

6. INTERRUPÇÕES

Neste capítulo, apresentam-se os conceitos relativos às interrupções, detalhando-se as interrupções externas proveniente de estímulos externos ao microcontrolador. As demais interrupções existentes no ATmega são tratadas nos capítulos subsequentes.

Por definição, interrupção é um processo pelo qual um dispositivo externo ou interno pode interromper a execução de uma determinada tarefa do microcontrolador e solicitar a execução de outra. Elas permitem que um programa responda a determinados eventos no momento em que eles ocorrem.


As interrupções são muito importantes no trabalho com microcontroladores porque permitem simplificar o código em determinadas situações, tornando mais eficiente o desempenho do processamento. Um exemplo clássico é o teclado de um computador: o processador não fica monitorando se alguma tecla foi pressionada; quando isso ocorre, o teclado gera um pedido de interrupção e a CPU interrompe o que está fazendo para executar a tarefa necessária. Assim, não há desperdício de tempo e de processamento.

6.1 INTERRUPÇÕES NO ATMEGA328

As interrupções no AVR são ‘vetoradas’, significando que cada interrupção tem um endereço de atendimento fixo na memória de programa. Se cada interrupção não possuísse esse endereço fixo, um único endereço seria destinado para todas as interrupções. Desta forma, para executar a tarefa de uma determinada interrupção, no caso de haver mais de uma habilitada, o programa teria que testar qual foi a interrupção ocorrida. Havendo um endereço fixo por interrupção, o programa se torna mais eficiente.

Todas as interrupções no AVR também são ‘mascaráveis’, ou seja, elas podem ser individualmente habilitadas ou desabilitadas agindo-se sobre um bit específico. O sistema de interrupção do AVR possui, ainda, um bit que pode desabilitar ou habilitar todas as interrupções de uma só vez, o bit I do registrador de *status* (SREG). Na tab. 6.1, apresenta-se o endereço de cada interrupção na memória de programa do ATmega328, existindo um total de 26 diferentes tipos de interrupções. A ordem dos endereços determina o nível de prioridade das interrupções, quanto menor o endereço do vetor de interrupção, maior será a sua prioridade. Por exemplo, a interrupção INT0 tem prioridade sobre a INT1.

Tab. 6.1 – Interrupções do ATmega328 e seus endereços na memória de programa.

Vetor	End.	Fonte	Definição da Interrupção	Prioridade
1	0x00	RESET	Pino externo, Power-on Reset, Brown-out Reset e Watchdog Reset	
2	0x01	INT0	interrupção externa 0	
3	0x02	INT1	interrupção externa 1	
4	0x03	PCINT0	interrupção 0 por mudança de pino	
5	0x04	PCINT1	interrupção 1 por mudança de pino	
6	0x05	PCINT2	interrupção 2 por mudança de pino	
7	0x06	WDT	estouro do temporizador <i>Watchdog</i>	
8	0x07	TIMER2 COMPA	igualdade de comparação A do TC2	
9	0x08	TIMER2 COMPB	igualdade de comparação B do TC2	
10	0x09	TIMER2 OVF	estouro do TC2	
11	0x0A	TIMER1 CAPT	evento de captura do TC1	
12	0x0B	TIMER1 COMPA	igualdade de comparação A do TC1	
13	0x0C	TIMER1 COMPB	igualdade de comparação B do TC1	
14	0x0D	TIMER1 OVF	estouro do TC1	
15	0x0E	TIMER0 COMPA	igualdade de comparação A do TC0	
16	0x0F	TIMER0 COMPB	igualdade de comparação B do TC0	
17	0x10	TIMER0 OVF	estouro do TC0	
18	0x11	SPI, STC	transferência serial completa - SPI	
19	0x12	USART, RX	USART, recepção completa	
20	0x13	USART, UDRE	USART, limpeza do registrador de dados	
21	0x14	USART, TX	USART, transmissão completa	
22	0x15	ADC	conversão do ADC completa	
23	0x16	EE_RDY	EEPROM pronta	
24	0x17	ANA_COMP	comparador analógico	
25	0x18	TWI	interface serial TWI – I2C	
26	0x19	SPM_RDY	armazenagem na memória de programa pronta	

Quando o microcontrolador é inicializado, o contador de programa começa com o valor zero (endereço de *reset*). Se for empregado um programa de *boot loader*, o contador de programa é inicializado para o acesso à área da memória dedicada a ele. Então, somente depois do *boot loader* ser executado, inicia-se a execução do programa principal.

Toda vez que uma interrupção ocorre, a execução do programa é interrompida: a CPU completa a instrução em andamento, carrega na pilha o endereço da próxima instrução que seria executada (endereço de retorno) e desvia para a posição de memória correspondente à interrupção. O código escrito no endereço da interrupção é executado até o programa encontrar o código RETI (*Return from Interruption*). Então, é carregado no PC o endereço de retorno armazenado na pilha e o programa volta a trabalhar a partir do ponto que parou antes da ocorrência da interrupção.

Ao atender uma interrupção, o microcontrolador desabilita todas as outras interrupções que possam estar habilitadas, zerando o bit I do SREG (chave geral das interrupções). Ao retornar da interrupção (encontrar a instrução RETI), ele coloca novamente o bit I em 1, permitindo que outras interrupções sejam atendidas. Se uma ou mais interrupções ocorrerem neste período, os seus bits de sinalização serão colocados em 1 e o microcontrolador as tratará de acordo com a ordem de prioridade de cada uma. O detalhe é que o AVR executará sempre uma instrução do programa principal antes de atender qualquer interrupção em espera.

O AVR possui dois tipos de interrupção. O primeiro ativa um bit de sinalização (*flag*) indicando que a interrupção ocorreu, o qual é mantido em 1 até que a interrupção seja atendida, sendo zerado automaticamente pelo hardware. Alternativamente, o *flag* pode ser limpo pela escrita de 1 no bit correspondente. O bit de sinalização permite que várias interrupções fiquem ativas enquanto uma está sendo atendida e sejam processadas por ordem de prioridade. O segundo tipo de interrupção é disparada quando o evento que a gera está presente, não existem bits de sinalização e a interrupção só é atendida se sua condição existir quando a chave geral das

interrupções estiver ativa. Nesse caso, não há fila de espera, pois não há sinalização da ocorrência da interrupção. Este é o caso das interrupção externas por nível, por exemplo.

Quando uma interrupção é executada o registrador de estado (SREG) não é automaticamente preservado nem restaurado ao término de sua execução. Assim, em *assembly*, o registrador SREG e outros empregados no programa principal, e também em sub-rotinas (interrupção ou não), devem ter seus conteúdos salvos antes da execução da sub-rotina e restaurados ao término dessa. Caso contrário, o programa pode apresentar respostas imprevisíveis, visto que os valores dos registradores podem ter sido alterados fora da sequência normal do programa. Os comandos PUSH e POP são perfeitos para salvar e restaurar valores de registradores. Em C, o salvamento dos registradores é feito automaticamente pelo compilador de forma transparente ao programador.

Como as interrupções possuem endereços próximos, cada interrupção deve apresentar um comando de desvio para outra posição da memória (RJMP desvio) onde o código para o seu tratamento possa ser escrito. Caso contrário, quando ocorrer a interrupção o código executado não corresponderá ao desejado. Só existe a necessidade do desvio se a interrupção for utilizada.

Um exemplo de programação *assembly* com os endereços das interrupções é dada a seguir.

Endereço	Código	Comentários
	.ORG 0x00	;diretiva do assembly para gravar ;o código abaixo no end. 0x00
0x00	RJMP INICIO	;desvia para o início do programa
0x01	RJMP EXT_INT0	;interrup. externa 0
0x02	RJMP EXT_INT1	;interrup. externa 1
0x03	RJMP PCINT0	;interrup. 0 por mudança de pino
0x04	RJMP PCINT1	;interrup. 1 por mudança de pino
0x05	RJMP PCINT2	;interrup. 2 por mudança de pino
0x06	RJMP WDT	;estouro do temporizador Watchdog
0x07	RJMP TIM2_COMPA	;igualdade de comparação A do TC2
0x08	RJMP TIM2_COMPB	;igualdade de comparação B do TC2
0x09	RJMP TIM2_OVF	;estouro do TC2
0x0A	RJMP TIM1_CAPT	;evento de captura do TC1
0x0B	RJMP TIM1_COMPA	;igualdade de comparação A do TC1

```

0x0C      RJMP TIM1_COMPB      ;igualdade de comparação B do TC1
0x0D      RJMP TIM1_OVF       ;estouro do TC1
0x0E      RJMP TIM0_COMPB     ;igualdade de comparação A do TC0
0x0F      RJMP TIM0_COMPB     ;igualdade de comparação B do TC0
0x10      RJMP TIM0_OVF       ;estouro do TC0
0x11      RJMP SPI_STC        ;transferência serial completa - SPI
0x12      RJMP USART_RX       ;USART, recepção completa
0x13      RJMP USART_UDRE     ;USART, limpeza do registr. de dados
0x14      RJMP USART_TX       ;USART, transmissão completa
0x15      RJMP ADC            ;conversão do ADC completa
0x16      RJMP EE_RDY         ;EEPROM pronta
0x17      RJMP ANA_COMP       ;comparador analógico
0x18      RJMP TWI            ;interface serial TWI
0x19      RJMP SPM_RDY        ;memór. de armazen. de programa pronta
...
INICIO:    ... ;código para o início, programa principal
EXT_INT0:  ... ;código para a interrup. externa 0
...

```

No compilador AVR-GCC, a tabela de vetores de interrupção é pré-definida para apontar para rotinas de interrupção com nomes pré-determinados. Usando o nome apropriado, a rotina será chamada quando ocorrer a interrupção correspondente. O AVR-GCC emprega os nomes abaixo para os tratadores de interrupção. As interrupções são escritas como funções e podem estar antes ou depois do programa principal (**main()**).

Código (função)

```

int main() { //aqui vai o programa principal
ISR(INT0_vect) { //interrupção externa 0
ISR(INT1_vect) { //interrupção externa 1
ISR(PCINT0_vect) { //interrupção 0 por mudança de pino
ISR(PCINT1_vect) { //interrupção 1 por mudança de pino
ISR(PCINT2_vect) { //interrupção 2 por mudança de pino
ISR(WDT_vect) { //estouro do temporizador Watchdog
ISR(TIMER2_COMPA_vect) { //igualdade de comparação A do TC2
ISR(TIMER2_COMPB_vect) { //igualdade de comparação B do TC2
ISR(TIMER2_OVF_vect) { //estouro do TC2}
ISR(TIMER1_CAPT_vect) { //evento de captura do TC1
ISR(TIMER1_COMPA_vect) { //igualdade de comparação A do TC1
ISR(TIMER1_COMPB_vect) { //igualdade de comparação B do TC1
ISR(TIMER1_OVF_vect) { //estouro do TC1
ISR(TIMER0_COMPA_vect) { //igualdade de comparação A do TC0
ISR(TIMER0_COMPB_vect) { //igualdade de comparação B do TC0
ISR(TIMER0_OVF_vect) { //estouro do TC0
ISR(SPI_STC_vect) { //transferência serial completa - SPI
ISR(USART_RX_vect) { //USART, recepção completa
ISR(USART_UDRE_vect) { //USART, limpeza do registrador de dados
ISR(USART_TX_vect) { //USART, transmissão completa
ISR(ADC_vect) { //conversão do ADC completa
ISR(EE_READY_vect) { //EEPROM pronta
ISR(ANALOG_COMP_vect) { //comparador analógico
ISR(TWI_vect) { //interface serial TWI
ISR(SPM_READY_vect) { //armazenagem na memória de programa pronta}

```

Existem registradores específicos e geral de controle das interrupções. Cada interrupção é habilitada por um bit específico no seu registrador de controle e a interrupção é efetivamente ativada quando o bit I do registrador SREG for colocado em 1. Na fig. 6.1, são ilustradas as chaves de habilitação das interrupções. A chave geral é utilizada para ligar ou desligar todas as interrupções ativas de uma única vez. No AVR-GCC, a função **sei()** liga a chave geral das interrupções e a função **clic()** a desliga. Os bits individuais das interrupções serão vistos em momentos oportunos.

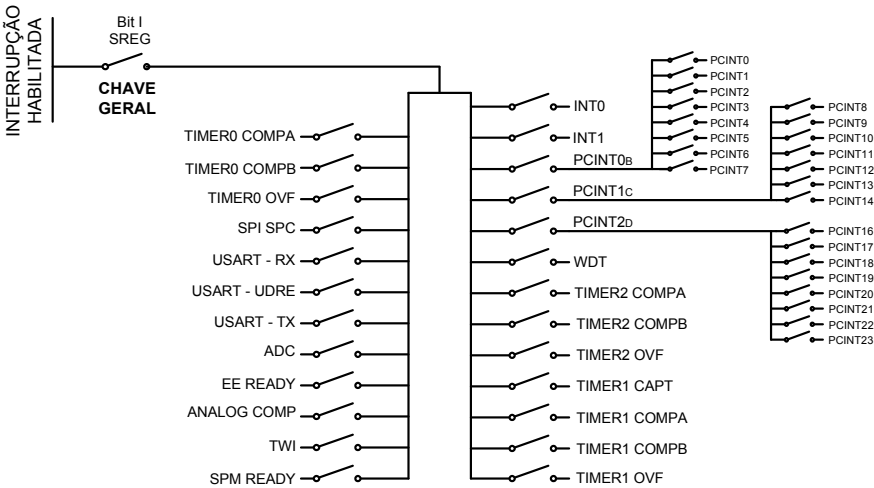


Fig. 6.1 – Chaves de habilitação das interrupções.

6.2 INTERRUPÇÕES EXTERNAS

As interrupções externas são empregadas para avisar o microcontrolador que algum evento externo a ele ocorreu, o qual pode ser um botão pressionado ou um sinal de outro sistema digital. Muitas vezes, elas são empregadas para retirar a CPU de um estado de economia de energia, ou seja, para ‘despertar’ o microcontrolador.

Todos os pinos de I/O do ATmega328 podem gerar interrupções externas por mudança de estado lógico no pino; o nome delas nos pinos são PCINT0 até PCINT23. Entretanto, existem dois pinos, INT0 e INT1, que podem gerar interrupções na borda de subida, descida ou na manutenção do nível do estado lógico no pino. Se habilitada, a interrupção pode ocorrer mesmo que o pino esteja configurado como saída, permitindo gerar interrupção por software ao se escrever no pino.

Cada uma das interrupções para os pinos INT0 e INT1 possuem um endereço fixo na memória de programa; já as interrupções PCINTx possuem somente 3 endereços, um para cada PORT. Caso mais de um pino no PORT esteja habilitado para gerar a interrupção, é necessário, via programação, testar qual pino gerou a interrupção.

O registrador de controle EICRA (*Extern Interrupt Control Register A*) contém os bits responsáveis pela configuração das interrupções externas INT0 e INT1, conforme tab. 6.2.

Bit	7	6	5	4	3	2	1	0
EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00
Lê/Escreve	L	L	L	L	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Tab. 6.2 – Bits de configuração da forma das interrupções nos pinos INT1 e INT0.

ISC11	ISC10	Descrição
0	0	Um nível baixo em INT1 gera um pedido de interrupção.
0	1	Qualquer mudança lógica em INT1 gera um pedido de interrupção.
1	0	Uma borda de decida em INT1 gera um pedido de interrupção.
1	1	Uma borda de subida em INT1 gera um pedido de interrupção.
ISC01	ISC00	Descrição
0	0	Um nível baixo em INT0 gera um pedido de interrupção.
0	1	Qualquer mudança lógica em INT0 gera um pedido de interrupção.
1	0	Uma borda de decida em INT0 gera um pedido de interrupção.
1	1	Uma borda de subida em INT0 gera um pedido de interrupção.

As interrupções são habilitadas nos bits INT1 e INT0 do registrador EIMSK (*External Interrupt Mask Register*) se o bit I do registrador SREG (*Status Register*) também estiver habilitado.

Bit	7	6	5	4	3	2	1	0
EIMSK	-	-	-	-	-	-	INT1	INT0
Lê/Escreve	L	L	L	L	L	L	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

O registrador EIFR (*External Interrupt Flag Register*) contém os dois bits sinalizadores que indicam se alguma interrupção externa ocorreu. São limpos automaticamente pela CPU após a interrupção ser atendida. Alternativamente, esses bits podem ser limpos pela escrita de 1 lógico.

Bit	7	6	5	4	3	2	1	0
EIFR	-	-	-	-	-	-	INTF1	INTF0
Lê/Escreve	L	L	L	L	L	L	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

O registrador PCICR (*Pin Change Interrupt Control Register*) é responsável pela habilitação de todas as interrupções externas nos pino de I/O, do PCINT0 até o PCINT23, as quais são divididas entre os três PORTs do microcontrolador:

Bit	7	6	5	4	3	2	1	0
PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0
Lê/Escreve	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

onde PCIE0 habilita a interrupção por qualquer mudança nos pinos do PORTB (PCINT0:7), PCIE1 nos pinos do PORTC (PCINT8:14) e PCIE2 nos pinos do PORTD (PCINT16:23).

Quando ocorrer um interrupção em algum dos pinos, o bit de sinalização do PORT no registrador PCIFR é ativo: PCIF0 para o PORTB, PCIF1 para o PORTC e PCIF2 para o PORTD.

Bit	7	6	5	4	3	2	1	0
PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0
Lê/Escreve	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Com pode ser observado na fig. 6.1, existem várias chaves ligadas às chaves de interrupção dos PORTs, PCINT0_B, PCINT1_C e PCINT2_D, as quais indicam que cada pino de I/O pode ser individualmente habilitado para gerar uma interrupção externa. É importante observar que a Atmel usa os mesmos nomes PCINT0, PCINT1 e PCINT2 para nomear as interrupções dos PORTs e, ainda, para três pinos, gerando confusão. Desta forma, os pinos PCINT0, 1 e 2 estão ligados ao PORTB e pertencem somente à interrupção PCINT0. A habilitação individual dos pinos é feita nos registradores PCMSK0:2.

Bit	7	6	5	4	3	2	1	0
PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Lê/Escreve	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit	7	6	5	4	3	2	1	0
PCMSK1	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
Lê/Escreve	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit	7	6	5	4	3	2	1	0
PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
Lê/Escreve	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Para ilustrar a operação das duas interrupções externas, INT0 e INT1, configuradas, respectivamente, para gerar interrupção por nível e por transição, é apresentado na fig. 6.2 um circuito exemplo. Um botão irá trocar o estado do LED (se ligado, desliga e vice-versa), o outro botão mantido pressionado piscará o LED. O código exemplo é apresentado em seguida.

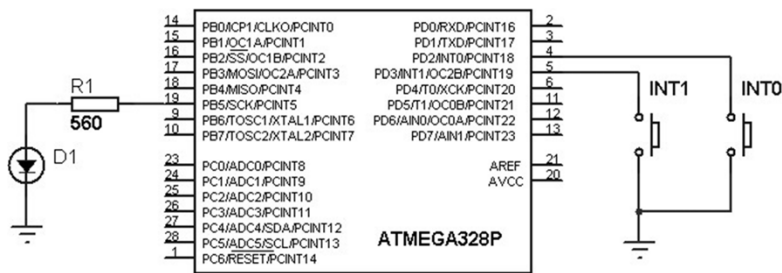


Fig. 6.2 – Circuito para emprego das interrupções externas INT0 e INT1.

INT0_1.c

```
//===== //
// HABILITANDO AS INTERRUPTÕES INT0 e INT1 POR TRANSIÇÃO E NÍVEL, RESPECTIVAMENTE //
//===== //
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

//Definições de macros - empregadas para o trabalho com bits
#define set_bit(Y,bit_x) (Y|=(1<<bit_x)) //ativa o bit x da variável
#define clr_bit(Y,bit_x) (Y&=~(1<<bit_x)) //limpa o bit x da variável
#define tst_bit(Y,bit_x) (Y&(1<<bit_x)) //testa o bit x da variável
#define cpl_bit(Y,bit_x) (Y^=(1<<bit_x)) //troca o estado do bit x

#define LED PB5 //LED está no pino PB5
```

```

ISR(INT0_vect);
ISR(INT1_vect);

//-----
int main()
{
    DDRD = 0x00;          //PORTD entrada
    PORTD = 0xFF;         //pull-ups habilitados
    DDRB = 0b00100000;    //somente pino do LED como saída
    PORTB = 0b11011111;   //desliga LED e habilita pull-ups

    UCSRB = 0x00;         /*necessário desabilitar RX e TX para trabalho com os pinos
                           do PORTD no Arduino*/

    EICRA = 1<<ISC01;     //interrupções externas: INT0 na borda de descida, INT1 no nível zero.
    EIMSK = (1<<INT1) | (1<<INT0); //habilita as duas interrupções
    sei();                 //habilita interrupções globais, ativando o bit I do SREG

    while(1){}
}
//-----
ISR(INT0_vect) //interrupção externa 0, quando o botão é pressionado o LED troca de estado
{
    cpl_bit(PORTB,LED);
}
//-----
ISR(INT1_vect) //interrupção externa 1, mantendo o botão pressionado o LED pisca
{
    cpl_bit(PORTB,LED);
    _delay_ms(200);       //tempo para piscar o LED
}
//=====

```

No programa acima, quando o botão ligado a INT0 é pressionado, o LED troca de estado (liga ou desliga). A interrupção foi habilitada para ocorrer somente na transição de 1 para 0 e, como não existe tratamento para o ruído gerado no acionamento do botão, a resposta pode não ser a desejada. Esse exemplo mostra que o uso de um botão com interrupção pode não ser adequado para algumas aplicações. Por sua vez, o botão ligado a interrupção INT1 foi habilitado para gerar uma interrupção por nível lógico 0, ou seja, enquanto o botão estiver pressionado a interrupção irá ocorrer e a rotina de tratamento da interrupção continuará sendo executada. Dessa forma, como existe um atraso de 200 ms dentro da interrupção, mantendo-se o botão pressionado, o LED irá piscar e não existe prejuízo devido ao ruído do botão.

Ao se utilizar rotinas de interrupção, deve-se evitar códigos extensos, pois o programa principal (**int main()**) fica suspenso até o término da execução da rotina. O programador deve avaliar o impacto do tempo gasto

dentro da rotina de interrupção em relação ao restante do programa. Outro ponto importante a se observar é evitar, sempre que possível, chamar outras funções dentro da interrupção, pois isso degrada o desempenho do programa e pode gerar problemas difíceis de detectar.

O circuito da fig. 6.3 e o seu respectivo programa de teste são empregados para exemplificar o uso das interrupções externas por mudança em qualquer pino de I/O. Quando um botão do PORTC é pressionado, o LED definido para o pino, liga ou desliga. Como as interrupções são por mudança de estado no pino, é gerado um pedido de interrupção sempre que se pressiona e se solta o botão. Existindo ruído no acionamento do botão, o resultado será imprevisível. Assim, foi colocado um atraso de 200 ms no final da rotina de interrupção para que o botão seja pressionado rapidamente, ativando a interrupção somente na borda de descida do sinal; caso contrário, o estado do LED seria aleatório. Como a interrupção externa PCINT1 só possui um endereço na memória de programa, é necessário testar qual foi o pino responsável pela interrupção.

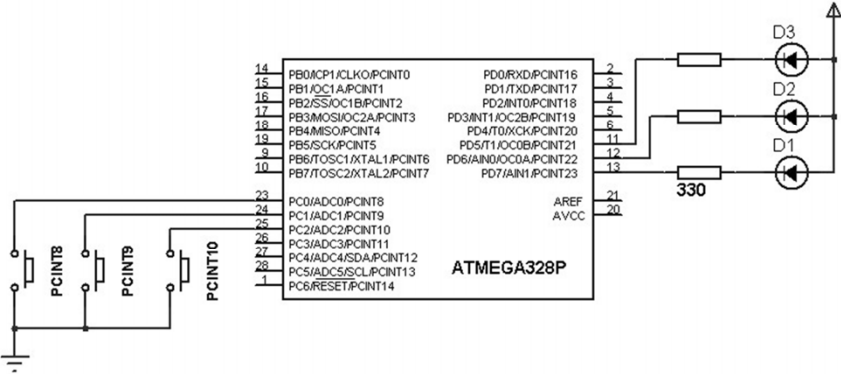


Fig. 6.3 – Circuito para teste das interrupções externas PCINT8:10.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL //frequência de trabalho
#include <avr/io.h> //definições do componente especificado
#include <avr/interrupt.h> //define algumas macros para as interrupções
#include <util/delay.h> //biblioteca para o uso das rotinas de delay

//Definições de macros para o trabalho com bits
#define set_bit(y,bit) (y|=(1<<bit))//coloca em 1 o bit x da variável Y
#define clr_bit(y,bit) (y&=~(1<<bit))//coloca em 0 o bit x da variável Y
#define cpl_bit(y,bit) (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define tst_bit(y,bit) (y&(1<<bit))//retorna 0 ou 1 conforme leitura do bit

#define LED0 PD5
#define LED1 PD6
#define LED2 PD7

#endif
```

PCINTx.c (programa principal)

```
//===== //
// Cada vez que um botão é pressionado o LED correspondente troca de estado //
//===== //
#include "def_principais.h"//inclui arquivo com as definições principais

ISR(PCINT1_vect);

int main()
{
    DDRC = 0x00; //PORTC como entrada, 3 botões
    PORTC = 0xFF; //habilita pull-ups
    DDRD = 0b11100000; //pinos PD5:7 do PORTC como saída (leds)
    PORTD = 0xFF; //apaga leds e habilita pull-ups dos pinos não utilizados

    PCICR = 1<<PCIE1;//habilita interrupção por qualquer mudança de sinal no PORTC
    PCMSK1 = (1<<PCINT10)|(1<<PCINT9)|(1<<PCINT8);/*habilita os pinos PCINT8:10 para
                                                    gerar interrupção*/

    sei(); //habilita as interrupções

    while(1){}
}
//-----
ISR(PCINT1_vect)
{
    //quando houver mais de um pino que possa gerar a interrupção é necessário testar qual foi
    if(!tst_bit(PINC,PC0))
        cpl_bit(PORTD,LED0);
    else if(!tst_bit(PINC,PC1))
        cpl_bit(PORTD,LED1);
    else if(!tst_bit(PINC,PC2))
        cpl_bit(PORTD,LED2);

    _delay_ms(200);
}
//=====
```

6.2.1 UMA INTERRUPÇÃO INTERROMPENDO OUTRA

O ATmega não suporta interrupções aninhadas, isto é, uma interrupção não pode interromper outra em andamento, mesmo que tenha maior prioridade. Ocorrendo uma ou mais interrupções enquanto uma está sendo tratada, é formado uma fila de espera que é atendida pela ordem de prioridade. Dessa forma, ao tratar uma interrupção, a CPU automaticamente desabilita todas as interrupções, voltando a ligá-las ao final da rotina de interrupção.

Para fazer o aninhamento de interrupções é necessário, via programação, habilitar a chave geral das interrupções, o bit I do registrador SREG, dentro da interrupção que se deseja interromper. Nesse caso, deve-se ter cuidado em preservar o registrador de *status* – SREG e analisar a prioridade das interrupções para que o programa não gere respostas inesperadas. A seguir, é apresentado um exemplo que ilustra esta ideia: uma interrupção de menor prioridade interrompendo uma de maior, INT1 versus INT0, ambas habilitadas para gerar a interrupção por nível zero. O arquivo **def_principais.h** é o mesmo apresentado anteriormente. Na fig. 6.4 pode-se ver o circuito de teste montado para o Arduino.

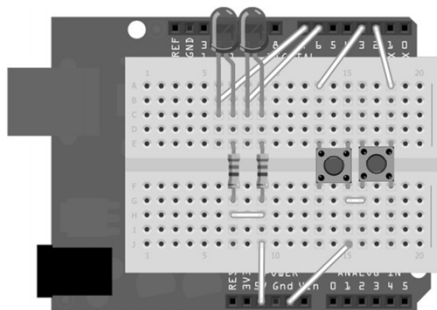


Fig. 6.4 – Circuito montado no Arduino para teste das interrupções INT0 e INT1.

INTO_1_aninhada.c

```
//===== //
//  INTERRUPÇÃO INT1 INTERROMPENDO A INTO  //
//===== //
#include "def_principais.h"

ISR(INT0_vect);
ISR(INT1_vect);

//-----
int main()
{
    DDRC = 0b11000000; //configurando os pinos de entrada e saída
    PORTD = 0b11111111; //desligando leds e habilitando pull-ups

    UCSRB = 0x00; //desabilitando RX e TX para trabalho com os pinos do Arduino

    EICRA = 0x00; //interrupções externas INT0 e INT1 no nível zero.
    EIMSK = (1<<INT1)|(1<<INT0); //habilita as duas interrupções
    sei(); //habilita interrupções globais, ativando o bit I do SREG

    while(1) //pisca led numa velocidade muito grande (visualmente fica ligado)
        cpl_bit(PORTD, LED2);
}
//-----
ISR(INT0_vect) //interrupção externa 0, quando o botão é pressionado o LED pisca
{
    unsigned char sreg;

    sreg = SREG; //salva SREG porque a interrupção pode alterar o seu valor

    clr_bit(EIMSK, INT0); //desabilita INTO para que ele não chame a si mesmo
    sei(); //habilita a interrupção geral, agora INT1 pode interromper INTO

    cpl_bit(PORTD, LED1); //pisca led a cada 300 ms
    _delay_ms(300);

    set_bit(EIMSK, INT0); //habilita novamente a interrupção INTO

    SREG = sreg; //restaura o valor de SREG que pode ter sido alterado
}
//-----
ISR(INT1_vect) //interrupção externa 1, mantendo o botão pressionado o LED pisca
{
    cpl_bit(PORTD, LED1); //pisca led a cada 200 ms
    _delay_ms(200);
}
//=====
```

Quando INTO está ativa, INT1 consegue interrompê-la, isso pode ser verificado pelo aumento na frequência que o LED1 pisca. Dentro da rotina de interrupção INTO alguns passos são necessários: primeiro, salvar o conteúdo do SREG, pois o compilador não sabe que pode haver outra interrupção que altere o seu valor; segundo, desabilitar a própria interrupção INTO, pois ela é prioritária e estando habilitada por nível vai gerar um pedido de interrupção ainda dentro da sua própria rotina,

entrando para a fila de prioridade de execução; terceiro, executar o que se deseja como ação para a interrupção; quarto, habilitar novamente a interrupção INT0 e; quinto, restaurar o valor original do SREG.

No código de atendimento de INT1 é necessário somente as instruções para a ação desejada, não é necessário nenhum cuidado especial com o SREG porque o compilador AVR-GCC automaticamente salva o seu conteúdo na entrada de uma interrupção e o retorna na saída.

Exercícios:

6.1 – Supondo que sejam habilitadas 3 interrupções: INT0, INT1 e PCINT0, e que INT0 e PCINT0 sempre estão ativas ao termino da interrupção INT1, quando a interrupção PCINT0 será executada?

6.2 – Teste o exemplo da interrupção INT1 interrompendo a INT0 (seção 6.2.1). Por que se consegue ver que o LED2 do laço principal está piscando quando um dos botões é mantido pressionado? Por que a interrupção INT0 não consegue interromper a INT1?
Exclua o trecho de código dentro da interrupção INT0 que salva o SREG e o restaura. O que acontece?

6.3 – Faça um programa para testar a interrupção PCINT2, usando dois botões nos pinos PD6 e PD7, onde cada um deve ligar um LED nos pinos PD0 e PD1.
