Reserved	
	rw	rw	rw	rw	rw	rw			rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 17 – Registrador AHB1ENR do módulo RCC.

O código abaixo exemplifica a configuração do pino PA0 como saída digital, sem alterar a configuração dos demais pinos. Perceba que antes de manipular os registradores de configuração do pino, o clock da porta A deve ser habilitado, uma vez que esses registradores são alimentados por esse sinal de clock. O pino é configurado como saída *push-pull*, sem resistores de *pull-up/down* , *slew rate* em very *high speed* e, ao final, leva o pino para nível lógico alto.

```
RCC->AHB1ENR |= 1;          //habilita o clock do GPIOA

GPIOA->MODER |= 1;          //seleciona modo de saída digital no pino PA0
GPIOA->OTYPER &= ~1;        //seleciona o tipo de saída como push-pull em PA0
GPIOA->PUPDR &= ~3;         //desabilita os resistores de pull-up/down
GPIOA->OSPEEDR |= 3;        //seleciona a velocidade very high speed

GPIOA->ODR |= 1;            //faz o estado do pino como HIGH
```

Já o código abaixo exemplifica a configuração do pino PA0 como entrada digital e o pino PA6 como saída digital, sem alterar a configuração dos demais pinos. O pino PA6 opera com saída *push-pull*, sem resistores de *pull-up/down, slew rate* em Low *speed* e no pino PA0 é ligado o resistor de *pull-down*. O programa lê o estado do pino PA0 e o reproduz no pino PA6.

```
RCC->AHB1ENR |= 1;               //habilita o clock do GPIOA

//Configuração do pino PA0 como entrada digital com resistor de pull-down
GPIOA->MODER &= ~(0b11);         //seleciona modo de entrada digital no pino PA0
GPIOA->PUPDR |= 0b10;            //habilita o resistor de pull-down no pino PA0

//Configuração do pino PA6 como saída digital
GPIOA->MODER |= (0b01 << 12);    //seleciona modo de saída digital no pino PA6

while(1)
{
   if(GPIOA->IDR & 1)             //faz a leitura da porta A e verifica se PA0 é alto
      GPIOA->ODR |= (1 << 6);     //faz o estado do pino como HIGH se PA0 for alto
   else
      GPIOA->ODR &= ~(1 << 6);    //faz o estado do pino como LOW se PA0 for baixo
}
```

Até agora, foi mostrado como configurar e utilizar pinos de GPIO fazendo o acesso direto aos registradores das portas, o que permite uma manipulação de mais baixo nível nos pinos do microcontrolador, garantindo operações mais rápidas do que usando funções de bilbioteca.

De um modo geral, manipular os pinos GPIO a partir do acesso direto aos registradores tem algumas implicações negativas:

* Se torna muito mais difícil para depurar e manter o código, e é menos claro para outras pessoas entenderem. Só leva alguns nanossegundos para o processador executar o código, mas pode levar horas para que se descubra por que ele não está funcionando corretamente e consertá-lo!
* O código é menos portável. É muito mais fácil escrever um código que possa ser executado em qualquer microcontrolador STM32, uma vez que os registradores de controle das portas, ou de outros periféricos, podem ser diferentes em cada tipo de microcontrolador.
* É muito mais fácil causar mal funcionamento no microcontrolador com o acesso direto à porta. Seria muito fácil, por exemplo, acidentalmente fazer com que uma porta serial parasse de funcionar alterando a direção dos pinos de entrada e saída serial.

Entretanto, as implicações positivas na manipulação direta de portas são:

* Talvez seja necessário ativar e desativar os pinos de modo muito mais rápido, em frações de microssegundo. As funções de alto nível para manipulação de portas são formadas por uma dúzia de linhas de código que fazem o aceso direto aos registradores, que são compiladas em algumas instruções de máquina. Cada instrução de máquina requer um ciclo de clock a 168 MHz, que pode se somar em aplicações sensíveis ao tempo, como aplicações de processamento digital de sinais em tempo real. O acesso direto à porta pode fazer o mesmo trabalho em muito menos ciclos de clock.
* Às vezes é necessário acionar vários pinos de saída exatamente ao mesmo tempo. Isso só é obtido fazendo o acesso direto aos registradores;
* Se há restrições quanto ao espaço de memória de programa disponível, pode-se usar a manipulação direta de portas para reduzir o código. Exige-se muito menos bytes de código compilado para configurar vários pinos de hardware simultaneamente através dos registradores de portas do que para configurar cada pino separadamente.

CMSIS é uma sigla para *Cortex Microcontroller Software Interface Standard*, um padrão criado pela própria ARM que define uma camada de abstração (API) de acesso ao hardware para processadores da linha Cortex-M. Na prática, o CMSIS é um conjunto de arquivos "*.c*" e "*.h*", com diversas definições e funções pré fornecidas, que possibilitam a criação de um *BSP* (*Board Support Package*) para microcontroladores da linha ARM Cortex-M. Dessa forma, se você usar as funções definidas pelo padrão CMSIS na sua aplicação, poderá migrar mais facilmente para outros chips da mesma família. A curva de aprendizado também é menor, já que você não precisa aprender como os registradores controlam a operação de um determinado periférico, basta aprender a interface provida pelo CMSIS.

Por exemplo, o código abaixo utiliza funções e estrututras (*structs*) baseadas no CMSIS, fornecidas pela ST Microelectronics, fabricante do STM32, para efetuar a configuração do pino PA0 como saída digital do tipo *push-pull*, sem resistores de *pull-up/down , slew rate* em Very *High Speed* e, ao final, leva o pino para nível lógico alto, sem alterar a configuração dos demais pinos.

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);      //habilita o clock do GPIOA

GPIO_InitTypeDef GPIO_InitDef;              //estrutura para configuração de GPIO

GPIO_InitDef.GPIO_Pin = GPIO_Pin_0;         //especifica o pino a ser configurado
GPIO_InitDef.GPIO_Mode = GPIO_Mode_OUT;     //GPIO como saída
GPIO_InitDef.GPIO_OType = GPIO_OType_PP;    //saída push-pull (PP)
GPIO_InitDef.GPIO_PuPd = GPIO_PuPd_NOPULL; //desabilita os resistores de pull-up/down
GPIO_InitDef.GPIO_Speed = GPIO_High_Speed; //velocidade (very high speed)
GPIO_Init(GPIOA, &GPIO_InitDef);            //configura o pino conforme especificado

GPIO_WriteBit(GPIOA, GPIO_Pin_0, Bit_SET); //faz o estado do pino como high
```

6.9. Componentes adicionais

Esta seção cobre a maioria dos componentes mais utilizados na prática que se conectam às entradas e saídas dos microcontroladores, tais como chaves, botões, transistores, LED's, relés, etc.

Chaves e botões

Chaves e botões de pulso (*push-buttons*) são provavelmente os mais simples dispositivos de entrada que possibilitam o aparecimento de tensão em um pino do microcontrolador. Eles são ligados aos pinos configurados como entradas. Entretanto, fazer essa conexão não é tão simples quanto parece. A razão para isso é o que se conhece por "*bounce*", ou oscilação, ou ainda, ruído elétrico.

A oscilação produzida pelos contatos no sinal elétrico de uma chave mecânica é um problema comum que deve ser tratado pelo projetista de circuitos microcontrolados. Esta oscilação causa problemas em alguns circuitos analógicos e digitais que respondem rapidamente às variações de tensão em seus pinos.

Na Figura 18, são apresentadas 4 formas de conectar uma chave, ou botão, a um pino do microcontrolador. Todas as formas são encontradas em sistemas microcontrolados, sendo diferentes opções do projetista de hardware.

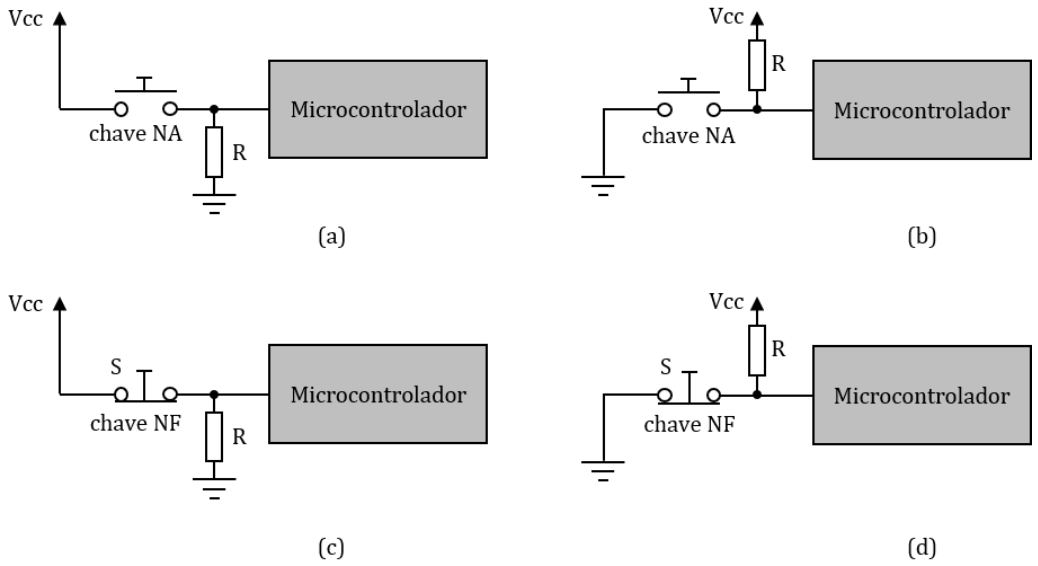


Figura 18 – Diferentes formas de se conectar uma chave a um pino de um microcontrolador.

Nas Figura 18(a) e 18(b), temos uma chave normalmente aberta (NA), onde os contatos só são fechados ao se pressionar a chave. Nas Figura 18(c) e 18(d), temos uma chave normalmente fechada (NF), onde os contatos são abertos ao se pressionar a chave.

Na Figura 18(a), enquanto a chave não for pressionada, o nível lógico presente no pino do microcontrolador é puxado para baixo devido ao resistor de *pull-down,* R. Quando a chave for pressionada, o sinal Vcc é aplicado diretamente no pino, levando-o para nível lógico alto. Na Figura 18(b), ocorre o contrário, enquanto a chave não for pressionada, o nível lógico presente no pino é forçado em alto devido ao resistor de *pull-up*. Quando a chave for pressionada, o pino é conectado diretamente no terra, levando-o para nível lógico baixo.

Na Figura 18(c), a chave conecta diretamente o pino do microcontrolador ao sinal Vcc e, enquanto a chave não for pressionada, o nível lógico presente no pino é alto. Quando a chave for pressionada, o nível lógico presente no pino é puxado para baixo devido ao resistor de *pull-down*. Na Figura 18(d), ocorre o contrário, a chave conecta diretamente o pino do microcontrolador ao terra e, enquanto a chave não for pressionada, o nível lógico presente no pino do microcontrolador é baixo. Quando a chave for pressionada, o nível lógico presente no pino é levado para alto ao resistor de *pull-up*.

Os resistores de *pull-up/down* da Figura 18 podem ser resistores externos ou, no caso do STM32, pode-se utilizar os resistores internos conectados em cada pino de GPIO.

Em qualquer uma das formas de conexão, o comportamento do sinal no pino durante uma transição de nível lógico é o mesmo, mostrado na Figura 19. Ao se pressionar a chave, a tensão no pino, idealmente, deveria alternar de estado instantaneamente. Mas, em vez disso, devido às imperfeições das superfícies dos contatos da chave, ela oscila enquanto o chaveamento do contato mecânico da chave não se estabilizar. Isso ocorre rapidamente, durando cerca de 0,01 a até 100ms, dependendo do tipo de chave.

Apesar do transitório ocorrer rapidamente, o microcontrolador percebe todas essas variações de tensão devido à sua alta velocidade de aquisição dos níveis lógicos presentes nos pinos de entrada. Assim, por exemplo, ao usar o microcontrolador para contar a quantidade de vezes que uma chave é pressionada, podem ocorrer erros de contagem devido a tais oscilações.

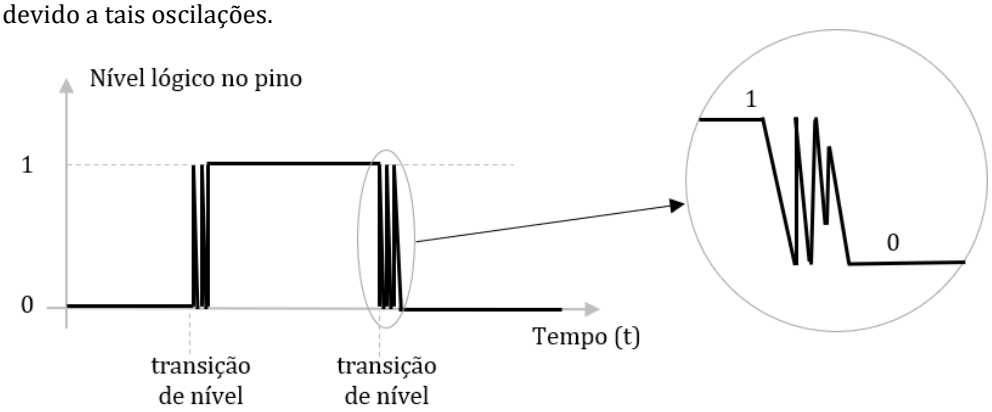


Figura 19 – Representação da tensão em um pino de entrada onde foi conectado uma chave.

Este problema pode ser resolvido conectando um circuito do tipo filtro RC para suprimir essas rápidas mudanças de tensão, uma vez que a tensão dos capacitores não pode variar rapidamente. Uma vez que o período de duração das oscilações não é definido, os valores dos componentes não podem ser precisamente determinados. Em muitos casos, é recomendado usar os valores mostrados nos exemplos da Figura 20.

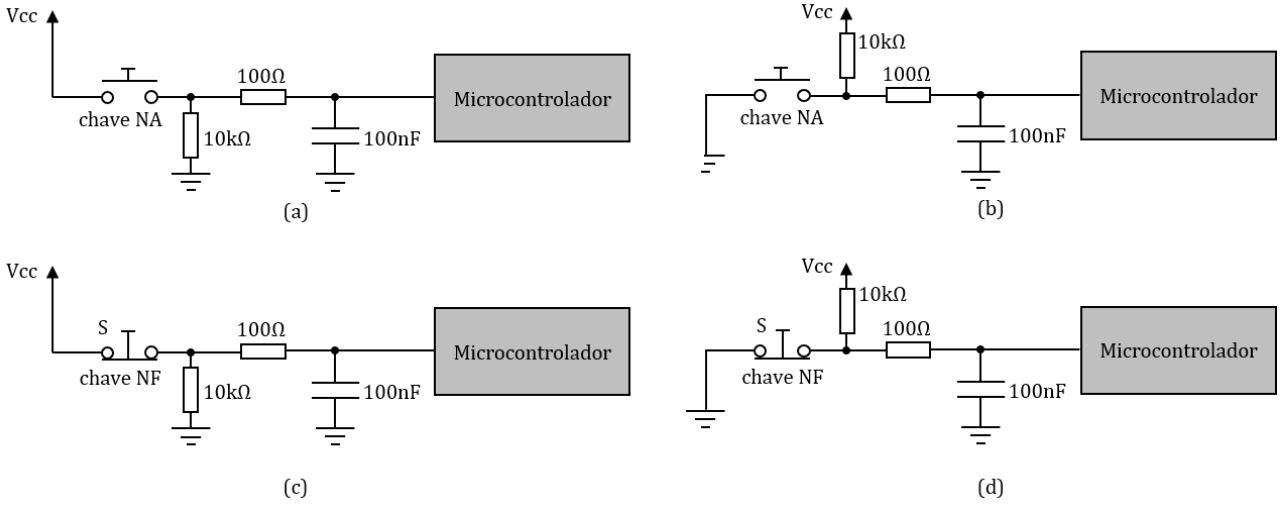


Figura 20 – Filtros RC conectados no pino do microcontrolador para contornar o efeito do chaveamento.

Alternativamente, ou em adição a esta solução por hardware, há também outras soluções de filtragem de ruídos por software: por exemplo, quando o programa testa o estado lógico de um pino de entrada e detecta uma mudança, o teste deve ser refeito depois de um determinado intervalo de tempo. Se o programa confirmar a mudança, isso pode significar que o botão, ou a chave, realmente mudou de posição e então o programa executa a tarefa associada. Filtros mais sofisticados, como filtros digitais passa-baixas também podem ser aplicados, dependendo da necessidade.

LEDs

Os LEDs são diodos emissores de luz que são utilizados, na maioria dos sistemas microcontrolados, como excelentes sinalizadores em substituição às lâmpadas comuns devido à sua praticidade de uso, longa vida útil e baixo custo. Na Figura 21 são mostrados alguns dos tipos de LEDs mais comumente utilizados em sistemas microcontrolados.

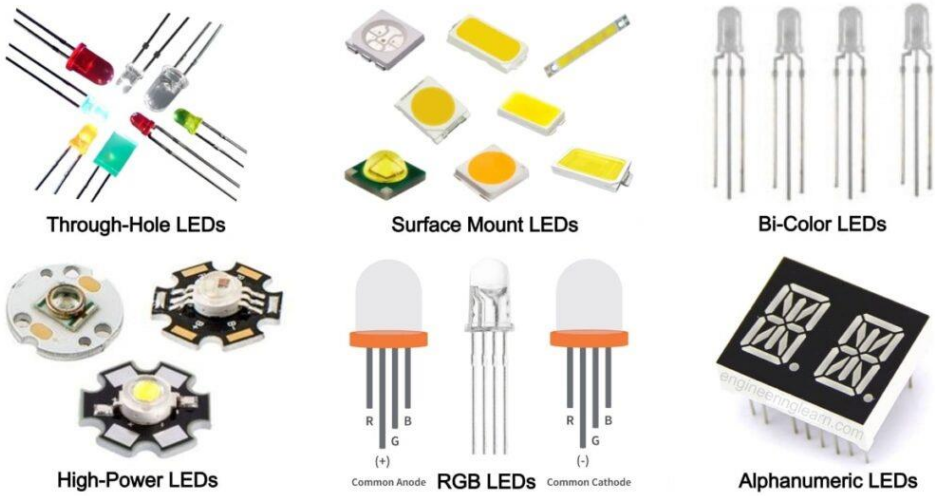


Figura 21 – Diferentes tipos de LEDs usados em sistemas microcontrolados.

Na Figura 22, são mostradas duas formas básicas de se conectar um LED a um pino de saída de um microcontrolador. Na Figura 22(a), o LED acende quando polarizado diretamente, isto é, quando o nível lógico alto está presente no pino, e apaga com nível baixo. Já na Figura 22(b), o LED acende quando o nível baixo está presente no pino e apaga com nível alto. Em ambos os casos, o resistor R, conectado em série com o LED, limita a corrente e compatibiliza a tensão de saída do pino com a tensão de polarização do LED, que é geralmente menor que a tensão que o pino fornece.

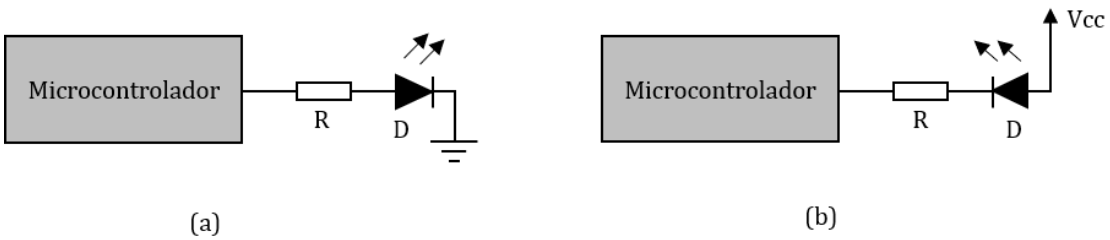


Figura 22 – Diferentes formas de coxão de um LED ao microcontrolador.

Displays de 7 segmentos

Um display de sete segmentos é composto de sete elementos (LEDs) que podem ser ligados ou desligados individualmente. Eles podem ser combinados para produzir representações simplificadas de alguns algarismos alfanuméricos. Também há displays de catorze e de dezesseis segmentos para exibição plena de caracteres alfanuméricos. Todavia, estes têm sido substituídos por displays de matriz de pontos, LCDs alfanuméricos ou displays gráficos com maiores flexibilidades de exibição.

Para simplificar as conexões, os anodos, ou catodos, de todos os segmentos de um displays de 7 segmentos são conectados a um terminal comum e com isso podemos ter displays de anodo comum e displays de catodo comum. Os segmentos do display são definidos pelas letras de "*a*" a "*g*", conforme indicado na Figura 23, onde o ponto decimal opcional *dp* (um "oitavo segmento") é usado para a exibição de números não inteiros.

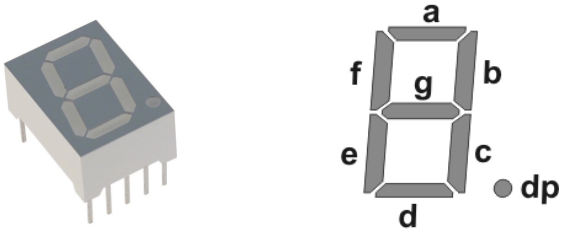


Figura 23 - Display de 7 segmentos e sua representação.

A formação e exibição dos caracteres são feitas acionando-se determinados segmentos e apagando-se outros. Os dígitos obtêm aparência reconhecível através da aplicação de um processo chamado "mascaramento". Em outras palavras, o formato binário de cada dígito é substituído por uma combinação diferente de bits. Por exemplo, o número de 8 bits 0b00001000, que representa 8 em decimal, é substituído pelo número binário 0b01111111, a fim de ativar todos os LEDs, exibindo o dígito 8 no display.

Se uma porta do microcontrolador é conectada ao display de tal forma que o bit 0 ativa o segmento "*a*", o bit 1 ativa o segmento "*b*", e assim sucessivamente, então a tabela abaixo mostra a máscara para cada dígito hexadecimal, considerando que o display seja do tipo catodo comum. Para displays do tipo anodo comum, todos os "uns" (1) da tabela devem ser substituídos por "zeros" (0), e vice-versa.

Dígitos hexaecimais	**Segmentos do display**							**Máscara (g f e d c b a)**
	a	**b**	**c**	**d**	**e**	**f**	**g**	
0	1	1	1	1	1	1	0	0b00111111
1	0	1	1	0	0	0	0	0b00000110
2	1	1	0	1	1	0	1	0b01011011
3	1	1	1	1	0	0	1	0b01001111
4	0	1	1	0	0	1	1	0b01100110
5	1	0	1	1	0	1	1	0b01101101
6	1	0	1	1	1	1	1	0b01111101
7	1	1	1	0	0	0	0	0b00000111
8	1	1	1	1	1	1	1	0b01111111
9	1	1	1	1	0	1	1	0b01111011
A	1	1	1	0	1	1	1	0b01110111
B	0	0	1	1	1	1	1	0b00011111
C	1	0	0	1	1	1	0	0b01001110
D	0	1	1	1	1	0	1	0b00111101
E	1	0	0	1	1	1	1	0b01001111
F	1	0	0	0	1	1	1	0b01000111

Um exemplo de aplicação é mostrado na Figura 24, onde o número 5, em binário, é mascarado com o valor 0b01101101 e em seguida é exibido no display de 7 segmentos, no qual cada LED possui um resistor limitador de corrente de 330 Ω.

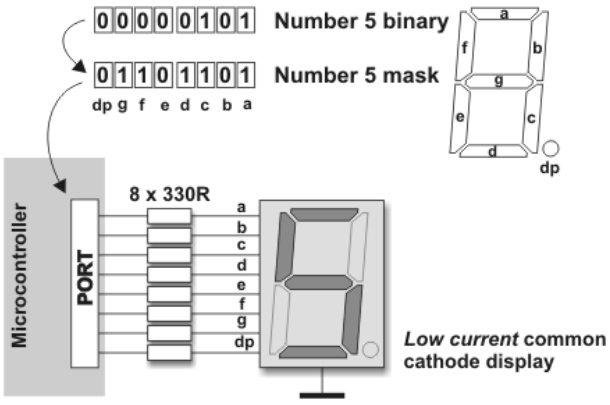


Figura 24 – Mascaramento e exibição do dígito 5 no display de sete segmentos.

Uma tabela com as máscaras dos dígitos hexadecimais para o display de 7 segmentos poderia ser armazenada na memória de um microcontrolador com o uso de matrizes, como mostrado a seguir. A tabela de constantes é do tipo u`int8_t` e possui 16 elementos, referenciados como mask[0] até mask[15].

```
// Decodificação de display de 7 segmentos com uso de matrizes
const uint8_t mask[16]={   0b00111111,   //0
                           0b00000110,   //1
                           0b01011011,   //2
                           0b01001111,   //3
                           0b01100110,   //4
                           0b01101101,   //5
                           0b01111101,   //6
                           0b00000111,   //7
                           0b01111111,   //8
                           0b01101111,   //9
                           0b01110111,   //A
                           0b01111100,   //B
                           0b00111001,   //C
                           0b01011110,   //D
                           0b01111001,   //E
                           0b01110001}; //F
```

Pela Figura 24, percebe-se que são necessários pelo menos 7 terminais do microcontrolador para fazer a correta exibição dos dígitos em um display. Imaginemos então a situação em que é necessário fazer o controle de 4 displays simultaneamente. Seriam então necessários 28 terminais para essa tarefa. Entretanto, existe uma técnica de multiplexação de displays que consiste em usar apenas 7 terminais, comuns a todos os displays, além de quatro terminais extras para selecionar um dos displays por vez. Esta é uma técnica que se beneficia de uma propriedade do olho humano, chamada persistência à luz, para criar uma ilusão de ótica. Assim, se a seleção do display for suficientemente rápida, o efeito obtido é como se os 4 displays estivessem ligados ao mesmo tempo, embora apenas um esteja ativo por vez.

Um circuito que executa tal tarefa é mostrado na Figura 25. Inicialmente, um byte representando as unidades deve ser enviado à porta (PORT2) ao mesmo tempo em que o transistor T1 é ativado pelo pino correspondente na outra porta (PORT1). Após um curto intervalo de tempo, o transistor T1 deve ser desligado, um segundo byte, representando as dezenas, deve ser enviado ao PORT2 e o transistor T2 ativado. Esse processo é repetido ciclicamente, em alta velocidade, para todos os dígitos e transistores correspondentes.

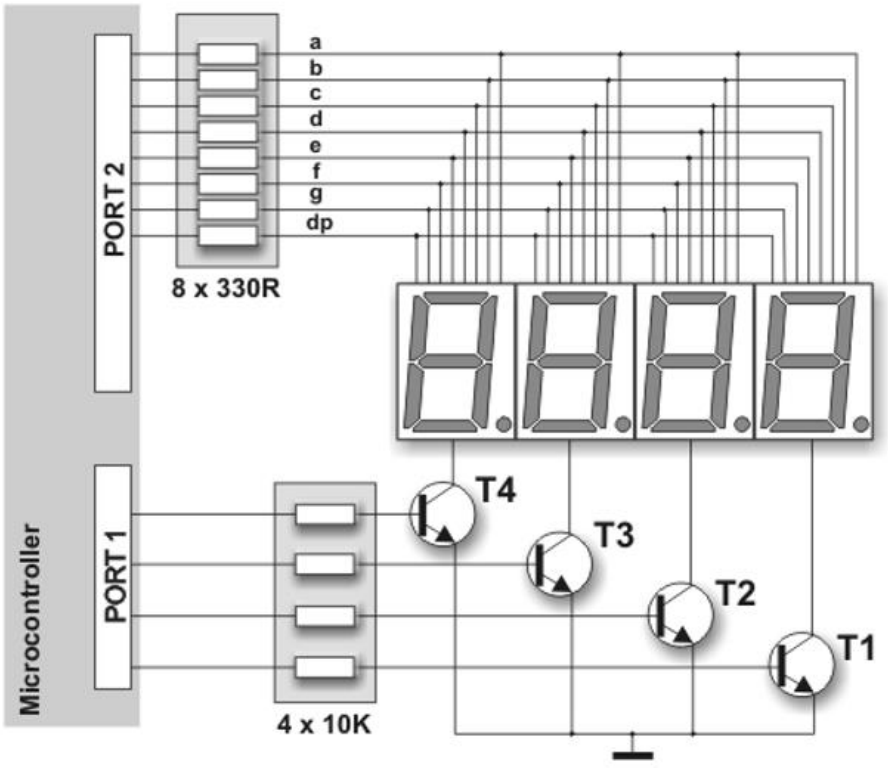


Figura 25 – Multiplexação de quatro displays de sete segmentos.

Display de cristal líquido (*Liquid Crystal Display* - LCD)

Este componente é fabricado especificamente para ser usado com microcontroladores, o que significa que não pode ser ativado por circuitos discretos comuns. O modelo aqui descrito é de baixo custo e grande aplicabilidade. É baseado no controlador de displays HD44780 (Hitachi), ou equivalente, e pode exibir mensagens em duas linhas com 16 caracteres cada (display 2x16), onde um caractere é composto por uma matriz de 5x8 ou 5x11 pontos, como mostrado na Figura 26. Ele pode exibir todas as letras do alfabeto, letras gregas, sinais de pontuação, símbolos matemáticos, etc. Também é possível exibir símbolos personalizados, criados pelo usuário. Outras características úteis incluem deslocamento automático de mensagem (para esquerda ou direita), diodo emissor de luz de fundo (*backlight*), exibição de cursor, etc.

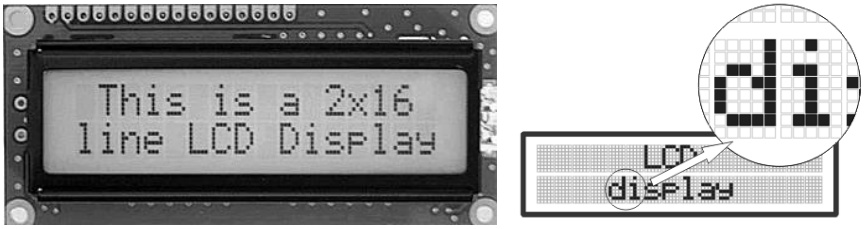


Figura 26 – Display 2x16 e detalhe de composição do caractere.

O display é alimentado por uma tensão de 5 V aplicada aos pinos VCC e GND e possui um terminal para ajuste do constraste. Variando a tensão de 0 a 5V aplicada ao terminal *VEE*, podemos variar o contraste da tela do display. Um trimpot é normalmente usado para este fim. Em aplicações mais elaboradas, um sinal PWM com um filtro passa-baixas pode ser usado para fazer o ajuste de contraste por meio de software.

Alguns desses displays têm embutido um backlight (LEDs de cor azul ou verde). Quando usado, um resistor limitador de corrente deve ser conectado em série a um dos pinos do backlight.

Um circuito básico de alimentação e controle de constraste do display é mostrado na figura 27.

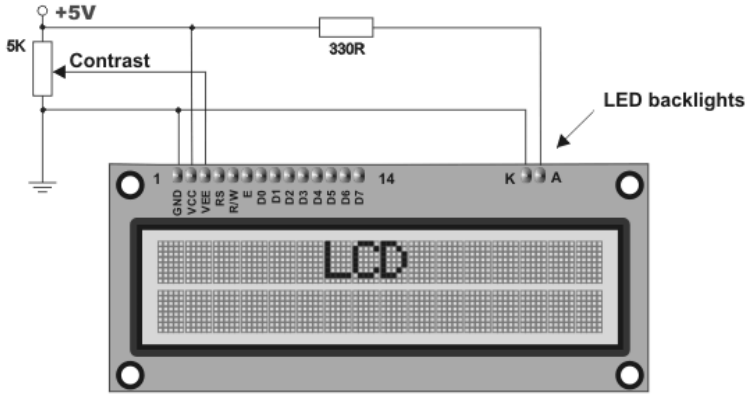


Figura 27 – Circuito básico de alimentação e ajuste do contraste do display.

O microcontrolador se comunica com o display, enviando ou recebendo informações, através de dois barramentos: um de controle e outro de dados. O barramento de controle é formado pelos terminais *RS, R/W* e *E,* que controlam a escrita e leitura no display. O barramento de dados é formado pelos terminais D0 a D7.

Há dois modos de uso do barramento de dados: modo de comunicação de 8 bits e modo de 4-bits. O modo apropriado é selecionado no início da operação do dispositivo em um processo chamado de "inicialização do display".

O modo de 8 bits usa as linhas D0-D7 para transferir dados de e para o LCD. O objetivo principal do modo de 4 bits é economizar o uso de pinos do microcontrolador. Nesse modo, apenas os 4 bits superiores (D4-D7) são usados para a comunicação, enquanto os outros podem ser deixados desligados. Em contrapartida, isso torna a comunicação um pouco mais lenta.

Outras interfaces com o display de cristal líquido ainda podem ser usadas, como uma interface serial TWI (I2C), que exige apenas dois pinos do microcontrolador.

Para a utilização desse tipo de display, é indicado o uso de funções de biblioteca que tornam transparente a inicialização e o controle da comunicação com o display.

O circuito completo de controle de um LCD é mostrado na figura 28.

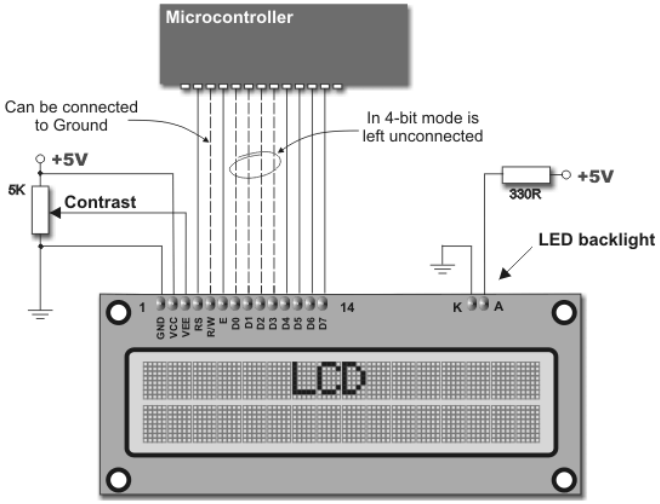


Figura 28 – Circuito completo de controle de um LCD.

Relés

Um relé (Figura 29) é um interruptor eletromecânico que abre e fecha sob o controle de outro circuito elétrico. É, portanto, ligado aos pinos de saída do microcontrolador e usado para ativar/desativar dispositivos de alta potência, como motores, transformadores, aquecedores, lâmpadas, etc. Existem vários tipos de relés, mas todos eles funcionam da mesma maneira.

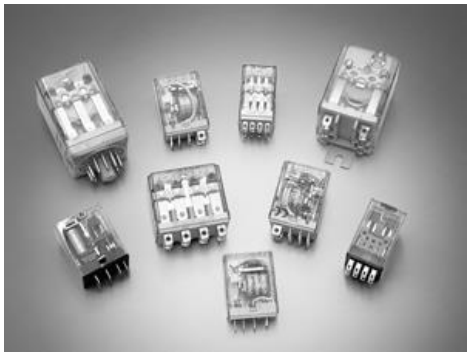


Figura 29 – Modelos de relés eletromecânicos.

O relé é operado por um eletroímã para abrir ou fechar um ou mais conjuntos de contatos mecânicos quando uma corrente elétrica flui através de sua bobina. Similarmente aos acopladores ópticos, não há nenhuma conexão elétrica entre os circuitos de entrada e saída, havendo apenas o acoplamento magnético da bobina que atrai e movimenta os contatos metálicos, os fazendo abrir ou fechar.

Na Figura 30 é mostrada a solução mais comumente empregada para se conectar um relé a um pino do microcontrolador.

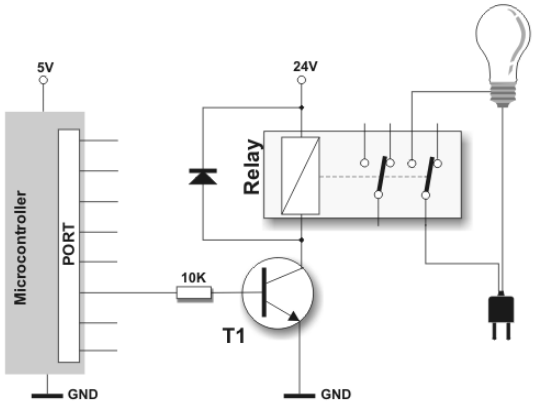


Figura 30 – Ligação de um relé a um pino do microcontrolador.

No circuito mostrado na Figura 30, o microcontrolador pode ligar/desligar a lâmpada, que é uma carga considerada de alta potência (quando comparada à potência do próprio microcontrolador) a partir do acionamento do relé. Uma vez que as correntes e tensões fornecidas diretamente nos pinos de saída do microcontrolador são normalmente bem pequenas, isso normalmente é insuficiente para ativar diretamente o relé. Assim, um transistor (T1) é utilizado como driver amplificador para ativar o relé.

A fim de prevenir o aparecimento de um alto valor de tensão auto induzida, causada por uma interrupção brusca do fluxo de corrente através da bobina quando o relé é desenergizado, um diodo inversamente polarizado, chamado de "diodo de roda livre", é ligado em antiparalelo com a bobina do relé. O objetivo deste diodo é manter a corrente fluindo pela bobina até que o seu valor seja reduzido até zerar completamente, eliminando os de picos de tensão no circuito. ■