



# OS PODEROSOS $\mu$ CONTROLADORES AVR



Prof. Charles Borges de Lima

Copyright© junho de 2009 para o  
Departamento Acadêmico de Eletrônica  
Av. Mauro Ramos, 950  
CEP 88020-300  
Florianópolis, SC – Brasil.

Este material pode ser empregado  
livremente, desde que citada a fonte.  
É de uso exclusivamente didático e sem  
fins comerciais.

## PREFÁCIO

O trabalho com  $\mu$ controladores é algo apaixonante para os projetistas de eletrônica digital. Os últimos avanços aumentaram a facilidade de uso desse tipo de componente. A internet está repleta de informações e códigos. Entretanto, um documento, com as principais questões de programação e hardware, resumido em um único compêndio, é difícil de encontrar. Este é o caso da literatura sobre o AVR em língua portuguesa. Assim, este trabalho tenta suprir esta lacuna.

Estudar o ATmega8 implica em direta aplicação dos conhecimentos aos demais  $\mu$ controladores da família AVR. Ele foi escolhido por conter as principais características desejáveis em um  $\mu$ controlador.

Ao longo deste trabalho foram mantidas várias designações da língua inglesa porque a tradução não ficaria elegante e também para colocar o estudante a par dos termos mais comuns empregados. Termos “abrasileirados” foram empregados pela falta de melhor tradução.

É fundamental a consulta ao manual do fabricante, pois, todos os detalhes estão lá. Aqui somente temos um resumo e algumas aplicações típicas, nada mais que isto. Importante notar que o fabricante disponibiliza vários *Application Notes* com muitas dicas, bem como códigos prontos para serem utilizados.

Se todas as atividades propostas ao longo deste trabalho forem realizadas, pode-se dizer que o estudante adquiriu aptidão suficiente para projetos com o AVR.

A complexidade e o conhecimento são gradualmente aumentados ao longo do trabalho. É importante seguir os capítulos ordenadamente.

Boa Sorte!

O autor.

“Talvez seja este o aprendizado mais difícil: manter o movimento permanente, a renovação constante, a vida vivida como caminho e mudança.”

*Maria Helena Kuhner.*

## ÍNDICE

<b>1. INTRODUÇÃO.....</b>	<b>3</b>
1.1 OS PODEROSOS $\mu$ CONTROLADORES AVR .....	5
1.2 A FAMÍLIA AVR.....	6
<b>2. O ATMEGA .....</b>	<b>8</b>
2.1 AS MEMÓRIAS .....	12
2.1.1 O <i>STACK POINTER</i> .....	13
2.2 DESCRIÇÃO DOS PINOS .....	15
2.3 SISTEMA DE CLOCK.....	16
2.4 O RESET.....	17
2.5 GERENCIAMENTO DE ENERGIA E O MODO SLEEP .....	18
<b>3. COMEÇANDO O TRABALHO .....</b>	<b>19</b>
3.1 CRIANDO UM PROJETO NO AVR STUDIO .....	19
3.2 SIMULANDO NO PROTEUS (ISIS) .....	21
<b>4. PORTAS DE ENTRADA E SAÍDA (I/Os).....</b>	<b>24</b>
4.1 LENDO UM BOTÃO E LIGANDO UM LED .....	25
4.2 ACIONANDO DISPLAYs DE 7 SEGMENTOS .....	30
4.3 ACIONANDO LCDs 16x2 .....	33
<b>5. INTERRUPÇÕES.....</b>	<b>40</b>
5.1 INTERRUPÇÕES EXTERNAS .....	41
<b>6. GRAVANDO A EEPROM .....</b>	<b>44</b>
<b>7. TECLADO MATRICIAL.....</b>	<b>46</b>
<b>8. TEMPORIZADORES/CONTADORES.....</b>	<b>49</b>
8.1 TEMPORIZADOR/CONTADOR 0 .....	51
8.2 TEMPORIZADOR/CONTADOR 2 .....	52
8.3 TEMPORIZADOR/CONTADOR 1 .....	58

<b>9. MULTIPLEXAÇÃO (VARREDURA DE DISPLAYs) .....</b>	<b>67</b>
<b>10. DISPLAY GRÁFICO (128x64 pontos).....</b>	<b>69</b>
<b>11. GERANDO FORMAS DE ONDA .....</b>	<b>76</b>
11.1 DISCRETIZANDO UM SINAL .....	76
11.2 MODULAÇÃO POR LARGURA DE PULSO – PWM .....	78
<b>12. SPI.....</b>	<b>80</b>
12.1 GRAVAÇÃO IN-SYSTEM DO ATMEGA .....	84
<b>13. USART.....</b>	<b>86</b>
<b>14. TWI (TWO WIRE SERIAL INTERFACE) – I2C.....</b>	<b>94</b>
14.1 REGISTRADORES DO TWI .....	98
14.2 USANDO O TWI.....	101
14.3 I2C VIA SOFTWARE SOMENTE .....	106
<b>15. COMUNICAÇÃO 1 FIO (VIA SOFTWARE).....</b>	<b>108</b>
<b>16. COMPARADOR ANALÓGICO .....</b>	<b>111</b>
16.1 MEDINDO RESISTÊNCIA E CAPACITÂNCIA .....	113
<b>17. CONVERSOR ANALÓGICO-DIGITAL .....</b>	<b>116</b>
<b>18. GRAVANDO O ATMEGA8.....</b>	<b>124</b>
<b>19. CONCLUSÕES.....</b>	<b>129</b>
<b>20. BIBLIOGRAFIA .....</b>	<b>130</b>
<b>ANEXOS .....</b>	<b>131</b>

## 1. INTRODUÇÃO

Os avanços tecnológicos demandam cada vez mais dispositivos eletrônicos. Assim, a cada dia são criados componentes mais versáteis e poderosos. Nesta categoria, os  $\mu$ controladores têm alcançado grande desenvolvimento. Sua facilidade de uso em ampla faixa de aplicações permite o projeto relativamente rápido e fácil de novos equipamentos.

O  $\mu$ controlador é o agrupamento de vários componentes em um sistema  $\mu$ processado. Basicamente o  $\mu$ controlador é um  $\mu$ processador com memória RAM e de programa, temporizadores e circuitos de *clock* embutidos. O único componente externo que pode ser necessário é um cristal para determinar a frequência de trabalho.

Os  $\mu$ controladores têm agregado inúmeras funcionalidades, tais como: gerador interno independente de *clock*; memória SRAM, EEPROM e FLASH; conversores A/D, D/A; vários temporizadores/contadores; comparadores analógicos; PWM; diferentes tipos de interface de comunicação, incluindo USB, UART, I2C, CAN, SPI, JTAG; relógios de tempo real; circuitos para gerenciamento de energia no chip; circuitos para controle de *reset*, alguns tipos de sensores; interface para LCD; e outras funcionalidades de acordo com o fabricante. Na Tab. 1.1 é apresentada uma lista das várias famílias dos principais fabricantes. A coluna Núcleo indica o tipo de arquitetura ou unidade de processamento que constitui a base do  $\mu$ controlador, a coluna IDE lista o nome dos ambientes de desenvolvimento que podem ser baixados do sítio da internet de cada fabricante.

Existem duas arquiteturas clássicas para os  $\mu$ controladores em geral: a arquitetura Von-Neumann, onde existe apenas um barramento interno por onde circulam instruções e dados e a arquitetura Harvard, que é caracterizada por dois barramentos internos, sendo um de instruções e outro de dados (Fig. 1.1). Pode-se dizer que a primeira é uma arquitetura serial e a segunda paralela; da mesma forma, pode-se dizer que a arquitetura Von-Neumann permite produzir um conjunto complexo de código de instruções para o processador (CISC – *Complex Instructions Set Computer*), com um tempo de execução por instrução de vários ciclos de *clock*. Já a arquitetura Harvard produz um conjunto simples de códigos de instruções e, dado ao paralelismo de sua estrutura, é capaz de executar uma instrução por ciclo de *clock*. A arquitetura Von-Neumann é mais simples, com menor número de portas lógicas, entretanto, sua velocidade é menor que a Harvard. A arquitetura Harvard necessita de mais linhas de código para executar a mesma tarefa que uma arquitetura Von-Neumann, a qual possui muito mais tipos de instruções.

Tab. 1.1 – Principais fabricantes de  $\mu$ controladores (fonte: revista Elektor 02/2006).

Fabricante	Internet	Barramento	Família	Arquitetura	Núcleo	IDE
<b>Analog Device</b>	www.analog.com	8-bits 32-bits	ADUC8xx ADUC7xx	CISC RISC	8051 ARM7	- - independent programs
<b>Atmel</b>	www.atmel.com	8-bits 32-bits 32-bits 32-bits	AT89xxx AVR AVR32 AT91xxx	RISC RISC RISC RISC	8051 - - ARM7/9	- - - AVR studio
<b>Cirrus Logic</b>	www.cirrus.com	32-bits 32-bits	EP73xxx EP93xxx	RISC RISC	ARM7 ARM9	- -
<b>Cygnal</b>	www.silabs.com	8-bits	C8051F	CISC	8051	-
<b>Freescape (ex. Motorola)</b>	www.freescape.com	8-bits 8-bits 8-bits 8-bits 16-bits 16-bits 16-bits 32-bits 32-bits 32-bits	HC05 HC08 HC11 HC12 HC512 HC16 56800 68K ColdFire MAC7100	CISC CISC CISC CISC CISC CISC CISC CISC CISC RISC	6800 6809 6809 - - - - 68000 - - ARM7	- Code Warrior - - Code Warrior - - - - - -
<b>Fujitsu</b>	www.fujitsu.com	8-bits 16-bits 32-bits	F2MC-8 F2MC-16 FR	CISC CISC RISC	- - -	- - -
<b>Infineon</b>	www.infineon.com	8-bits 16-bits 16-bits 32-bits	C5xxx, C8xxx C16xxx XC16xxx TCxxx	CISC CISC CISC CISC	8051 - - -	- - - -
<b>Intel</b>	www.intel.com	8-bits 16-bits 8-bits 8-bits 8-bits 16-bits	MC5251 MC596/296 DS80Cxxx DS83Cxxx DS89Cxxx MAXQ	CISC CISC CISC CISC CISC RISC	8051 - 8051 8051 8051 -	- - - - - MPLAB
<b>Maxim (Dallas)</b>	www.maxim-ic.com	8-bits 16-bits	PIC-10,12,16,18 dsPIC	RISC RISC	- -	MPLAB MPLAB
<b>Microchip</b>	www.microchip.com	16-bits		RISC	-	
<b>NS</b>	www.national.com	8-bits 16-bits 16-bits	COP8xxx CR16Cxxx CP3000	CISC CISC RISC	- - -	- - -
<b>Philips</b>	www.semiconductors.philips.com	8-bits 16-bits 32-bits	P8xxx Xxxx LPC2000	CISC CISC RISC	8051 - ARM7	- - -
<b>Rabbit Semiconductor</b>	www.rabbitsemi-conductor.com	8-bits 8-bits	Rabbit2000 Rabbit3000	CISC CISC	- -	- -
<b>Renesas</b>	www.renesas.com	8-bits 16-bits 16-bits 16-bits 16-bits 32-bits 32-bits	740 H8 H8S M16C 7700 H8SX Super H	CISC CISC CISC CISC CISC CISC CISC	- - - - - - -	- HEW HEW - - - - HEW
<b>ST</b>	www.stm.com	8-bits 8-bits 8-bits 8-bits 16-bits 16-bits 32-bits	ST5 ST6 ST7 ST9 ST9 ST10 ARM7 MSC12xxx MSP430 TMS470	CISC CISC CISC CISC CISC CISC RISC CISC CISC RISC	- - - - - - ARM7 8051 - ARM7	- Visual FIVE - STWD-7 STWD-9 STWD-9 - - -
<b>Texas Instruments</b>	www.ti.com	8-bits 16-bits 32-bits	870 900/900H 900/900H	CISC CISC CISC	- - -	- - -
<b>Toshiba</b>	chips.toshiba.com	8-bits 16-bits 32-bits	5xxx Z8xxx Z8Cxxx	RISC CISC CISC	- Z80 Z80	- - -
<b>Ubicom (ex. Scentix)</b>	www.ubicom.com	8-bits 8-bits 8-bits	5xxx Z8xxx Z8Cxxx	RISC CISC CISC	- Z80 Z80	- - -
<b>Zilog</b>	www.zilog.com	8-bits	eZ8040aim	CISC	Z80	-



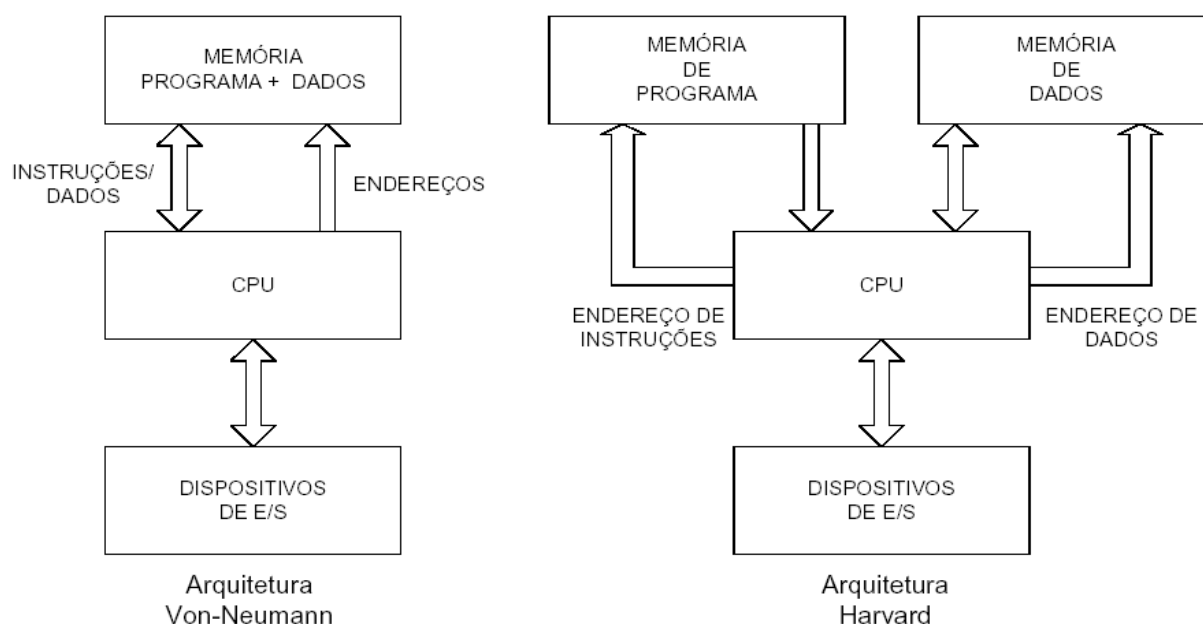


Fig. 1.1 – Arquiteturas clássicas de processadores: Von-Neumann x Harvard.

Atualmente nas modernas arquiteturas de  $\mu$ controladores está havendo o domínio da Harvard, a qual evoluiu para uma arquitetura que pode ser chamada de Harvard estendida ou avançada. Sendo composta por um grande número de instruções e ainda com a redução da quantidade necessária de portas lógicas, produzindo um núcleo de processamento compacto, veloz e com programação eficiente (menor número de linhas de código). Devido às questões de desempenho, compatibilidade eletromagnética e economia de energia, hoje é praticamente inaceitável que um  $\mu$ controlador não execute a maioria das instruções em poucos ciclos de *clock* (o que diminui o consumo e a dissipação de energia).

## 1.1 OS PODEROSOS $\mu$ CONTROLADORES AVR

Os  $\mu$ controladores AVR foram desenvolvidos na Noruega em 1995 e são produzidos pela ATMEL, apresentam ótima eficiência de processamento e núcleo compacto (poucos milhares de portas lógicas). Com uma estrutura RISC avançada, apresentam mais de uma centena de instruções e uma arquitetura voltada à programação C, a qual permite produzir códigos compactos. Também, dado sua arquitetura, o desempenho do seu núcleo de 8 bits é equivalente ao desenvolvido por  $\mu$ controladores de 16bits.

As principais características dos  $\mu$ controladores AVR são:

- Executam poderosas instruções em um simples ciclo de *clock* e operam com tensões entre 1,8 e 5,5 V, com velocidades de até 20 MHz. Sendo disponíveis em diversos encapsulamentos (de 8 até 64 pinos).
- Alta integração e grande número de periféricos com efetiva compatibilidade entre toda a família AVR.

- Possuem vários modos para redução do consumo de energia e características adicionais (*picoPower*) para sistemas críticos.
- Possuem 32 registradores de propósito geral, memória de acesso *load-store* e a maioria das instruções é de 16 bits.
- Memória de programação FLASH programável *in-system*, SRAM e EEPROM, para desenvolvimentos rápidos e flexibilidade de projeto.
- Facilmente programados e com debug *in-system* via interface simples, ou com interfaces JTAG compatível com 6 ou 10 pinos.
- Um conjunto completo e gratuito de softwares.
- Preço acessível.

Existem  $\mu$ controladores AVR específicos para diversas áreas, tais como: automotiva, controle de LCDs, redes de trabalho CAN, USB, controle de motores, controle de lâmpadas, monitoração de bateria, 802.15.4/ZigBee™ e controle por acesso remoto.

## 1.2 A FAMÍLIA AVR

Dentre os principais componentes da família AVR podemos citar:

- **tinyAVR® - ATtiny**  
 $\mu$ controladores de propósito geral de até 8 kbytes de memória Flash, 512 bytes de SRAM e EEPROM.
- **megaAVR® - ATmega**  
 $\mu$ controladores de alto desempenho com multiplicador por hardware, com até 256 kbytes de memória Flash, 4 kbytes de EEPROM e 8 kbytes de SRAM.
- **picoPower™ AVR**  
 $\mu$ controladores com características especiais para economia de energia.
- **XMEGA™ ATxmega**  
Os novos  $\mu$ controladores XMEGA 8/16-bit dispõem de novos e avançados periféricos com aumento de desempenho, DMA (*Direct Memory Access*) e sistema de eventos.
- **AVR32** (não pertence às famílias acima)  
 $\mu$ controladores de 32 bits com arquitetura RISC projetada para maior processamento por ciclos de clock, com eficiência de 1,3 mW/MHz e até 210 DMIPS (*Dhrystone Million Instructions per Second*) a 150 MHz, conjunto de instruções para DSP (*Digital Signal Processing*) com SIMD (*Single Instruction, Multiple Data*) com soluções SoC (*System-on-a-chip*) e completo suporte ao Linux.

As Tabs. 1.2 e 1.3 apresentam as principais características dos AVRs ATmega e ATtiny.

Tab. 1.2 – Comparação entre os ATmega (04/2009).

Devices/Devices	Flash (Kbytes)	EEPROM (Kbytes)	SRAM (Bytes)	Max I/O Pins	F max (MHz)	Vcc (V)	10-bit A/D Channels	Analog Comparator	16-bit Timers	8-bit Timer	Brown Out Detector	Ext Interrupts	Hardware Multiplier	Interrupts	ISP	On Chip Oscillator	PWM Channels	RTC	Self Program Memory	SPI	TWI	UART	Watchdog
<b>ATmega128</b>	128	4	4096	53	16	2,7-5,5	8	Yes	2	2	Yes	8	Yes	34	Yes	Yes	8	Yes	Yes	1	Yes	2	Yes
<b>ATmega1280</b>	128	4	8192	86	16	1,8-5,5	16	Yes	4	2	Yes	32	Yes	57	Yes	Yes	16	Yes	Yes	1+USART	Yes	4	Yes
<b>ATmega1281</b>	128	4	8192	54	16	1,8-5,5	8	Yes	4	2	Yes	17	Yes	48	Yes	Yes	9	Yes	Yes	1+USART	Yes	2	Yes
<b>ATmega1284P</b>	128	4	16K	32	20	1,8-5,5	Yes	Yes	2	2	Yes	3	Yes	35	Yes	Yes	6	Yes	Yes	Yes	Yes	2	Yes
<b>ATmega162</b>	16	0,5	1024	35	16	1,8-5,5	--	Yes	2	2	Yes	3	Yes	28	Yes	Yes	6	Yes	Yes	1	--	2	Yes
<b>ATmega164P</b>	16	0,5	1024	32	20	1,8-5,5	8	Yes	1	2	Yes	32	Yes	31	Yes	Yes	6	Yes	Yes	1+USART	Yes	2	Yes
<b>ATmega164PA</b>	16	0,5	1024	32	20	1,8-5,5	8	Yes	1	2	Yes	32	Yes	31	Yes	Yes	6	Yes	Yes	1+USART	Yes	2	Yes
<b>ATmega165P</b>	16	0,5	1024	54	16	1,8-5,5	8	Yes	1	2	Yes	17	Yes	23	Yes	Yes	4	Yes	Yes	1+USI	USI	1	Yes
<b>ATmega168</b>	16	0,5	1024	23	20	1,8-5,5	6/8	Yes	1	2	Yes	26	Yes	26	Yes	Yes	6	Yes	Yes	1+USART	Yes	1	Yes
<b>ATmega168P</b>	16	0,5	1024	23	20	1,8-5,5	8	Yes	1	2	Yes	26	Yes	26	Yes	Yes	6	Yes	Yes	1+USART	Yes	1	Yes
<b>ATmega16A</b>	16	0,5	1024	32	16	2,7-5,5	8	Yes	1	2	Yes	3	Yes	20	Yes	Yes	4	Yes	Yes	1	Yes	1	Yes
<b>ATmega2560</b>	256	4	8192	86	16	1,8-5,5	16	Yes	4	2	Yes	32	Yes	57	Yes	Yes	16	Yes	Yes	1+USART	Yes	4	Yes
<b>ATmega2561</b>	256	4	8192	54	16	1,8-5,5	8	Yes	4	2	Yes	17	Yes	48	Yes	Yes	9	Yes	Yes	1+USART	Yes	2	Yes
<b>ATmega324P</b>	32	1	2048	32	20	1,8-5,5	8	Yes	1	2	Yes	32	Yes	31	Yes	Yes	6	Yes	Yes	1+USART	Yes	2	Yes
<b>ATmega324PA</b>	32	1	2048	32	20	1,8-5,5	8	Yes	1	2	Yes	32	Yes	31	Yes	Yes	6	Yes	Yes	1+USART	Yes	2	Yes
<b>ATmega325</b>	32	1	2048	54	16	1,8-5,5	8	Yes	1	2	Yes	17	Yes	23	Yes	Yes	4	Yes	Yes	1+USI	USI	1	Yes
<b>ATmega3250</b>	32	1	2048	69	16	1,8-5,5	8	Yes	1	2	Yes	17	Yes	32	Yes	Yes	4	Yes	Yes	1+USI	USI	1	Yes
<b>ATmega3250P</b>	32	1	2048	69	20	1,8-5,5	8	Yes	1	2	Yes	17	Yes	32	Yes	Yes	4	Yes	Yes	1+USI	USI	1	Yes
<b>ATmega325P</b>	32	1	2048	54	20	1,8-5,5	8	Yes	1	2	Yes	17	Yes	23	Yes	Yes	4	Yes	Yes	1+USI	USI	1	Yes
<b>ATmega328P</b>	32	1	2048	23	20	1,8-5,5	8	Yes	1	2	Yes	26	Yes	26	Yes	Yes	6	Yes	Yes	1+USART	Yes	1	Yes
<b>ATmega32A</b>	32	1	2048	32	16	2,7-5,5	8	Yes	1	2	Yes	3	Yes	19	Yes	Yes	4	Yes	Yes	1	Yes	1	Yes
<b>ATmega48PA</b>	4	0,25	512	23	20	1,8-5,5	8	Yes	1	2	Yes	26	Yes	26	Yes	Yes	6	Yes	Yes	1+USART	Yes	1	Yes
<b>ATmega64</b>	64	2	4096	54	16	2,7-5,5	8	Yes	2	2	Yes	8	Yes	34	Yes	Yes	8	Yes	Yes	1	Yes	2	Yes
<b>ATmega640</b>	64	4	8192	86	16	1,8-5,5	16	Yes	4	2	Yes	32	Yes	57	Yes	Yes	16	Yes	Yes	1+USART	Yes	4	Yes
<b>ATmega644</b>	64	2	4096	32	20	1,8-5,5	8	Yes	1	2	Yes	32	Yes	31	Yes	Yes	6	Yes	Yes	1+USART	Yes	1	Yes
<b>ATmega644P</b>	64	2	4096	32	20	1,8-5,5	8	Yes	1	2	Yes	32	Yes	31	Yes	Yes	6	Yes	Yes	1+USART	Yes	2	Yes
<b>ATmega645</b>	64	2	4096	54	16	1,8-5,5	8	Yes	1	2	Yes	17	Yes	23	Yes	Yes	4	Yes	Yes	1+USI	USI	1	Yes
<b>ATmega6450</b>	64	2	4096	69	16	1,8-5,5	8	Yes	1	2	Yes	17	Yes	32	Yes	Yes	4	Yes	Yes	1+USI	USI	1	Yes
<b>ATmega64A</b>	64	2	4096	54	16	2,7-5,5	8	Yes	2	2	Yes	8	Yes	34	Yes	Yes	8	Yes	Yes	1	Yes	2	Yes
<b>ATmega8</b>	8	0,5	1024	23	16	2,7-5,5	6/8	Yes	1	2	Yes	2	Yes	18	Yes	Yes	3	Yes	Yes	1	Yes	1	Yes
<b>ATmega8515</b>	8	0,5	512	35	16	2,7-5,5	--	--	1	1	Yes	3	Yes	16	Yes	Yes	3	--	Yes	1	--	1	Yes
<b>ATmega8535</b>	8	0,5	512	32	16	2,7-5,5	8	Yes	1	2	Yes	3	Yes	20	Yes	Yes	4	--	Yes	1	Yes	1	Yes
<b>ATmega88PA</b>	8	0,5	1024	23	20	1,8-5,5	8	Yes	1	2	Yes	26	Yes	26	Yes	Yes	6	Yes	Yes	1+USART	Yes	1	Yes

Tab. 1.3: Comparação entre os ATtiny (04/2009).

Devices	Flash (Kbytes)	EEPROM (Kbytes)	SRAM (Bytes)	Max I/O Pins	F.max (MHz)	Vcc (V)	10-bit A/D Channels	Analog Comparator	16-bit Timers	8-bit Timer	Brown Out Detector	Ext Interrupts	Interrupts	ISP	On Chip Oscillator	PWM Channels	RTC	Self Program Memory	SPI	TWI	UART	Watchdog
ATtiny12	1	0.0625	--	6	8	1.8-5.5	--	Yes	--	1	Yes	1	5	Yes	Yes	--	--	--	--	--	--	Yes
ATtiny13A	1	0.0625	64B + 32 reg	6	20	1.8-5.5	4	Yes	--	1	Yes	6	9	Yes	Yes	2	--	Yes	--	--	--	Yes
ATtiny2313	2	0.125	128	18	20	1.8-5.5	--	Yes	1	1	Yes	2	8	Yes	Yes	4	--	Yes	USI	USI	1	Yes
ATtiny24	2	0.125	128	12	20	1.8-5.5	8	Yes	1	1	Yes	12	17	Yes	Yes	4	--	Yes	USI	USI	--	Yes
ATtiny24A	2	0.125	128	12	20	1.8-5.5	8	Yes	1	1	Yes	12	17	Yes	Yes	4	--	Yes	USI	USI	--	Yes
ATtiny25	2	0.125	128	6	20	1.8-5.5	4	Yes	--	2	Yes	7	15	Yes	Yes	4	--	Yes	USI	USI	--	Yes
ATtiny261	2	0.125	128	16	20	1.8 - 5.5	11	Yes	1	2	Yes	2	19	Yes	Yes	2	--	Yes	Yes	USI	--	Yes
ATtiny28L	2	--	32	11	4	1.8-5.5	--	Yes	--	1	--	2(+8)	5	--	Yes	--	--	--	--	--	--	Yes
ATtiny43U	4	0.0625	256	16	8	0.7 - 1.8	4	Y	--	2	Y	17	16	Y	--	4	--	--	Y	--	--	Y
ATtiny44A	4	0.25	256	12	20	1.8-5.5	8	Yes	1	1	Yes	12	17	Yes	Yes	4	--	Yes	USI	USI	--	Yes
ATtiny45	4	0.25	256	6	20	1.8-5.5	4	Yes	--	2	Yes	7	15	Yes	Yes	4	--	Yes	USI	USI	--	Yes
ATtiny461	4	0.25	256	16	20	1.8 - 5.5	11	Yes	1	2	Yes	2	19	Yes	Yes	2	--	Yes	Yes	USI	--	Yes
ATtiny48	4	0.0625	256	24/28	12	1.8 - 5.5	6/8	Yes	1	1	Yes	30	20	Yes	Yes	1	--	Yes	Yes	Yes	--	Yes
ATtiny84	8	0.5	512	12	20	1.8-5.5	8	Yes	1	1	Yes	12	17	Yes	Yes	4	--	Yes	USI	USI	--	Yes
ATtiny85	8	0.5	512	6	20	1.8-5.5	4	Yes	--	2	Yes	7	15	Yes	Yes	4	--	Yes	USI	USI	--	Yes
ATtiny861	8	0.5	512	16	20	1.8 - 5.5	11	Yes	1	2	Yes	2	19	Yes	Yes	2	--	Yes	Yes	USI	--	Yes
ATtiny88	8	0.0625	512	24/28	12	1.8 - 5.5	6/8	Yes	1	1	Yes	30	20	Yes	Yes	1	--	Yes	Yes	Yes	--	Yes

Para aplicações que exijam outras funcionalidades de hardware que as apresentadas pelos controladores das tabelas acima, o sítio do fabricante deve ser consultado.

## 2. O ATMEGA

Neste trabalho será abordado o ATmega8 por ser um  $\mu$ controlador que apresenta a maioria das características da família AVR e ser compacto (28 pinos PDIP), apresentando uma memória Flash de tamanho razoável. O importante é saber que ao programar este  $\mu$ controlador, os conceitos de programação de qualquer outro da família AVR são aprendidos dada a similaridade entre as famílias. As pequenas mudanças de hardware e software são resolvidas com uma busca ao referido *Datasheet*.

As características do ATmega8 são:

- $\mu$ controlador de baixa potência e alto desempenho, com arquitetura RISC avançada.
- 130 instruções, a maior parte executada em um único ciclo de relógio.
- 32x8 registradores de trabalho de propósito geral
- Operação de até 16 MIPS (milhões de instruções por segundo) a 16 MHz (ATmega88 – 20MHz)
- Multiplicação por hardware em 2 ciclos de relógio.
- 8 kbytes de memória de programa Flash de auto programação *In-System* (16K, 32K, 64K, 128K nos respectivos ATmega16, ATmega32, ATmega64 e ATmega128).
- 512 bytes de memória EEPROM.
- 1 kbyte de memória SRAM.
- Ciclos de escrita e apagamento: memória FLASH 10.000 vezes, EEPROM 100.000 vezes.
- Seção opcional para código de *Boot* com bits de bloqueio para programação *In-System* por *Boot Loader*.
- Bits de bloqueio para proteção do software.
- Possui os seguintes periféricos:
  - ➔ 23 entradas e saídas (I/Os) programáveis.
  - ➔ 2 Temporizadores/Contadores de 8 bits com *Prescaler* separado, 1 modo de comparação.

- ➔ 1 Temporizador/Contador de 16 bits com *Prescaler* separado, modo de comparação e captura.
- ➔ contador de tempo real (com cristal externo de 32.768 Hz conta precisamente 1s).
- ➔ 3 canais PWM.
- ➔ 8 canais A/D com precisão de 10 bits na versão TQFP, 6 canais na versão PDIP.
- ➔ interface serial para dois fios orientada a byte (TWI), compatível com o protocolo I2C.
- ➔ interface serial USART.
- ➔ interface serial SPI *Master/Slave*.
- ➔ *Watchdog Timer* com oscilador interno separado.
- ➔ 1 comparador analógico.
- Características especiais:
  - ➔ *Power-on Reset* e detecção *Brown-out* programável.
  - ➔ Oscilador interno RC (não há a necessidade do uso de cristal externo ou de outra fonte de *clock*).
  - ➔ Fontes de interrupções internas e externas.
  - ➔ 5 modos de *Sleep*: *Idle*, Redução de ruído do A/D, *Power-down*, *Power-save* e *Standby*.
- Tensão de operação: 2,7-5,5 V (ATmega8L), 4,5-5,5 V (ATmega8)
- Consumo de potência a 4 MHz (3V, 25°C): ativo = 3,6 mA, *Idle* = 1 mA e *Power-down* = 0,5 µA.

O núcleo AVR combina um rico conjunto de instruções com 32 registradores de trabalho, os quais estão diretamente conectados à Unidade Lógico-Aritmética (ALU), permitindo que dois registradores independentes sejam acessados com uma simples instrução em um único ciclo de *clock*. Seis dos 32 registradores podem ser usados como registradores de endereçamento indireto de 16 bits (ponteiros para o acesso de dados). Um destes ponteiros de dados pode também ser usado para acessar tabelas na memória flash. Estes registradores de 16 bits são denominados X, Y e Z. A Fig. 2.1 ilustra esses registradores e seus respectivos endereços na memória de dados.

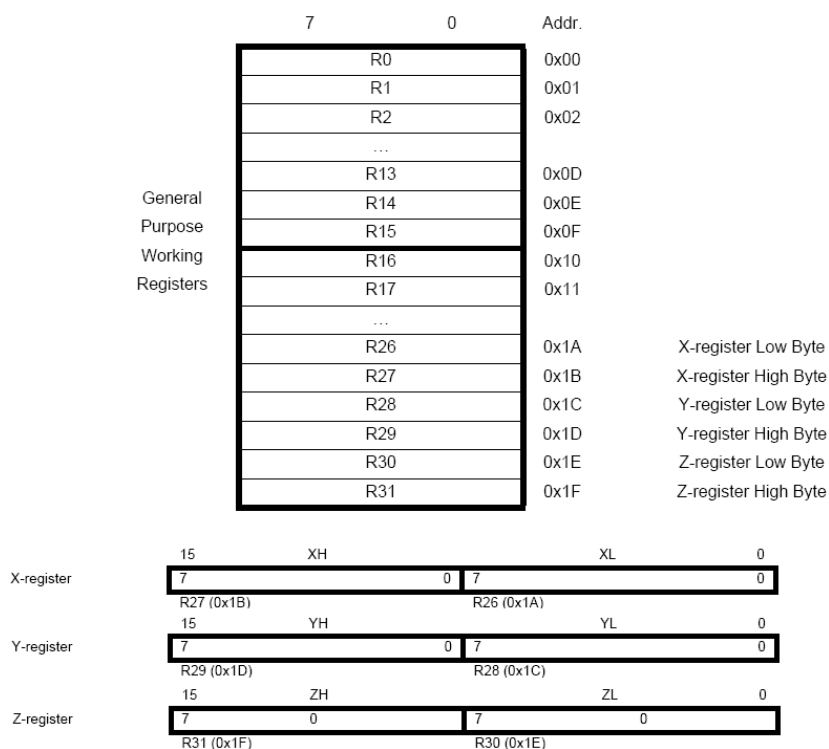


Fig. 2.1 – Registradores de trabalho da CPU do ATmega.

A função principal da Unidade de Processamento Central (CPU) é garantir a correta execução do programa, sendo capaz de acessar as memórias, executar cálculos, controlar os periféricos e tratar interrupções. Um diagrama em blocos mais detalhado da CPU do AVR pode ser visto na Fig. 2.2 e outro mais geral, incluindo os periféricos, na Fig. 2.3. Da arquitetura Harvard percebe-se a existência de barramento de dados para programa e para dados. O paralelismo permite que uma instrução seja executada enquanto a próxima é buscada na memória de programa, o que produz a execução de uma instrução por ciclo de *clock*.

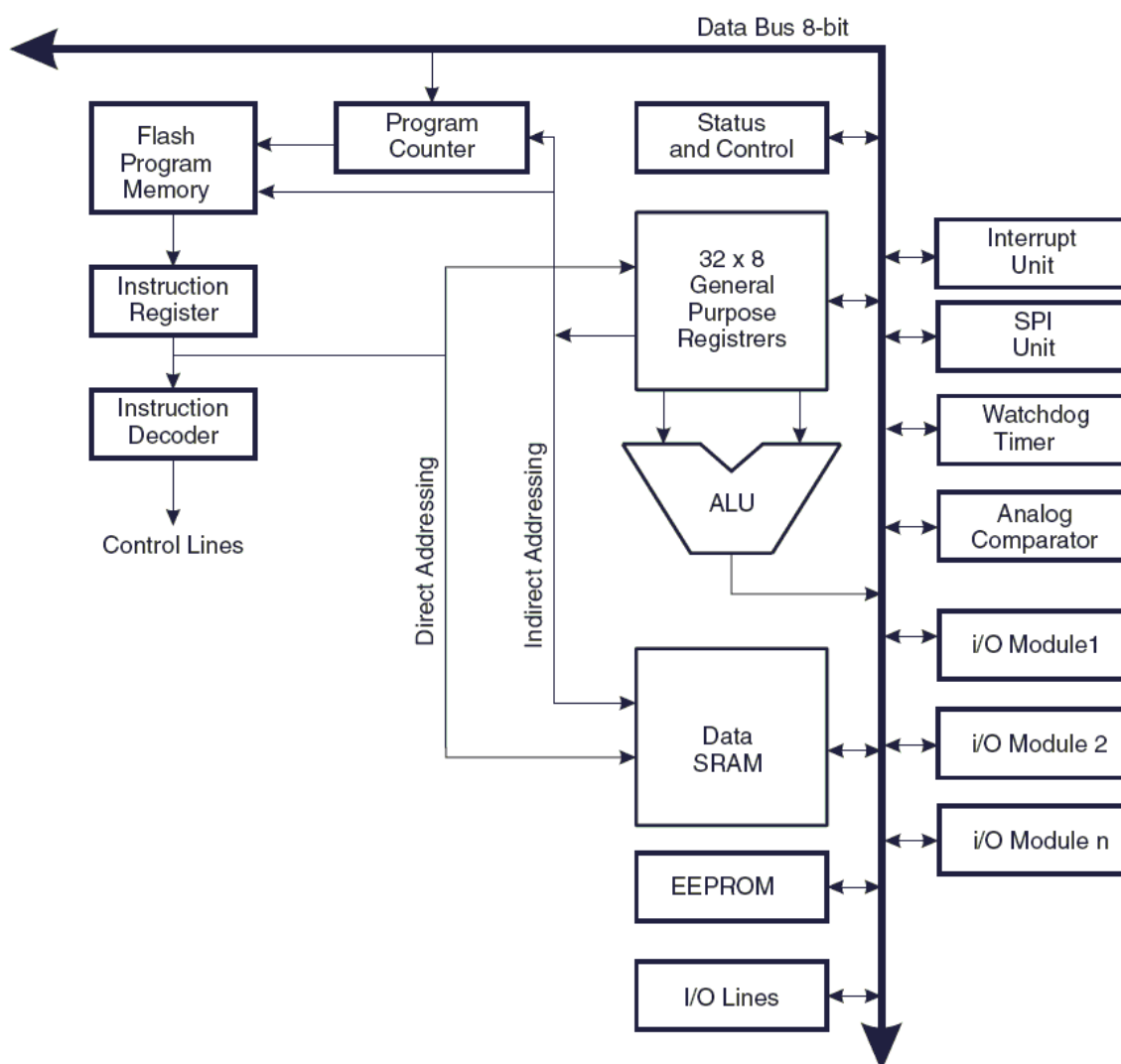


Fig. 2.2 – Diagrama em blocos da CPU do ATmega8.

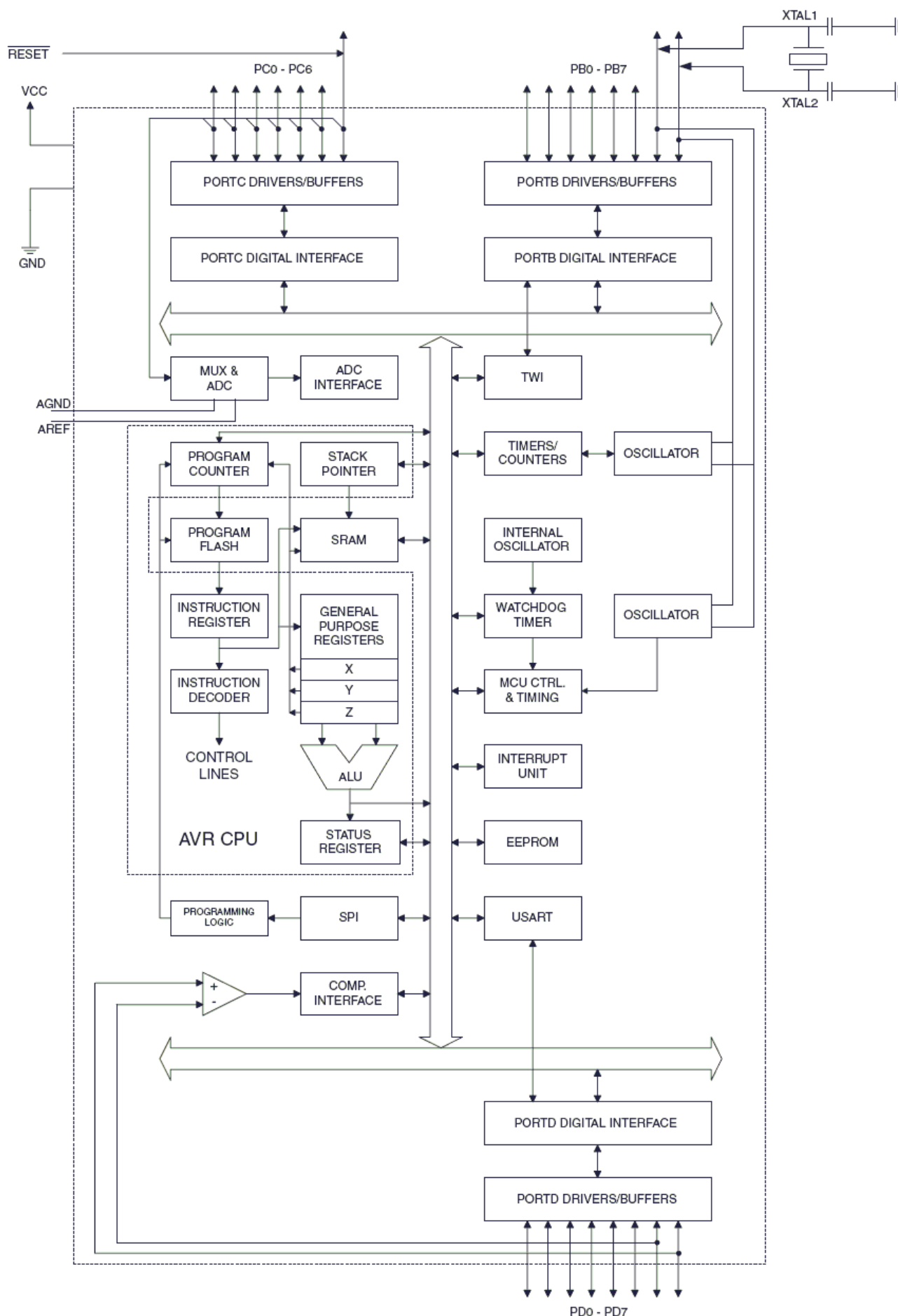


Fig. 2.3 – Diagrama em blocos do ATmega8.

## 2.1 AS MEMÓRIAS

A memória de dados e memória SRAM do ATmega8 pode ser vista na Fig. 2.4. A memória é linear começando no endereço 0 e indo até o endereço 1119 (0x045F). Destas 1120 posições de memória, 32 pertencem aos registradores de uso geral (Fig. 2.1), 64 aos registradores de entrada e saída (0x0020 até 0x005F) e 1024 bytes pertencem à memória SRAM (0x0060 até 0x045F).

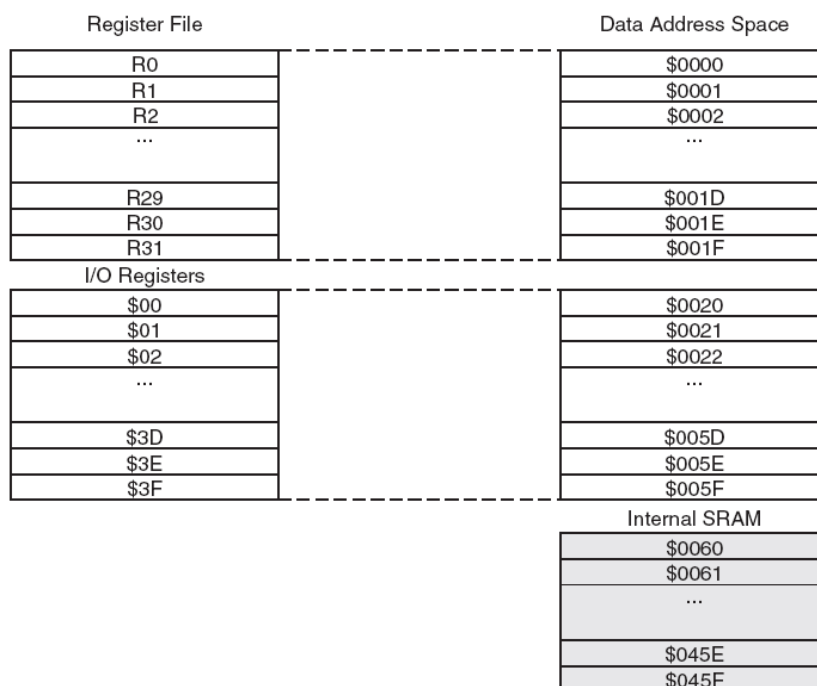


Fig. 2.4 – Memória de dados e memória SRAM.

A organização da memória de programa pode ser vista na Fig. 2.5. Existe uma seção específica para carregar o *Boot Loader* (programa para auto programação), ver Anexo 1.

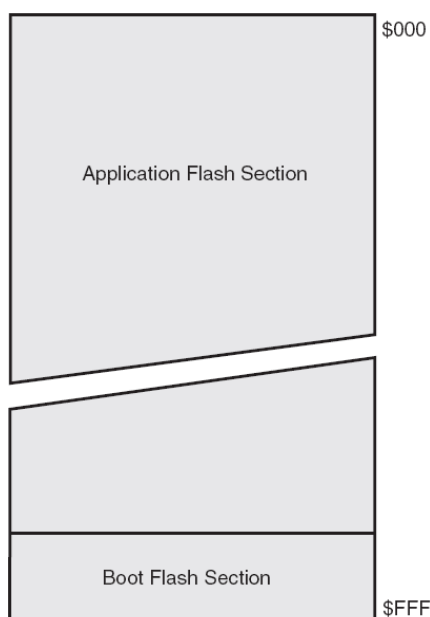


Fig. 2.5 – Organização da memória de programa.



A maioria das instruções do AVR emprega 16 bits (algumas, 32), ou seja, 2 bytes. Portanto, como o ATmega8 possui 8 kbytes de memória de programa, existem 4096 endereços (0x000 até 0xFFFF). Pode-se escrever até 4096 linhas de código em *Assembly*. O contador do programa (PC) é de 12 bits, endereçando todas as 4096 posições de memória ( $2^{12} = 4096$ ). A memória *flash* suporta no mínimo 10.000 ciclos de escrita e apagamento.

A memória EEPROM é de 512 bytes e é organizada separadamente. Cada byte individual pode ser lido ou escrito.

Um ponto importantíssimo são os registradores de entrada e saída, os quais possuem todas as informações referentes ao processamento da CPU. Permitem configurar e acessar todos os periféricos. É com esses registradores que o programador terá que se familiarizar para poder trabalhar com os periféricos (as “chaves” que ligam e desligam tudo). Dada a sua importância, os mesmos são apresentados na Tab. 2.1 (observar seus endereçamentos na Fig. 2.4). Os registradores de entrada e saída serão vistos com detalhes posteriormente.

### 2.1.1 O STACK POINTER

O *Stack Pointer* (SP, ponteiro de pilha) é usado principalmente para armazenagem temporária de dados: variáveis locais e endereços de retorno após chamadas de sub-rotinas e interrupções. O SP sempre aponta para o topo da pilha, crescendo dos endereços mais altos da memória para os mais baixos. Isto implica que o comando POP aumenta o SP e o comando PUSH o diminui. O SP é implementado na memória SRAM, devendo ter seu espaço definido por programação e apontar acima do endereço 0x60. Na Fig. 2.6 são apresentados os dois registradores do *Stack Pointer* (endereços 0x5E e 0x5D da memória de dados), onde se gravam os endereços da parte alta e baixa do início da pilha (SPH e SPL respectivamente).

Bit	15	14	13	12	11	10	9	8	
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Fig. 2.6- Detalhamento dos registradores do *Stack Pointer*.

Tab. 2.1: Registradores de entrada e saída da memória de dados.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C
0x3E (0x5E)	SPH	–	–	–	–	–	SP10	SP9	SP8
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
0x3C (0x5C)	Reserved								
0x3B (0x5B)	GICR	INT1	INT0	–	–	–	–	IVSEL	IVCE
0x3A (0x5A)	GIFR	INTF1	INTF0	–	–	–	–	–	–
0x39 (0x59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0
0x38 (0x58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	–	TOV0
0x37 (0x57)	SPMCR	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
0x36 (0x56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
0x35 (0x55)	MCUCR	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
0x34 (0x54)	MCUCSR	–	–	–	–	WDRF	BORF	EXTRF	PORF
0x33 (0x53)	TCCR0	–	–	–	–	–	CS02	CS01	CS00
0x32 (0x52)	TCNT0	Timer/Counter0 (8 Bits)							
0x31 (0x51)	OSCCAL	Oscillator Calibration Register							
0x30 (0x50)	SFIOR	–	–	–	–	ACME	PUD	PSR2	PSR10
0x2F (0x4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
0x2E (0x4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
0x2D (0x4D)	TCNT1H	Timer/Counter1 – Counter Register High byte							
0x2C (0x4C)	TCNT1L	Timer/Counter1 – Counter Register Low byte							
0x2B (0x4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High byte							
0x2A (0x4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low byte							
0x29 (0x49)	OCR1BH	Timer/Counter1 – Output Compare Register B High byte							
0x28 (0x48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low byte							
0x27 (0x47)	ICR1H	Timer/Counter1 – Input Capture Register High byte							
0x26 (0x46)	ICR1L	Timer/Counter1 – Input Capture Register Low byte							
0x25 (0x45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
0x24 (0x44)	TCNT2	Timer/Counter2 (8 Bits)							
0x23 (0x43)	OCR2	Timer/Counter2 Output Compare Register							
0x22 (0x42)	ASSR	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB
0x21 (0x41)	WDTCSR	–	–	–	WDCE	WDE	WDP2	WDP1	WDP0
0x20 <sup>(1)</sup> (0x40) <sup>(1)</sup>	UBRRH	URSEL	–	–	–	UBRR[11:8]			
	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
0x1F (0x3F)	EEARH	–	–	–	–	–	–	–	EEAR8
0x1E (0x3E)	EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
0x1D (0x3D)	EEDR	EEPROM Data Register							
0x1C (0x3C)	EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE
0x1B (0x3B)	Reserved								
0x1A (0x3A)	Reserved								
0x19 (0x39)	Reserved								
0x18 (0x38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x17 (0x37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x16 (0x36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x15 (0x35)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
0x14 (0x34)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x13 (0x33)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x12 (0x32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
0x11 (0x31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x10 (0x30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x0F (0x2F)	SPDR	SPI Data Register							
0x0E (0x2E)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X
0x0D (0x2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
0x0C (0x2C)	UDR	USART I/O Data Register							
0x0B (0x2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
0x0A (0x2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
0x09 (0x29)	UBRRL	USART Baud Rate Register Low byte							
0x08 (0x28)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
0x07 (0x27)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0
0x06 (0x26)	ADCSRA	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
0x05 (0x25)	ADCH	ADC Data Register High byte							
0x04 (0x24)	ADCL	ADC Data Register Low byte							
0x03 (0x23)	TWDR	Two-wire Serial Interface Data Register							
0x02 (0x22)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
0x01 (0x21)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0
0x00 (0x20)	TWBR	Two-wire Serial Interface Bit Rate Register							

## 2.2 DESCRIÇÃO DOS PINOS

Na Fig. 2.7 os nomes dos pinos do ATmega8 são apresentados para os encapsulamentos PDIP (Plastic Dual Inline Package) e TQFP (Thin profile plastic Quad Flat Package). As siglas nos pinos resumem as funcionalidades destes e serão abordadas em momento oportuno. A Tab. 2.2 contém a descrição sucinta dos referidos pinos.

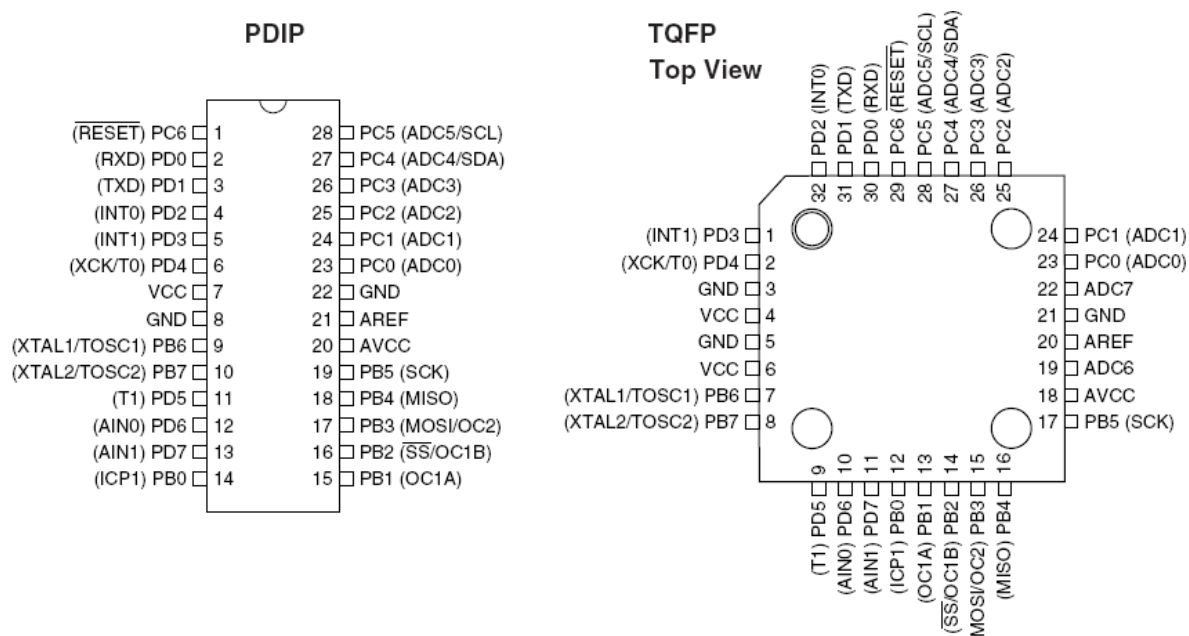


Fig. 2.7- Encapsulamentos PDIP e TQFP para o ATmega8.

Tab. 2.2 - Descrição sucinta dos pinos do Atmega8.

<b>VCC</b>	Tensão de alimentação.
<b>GND</b>	Terra.
<b>Port B (PB7..PB0) XTAL1/XTAL2/TOSC1/TOSC2</b>	<p>A Port B é uma porta bi-direcional de I/O de 8 bits com resistores internos de <i>pull-up</i> (selecionável para cada bit). Os <i>buffers</i> de saída possuem características simétricas com alta capacidade de fornecer e receber corrente. Como entradas, os pinos que forem externamente colocados em zero fornecerão corrente se os resistores de <i>pull-up</i> estiverem habilitados. Os pinos ficarão em <i>tri-state</i> quando uma condição de reset estiver ativa, mesmo que o <i>clock</i> não esteja rodando. Dependendo da seleção dos fusíveis de ajuste do <i>clock</i>, PB6 pode ser usado como entrada para o amplificador oscilador inversor e entrada para o circuito interno de <i>clock</i>. Da mesma forma, PB7 pode ser usado como saída do amplificador oscilador inversor. Se o oscilador RC calibrado internamente for empregado como fonte de <i>clock</i>, PB7..6 é usado como entrada TOSC2..1 para o Temporizador/Contador2 assíncrono se o bit AS2 no registrador ASSR estiver ativo.</p> <p>Dentre as outras funcionalidades da Port B estão: a interface SPI (SCK - <i>Master Clock Input</i>, MISO - <i>Master Input/Slave Output</i>, MOSI - <i>Master Output/Slave Input</i>, SS - <i>Master Slave Select</i>), OC1B (<i>Timer/Counter1 Output Compare Match B Output</i>), OC1A (<i>Timer/Counter1 Output Compare Match A Output</i>), ICP1 (<i>Timer/Counter1 Input Capture Pin</i>). Os últimos podem ser utilizados para gerar sinais PWM.</p>
<b>Port C (PC5..PC0)</b>	<p>A Port C é uma porta bi-direcional de I/O de 7 bits com resistores internos de <i>pull-up</i> (selecionável para cada bit). Os <i>buffers</i> de saída possuem características simétricas com alta capacidade de fornecer e receber corrente. Como entradas, os pinos que forem externamente colocados em zero fornecerão corrente se os resistores de <i>pull-up</i> estiverem habilitados. Os pinos ficarão em <i>tri-state</i> quando uma condição de reset estiver ativa, mesmo que o <i>clock</i> não esteja rodando.</p> <p>Estes pinos são as entradas do conversor Analógico/Digital (ADCx) e também servem para a comunicação por dois fios (SCL - <i>Two-wire Serial Bus Clock Line</i>, SDA - <i>Two-wire Serial Bus Data Input/Output Line</i>).</p>

<b>PC6/RESET</b>	Se o fusível RSTDISBL for programado, PC6 é usado com um pino de I/O, sendo que suas características elétricas diferem dos outros pinos da Port C. Caso contrário, PC6 é usado com entrada de <i>reset</i> . Um nível de tensão baixo neste pino por um período maior que uma determinada largura de pulso produzirá um <i>reset</i> , mesmo que o <i>clock</i> não esteja rodando.
<b>Port D (PD7..PD0)</b>	A Port D é uma porta bi-direcional de I/O de 8 bits com resistores internos de <i>pull-up</i> (selecionável para cada bit). Os <i>buffers</i> de saída possuem características simétricas com alta capacidade de fornecer e receber corrente. Como entradas, os pinos que forem externamente colocados em zero fornecerão corrente se os resistores de <i>pull-up</i> estiverem habilitados. Os pinos ficarão em <i>tri-state</i> quando uma condição de reset estiver ativa, mesmo que o <i>clock</i> não esteja rodando.  Dentre as outras funcionalidades da Port D estão: AIN1 ( <i>Analog Comparator Negative Input</i> ), AIN0 ( <i>Analog Comparator Positive Input</i> ), T1 ( <i>Timer/Counter 1 External Counter Input</i> ), XCK ( <i>USART External Clock Input/Output</i> ), T0 ( <i>Timer/Counter 0 External Counter Input</i> ), INT1 ( <i>External Interrupt 1 Input</i> ), INT0 ( <i>External Interrupt 0 Input</i> ), TXD ( <i>USART Output Pin</i> ) e RXD ( <i>USART Input Pin</i> ).
<b>AVCC</b>	Pino para a tensão de alimentação do conversor AD. Deve ser externamente conectado ao VCC, mesmo se o A/D não estiver sendo utilizado. Se o A/D for usado deve ser empregado um filtro passa-baixas entre este pino e o VCC.
<b>AREF</b>	Pino para a tensão de referência analógica do conversor AD.
<b>ADC7..6</b>	Disponível nos encapsulamentos TQFP e QFN/MLF. ADC7..6 servem como entradas analógicas para o conversor AD.

## 2.3 SISTEMA DE CLOCK

A Fig. 2.9 apresenta o sistema principal de *clock* do AVR e sua distribuição. Todos os *clocks* precisam ser ativos em algum momento. Para a redução do consumo de potência, os módulos de *clock* podem ser suspensos usando diferentes modos de programação. O AVR suporta as seguintes opções de *clock*: cristal ou ressonador cerâmico externo, cristal de baixa frequência externo, oscilador RC externo, sinal de *clock* externo e oscilador RC interno, ver Fig. 2.8.

Interessante é utilizar o oscilador interno quando o *clock* não precisa ser preciso, eliminando a necessidade de componentes externos ao  $\mu$ controlador. O oscilador interno pode ser programado para operar a 1 MHz, 2 MHz, 4 MHz ou 8 MHz.

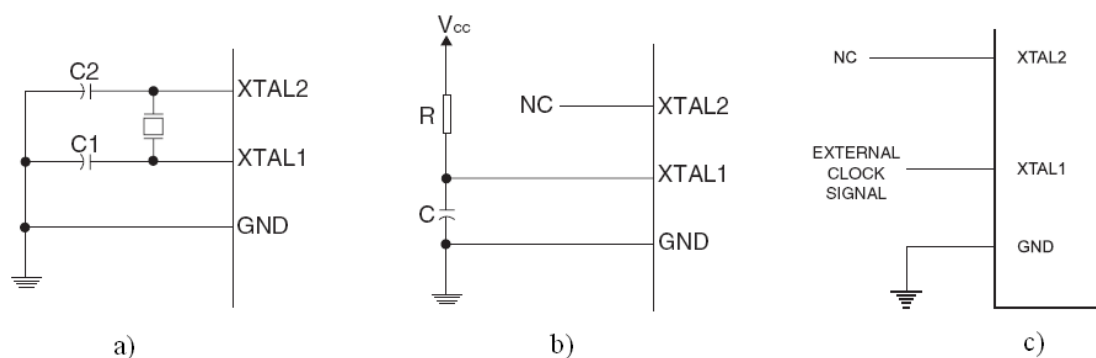


Fig. 2.8- Opções de *clock* externo para o AVR: a) cristal, b) rede RC e c) sinal externo.

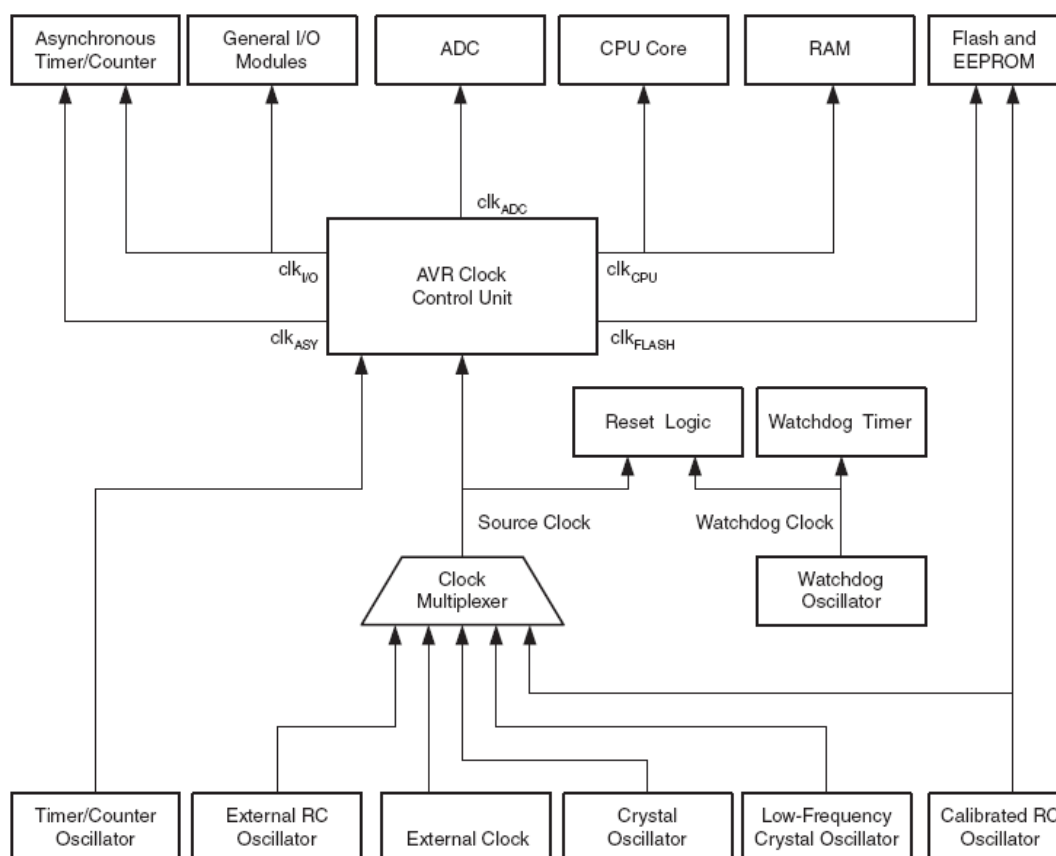


Fig. 2.9- Sistema de *clock* do AVR e sua distribuição.

## 2.4 O RESET

Durante o *reset*, todos os registradores de entrada e saída são ajustados aos seus valores *default* e o programa começa a ser executado a partir do vetor de *reset* (endereço 0 da memória de programa). O diagrama do circuito de *reset* é apresentado na Fig. 2.10.

As portas de I/O são imediatamente inicializadas quando uma fonte de *reset* é ativa. Isto não exige qualquer fonte de *clock*. Após a fonte de *reset* ficar inativa, é efetuado um atraso interno (configurável) mantendo o *reset* por um pequeno período de tempo. Assim, a tensão de alimentação pode alcançar um nível estável antes do  $\mu$ controlador começar a trabalhar.

O ATmega8 possui 4 fontes de *reset*:

- Power-on Reset: ocorre na energização enquanto a fonte de alimentação estiver abaixo do limiar de *Power-on Reset* ( $V_{POT}$ ).
- Reset externo: ocorre quando um nível baixo é aplicado ao pino de *reset* por um determinado período de tempo.
- Watchdog Reset: ocorre quando o *Watchdog* está habilitado e o seu temporizador estoura.
- Brown-out Reset: ocorre quando a tensão de alimentação cair abaixo do valor definido para o *Brown-out Reset* ( $V_{BOT}$ ) e o seu detector estiver habilitado.

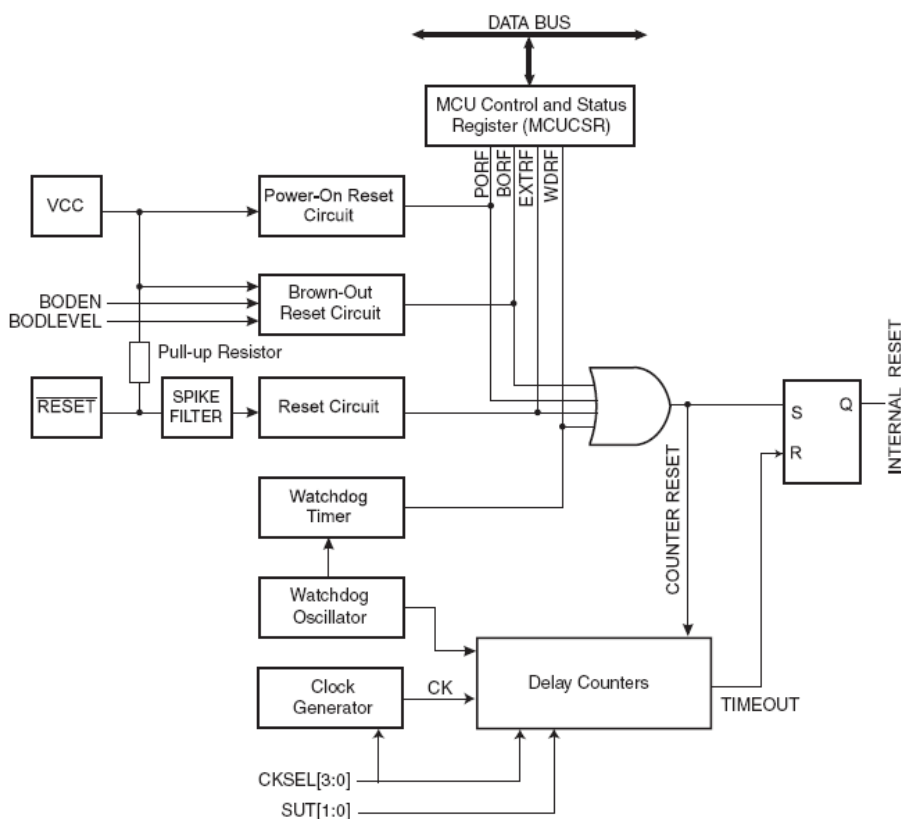


Fig. 2.10- Diagrama do circuito de *reset* do AVR.

## 2.5 GERENCIAMENTO DE ENERGIA E O MODO *SLEEP*

O modo *sleep* habilita o desligamento de módulos não utilizados pelo  $\mu$ controlador para a economia de energia, necessária no uso de baterias. Esse modo é tão importante que existe um código de instrução para ele, o SLEEP. O AVR possui 5 modos possíveis de *sleep*:

- Idle: a CPU é parada, mas SPI, USART, comparador analógico, AD, Interface Serial 2 Fios, Contadores/Temporizadores, Watchdog e o sistema de interrupção continuam operando.
- Redução de Ruído do AD: a CPU é parada, mas continuam operando o AD, as interrupções externas, o Temporizador/Contador2 e o Watchdog (se habilitado). Este modo é empregado para reduzir o ruído para o A/D e garantir sua alta resolução.
- Power-down: o oscilador externo é parado, mas continuam operando a Interface Serial 2 Fios, as interrupções externas e o Watchdog (se habilitado).
- Power-save: igual ao *Power-down* com exceção que o Contador/Temporizador2 trabalha assincronamente.
- Standby: é idêntico ao *Power-down* com exceção que o oscilador é mantido funcionando (válido para oscilador externo a cristal ou ressonante cerâmico). O  $\mu$ controlador “acorda” do *sleep* em 6 ciclos de *clock*.

### 3. COMEÇANDO O TRABALHO

Para programar o AVR é necessário o programa **AVR Studio** obtido gratuitamente do sítio da ATMEL ([www.atmel.com](http://www.atmel.com)). Para programação em C no AVR Studio, pode-se utilizar o programa **WinAVR** (<http://winavr.sourceforge.net/>), também gratuito. Outra opção para a linguagem C é o eficiente compilador IAR; por ser pago não será abordado neste trabalho. Para a gravação dos  $\mu$ controladores deve-se adquirir hardware específico, para tal, existem esquemas disponíveis na internet. A ATMEL, por exemplo, produz o **AVR Dragon**, com capacidade de emulação e *Debug in system* possuindo interface JTAG.

#### 3.1 CRIANDO UM PROJETO NO AVR STUDIO

Após instalado o WinAVR e o AVR Studio, é hora de criar o primeiro projeto, o qual conterá o arquivo de programa e todos os demais arquivos necessários para o trabalho com o  $\mu$ controlador. Primeiro no menu <Project> clique em <Project Wizard> (Fig. 3.1). Será aberta uma janela, na qual se deve clicar em <New Project>, Fig. 3.2.

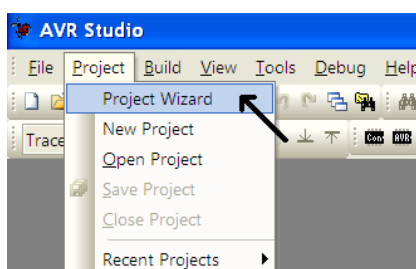


Fig. 3.1 – Menu *Project*.

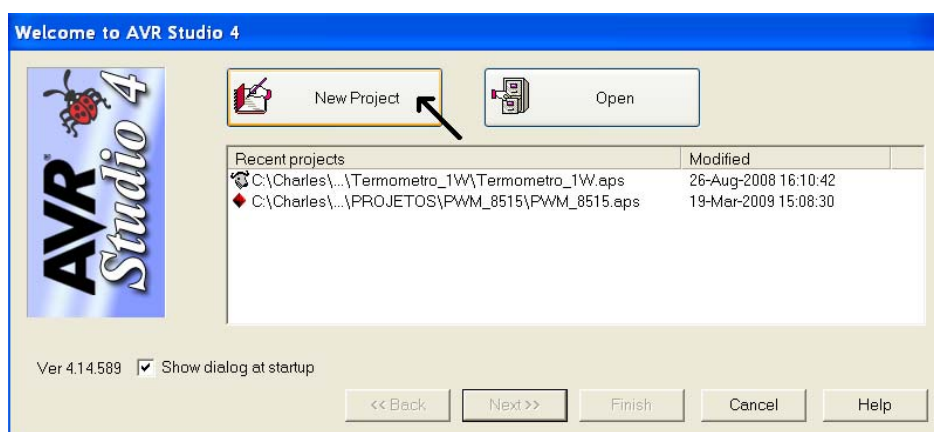


Fig. 3.2 – Janela do *Project Wizard*.

Agora, define-se qual tipo de programação será feita: assembly (Atmel AVR Assembly) ou C (AVR GCC), dá-se nome ao projeto e o nome do programa (no caso da Fig. 3.3, Primeiro\_Programa.asm). Após essas definições clica-se em <Next>.



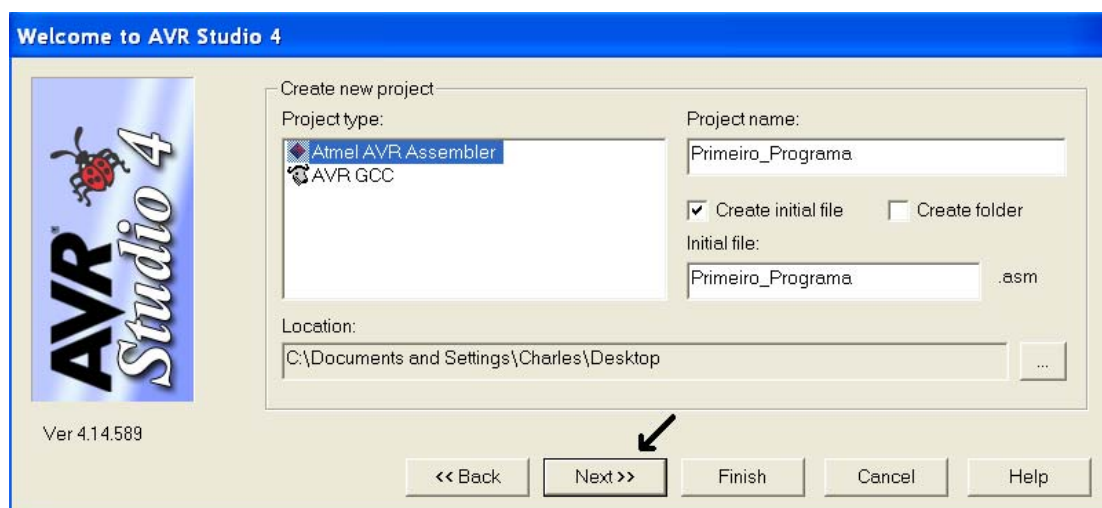


Fig. 3.3 – Definindo o tipo de programação e o nome do projeto.

A última definição é o modelo do  $\mu$ controlador que será empregado e qual a plataforma para o *Debug*. No caso somente de simulação: *AVR Simulator* (Fig. 3.4). Clica-se em <Finish>, o arquivo de programação é criado (Fig. 3. 5), agora só basta escrever as linhas de código!

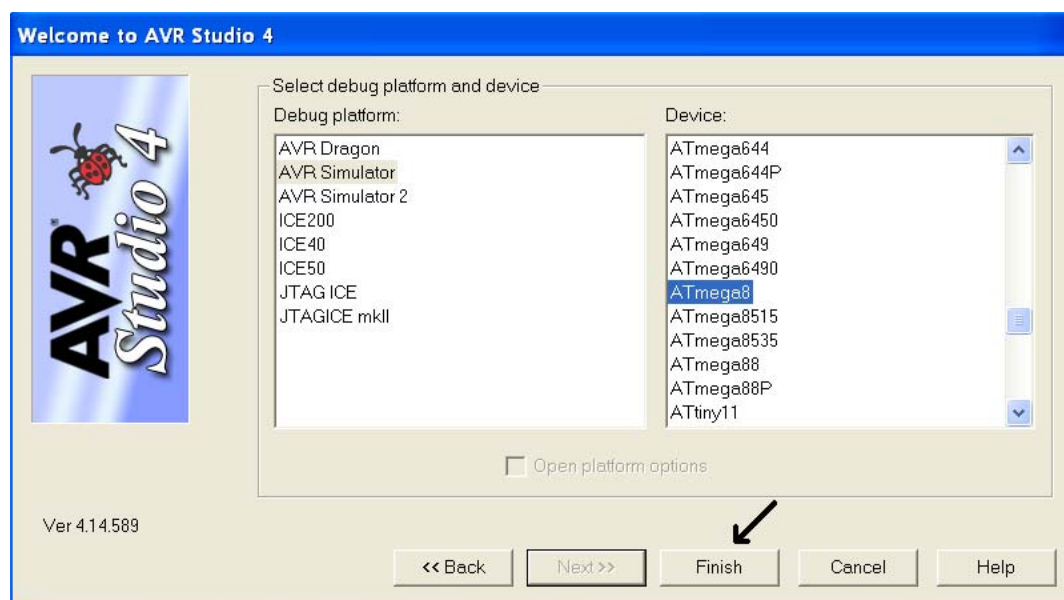


Fig. 3.4 – Definindo o  $\mu$ controlador e a ferramenta de *Debug*.



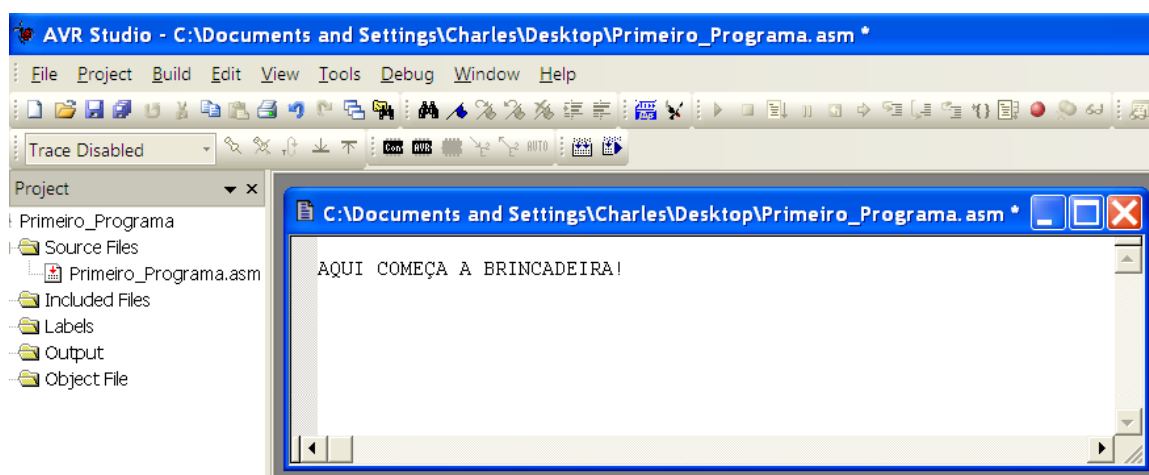


Fig. 3.5 – Projeto novo com arquivo pronto para o início da programação.

### 3.2 SIMULANDO NO PROTEUS (ISIS)

Analisar a programação utilizando o *Debug* do *AVR Studio* é válido para achar erros no programa, mas avaliar o comportamento real do  $\mu$ controlador exige a montagem do hardware. Outra saída, menos custosa e de rápido desenvolvimento é empregar um software capaz de simular o hardware. O PROTEUS (ISIS *schematic capture*) produzido pela *Labcenter Electronics* ([www.labcenter.co.uk](http://www.labcenter.co.uk)) simula inúmeros  $\mu$ controladores, incluindo vários ATmegas, e permite o *Debug* em conjunto com o *AVR Studio*. É possível adquirir diferentes licenças para diferentes  $\mu$ controladores (claro que todas pagas). O PROTEUS também possui o *ARES PCB Layout*, permitindo a confecção da placa de circuito impresso diretamente do esquema do circuito simulado.

Para criar um circuito  $\mu$ controlado no ISIS clique em <New Design> no menu <File>, Fig. 3.6.

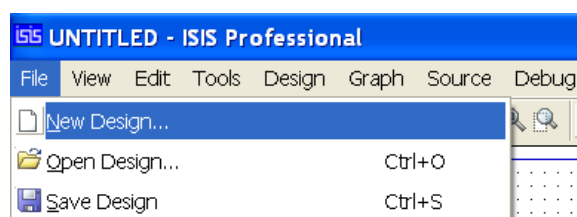



Fig. 3.6 – Criando um circuito  $\mu$ controlado no PROTEUS.

Os componentes eletrônicos disponíveis podem ser encontrados no ícone  (Fig. 3.7) que abre uma janela com todas as bibliotecas de componentes disponíveis (Fig. 3.8). Nesta janela, é apresentado o desenho do componente e seu encapsulamento, se disponível. Também existe um campo para a procura de componentes por palavras (Fig. 3.8).

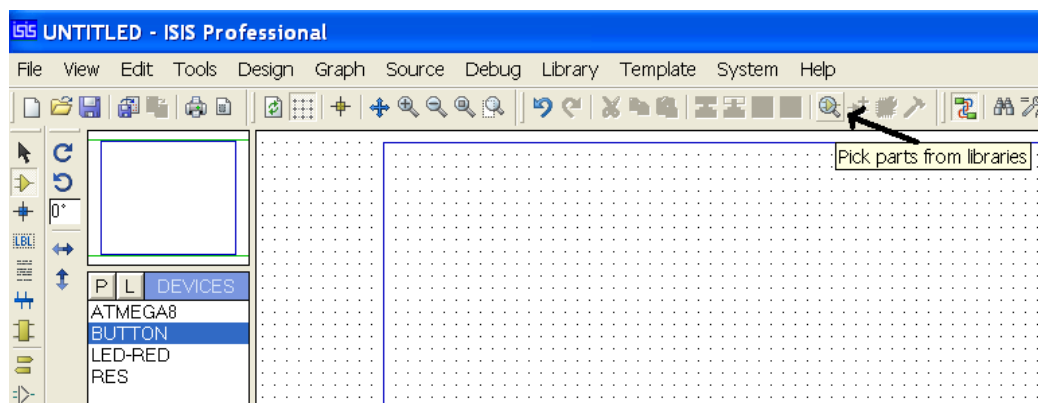


Fig. 3.7 – Encontrando os componentes eletrônicos no PROTEUS.

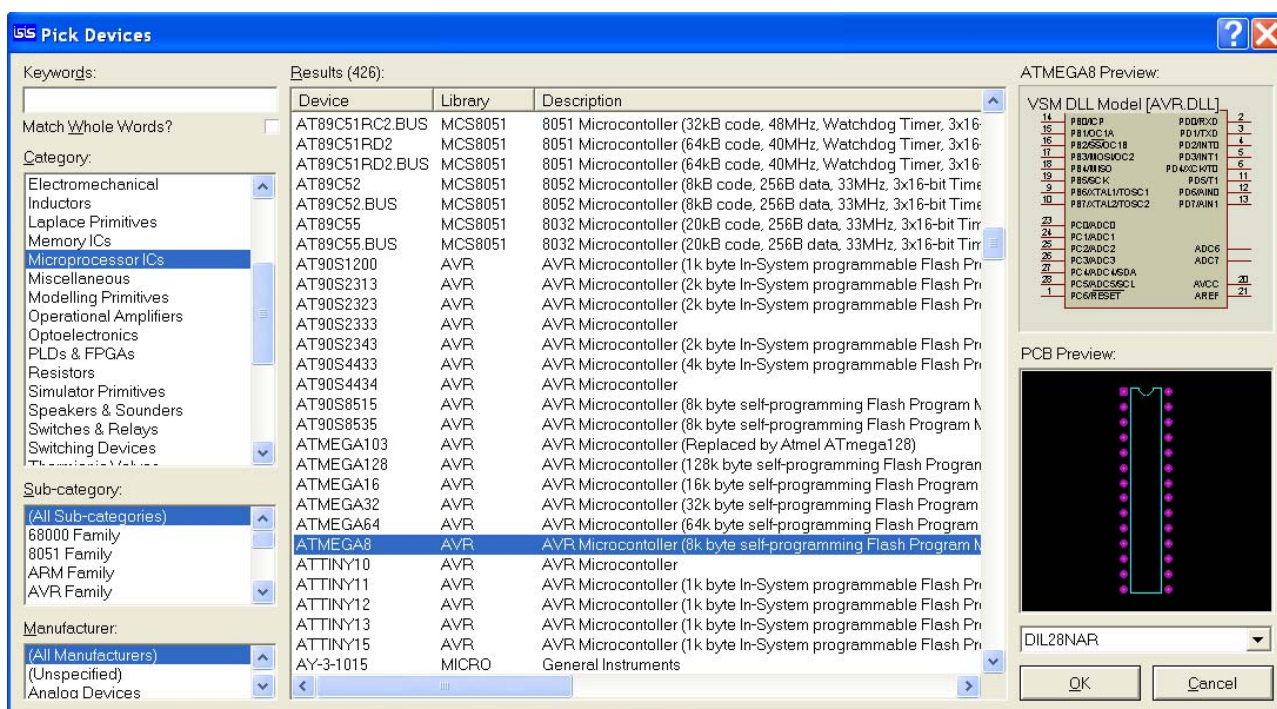





Fig. 3.8 – Janela de procura de componentes do PROTEUS.

No ícone da barra lateral esquerda  se encontram os símbolos de terra e de alimentação. A conexão entre os componentes do circuito é feita com o emprego do ícone , ver Fig. 3.9. Após a conclusão do diagrama esquemático, basta um duplo *click* sobre o  $\mu$ controlador para que se abra uma janela onde se deve indicar a localização do arquivo \*.hex, o qual possui o código a ser executado (criado na compilação do programa no AVR Studio) e se define a frequência de trabalho (Fig. 3.10). Agora, basta clicar no ícone  da barra horizontal inferior do canto esquerdo (Fig. 3.11) e o programa contido no arquivo \*.hex será executado. Se tudo estiver correto, nenhuma mensagem de erro será gerada e a simulação ocorrerá.

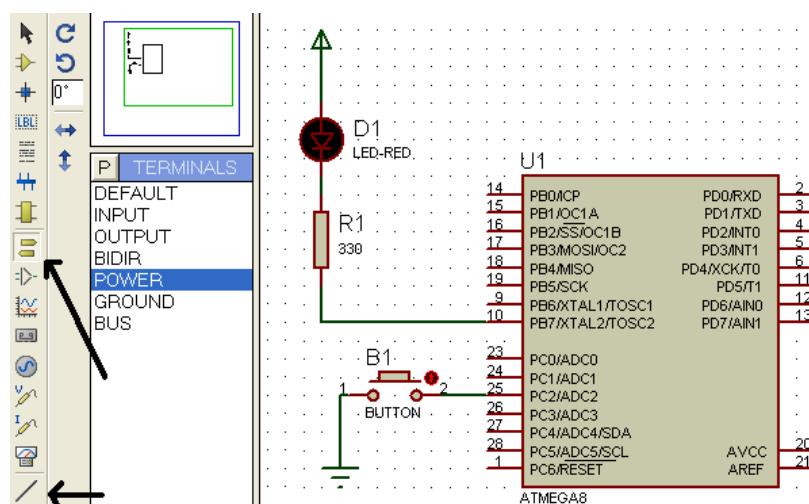


Fig. 3.9 – Ícones para o terra, a alimentação e fio de conexão.

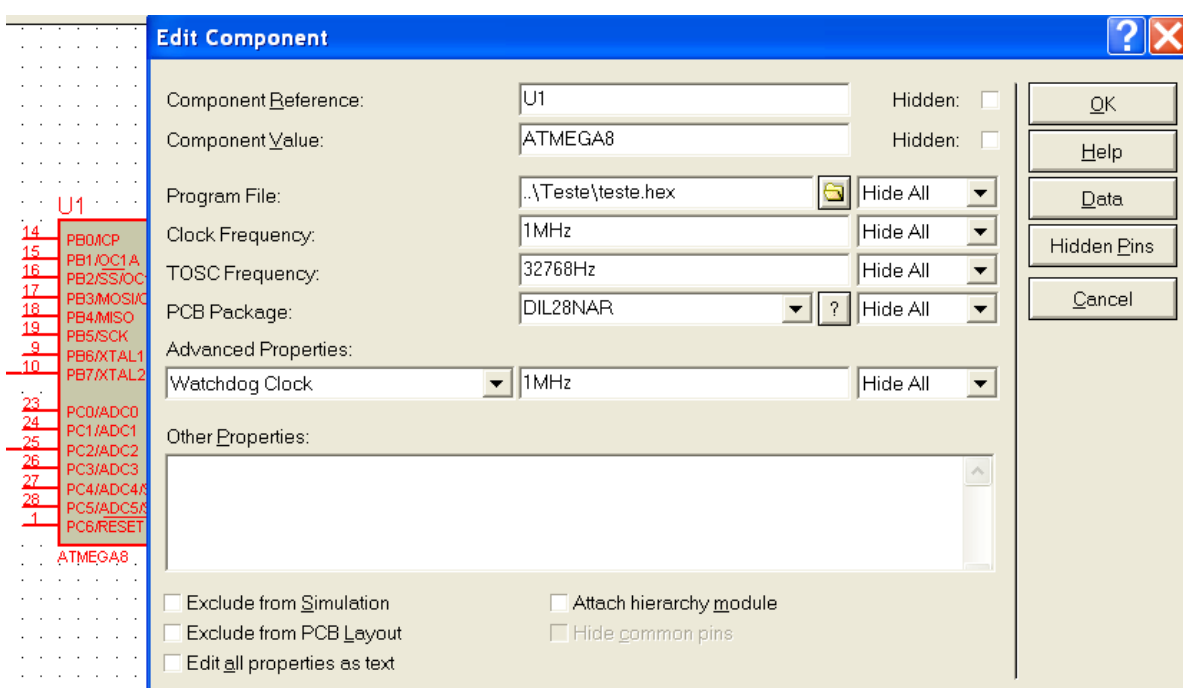


Fig. 3.10 – Janela para definir o arquivo de programa e outros detalhes referentes ao  $\mu$ controlador.



Fig. 3.11 – Executando a simulação do circuito.

#### 4. PORTAS DE ENTRADA E SAÍDA (I/Os)

O ATmega8 possui 3 portas: PORTB, PORTC e PORTD, com seus respectivos pinos: PB7 .. PB0, PC6 .. PC0 e PD7 .. PD0. Todos os pinos do AVR possuem a função Lê – Modifica – Escreve quando utilizados como portas genéricas de I/Os. Isto significa que a direção de um pino pode ser alterada sem mudar a direção de qualquer outro pino da mesma porta (instruções SBI e CBI). Da mesma forma, os valores lógicos dos pinos, bem como a habilitação dos resistores de *pull-up* (se configurado como entrada), podem ser alterados individualmente. O *buffer* de saída tem características simétricas com capacidade de receber ou fornecer corrente, suficientemente forte para alimentar LEDs diretamente (40 mA por pino). Todas as portas têm resistores de *pull-up* e diodos de proteção para o  $V_{CC}$  e o terra, como indicado na Fig. 4.1.

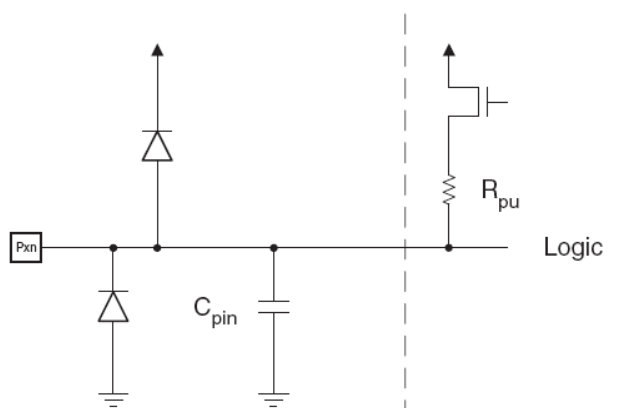


Fig. 4.1 – Esquema geral dos pinos de I/O (PXn).

Os registradores responsáveis pelos pinos de I/O são:

- PORTx: registrador de dados da porta, usado para escrever nos pinos.
- DDRx: registrador de direção da porta, usado para definir se os pinos são de entrada ou saída.
- PINx: registrador de entrada da porta, usado para ler o conteúdo dos pinos.

Em resumo, para uso de um pino de I/O, deve-se primeiro definir no registrador DDRx se ele será entrada ou saída. Se o pino for de saída, uma escrita no registrador PORTx altera o estado lógico do pino, também empregado para habilitar os *pull-ups*. Os estados dos pinos da porta são lidos do registrador PINx. Detalhe: para a leitura do PINx logo após uma escrita do PORTx e DDRx, deve ser gasto pelo menos um ciclo de máquina para sincronização dos dados pelo  $\mu$ controlador.

A Tab. 4.1 apresenta as configurações dos bits de controle dos registradores responsáveis pela definição do comportamento dos pinos (ver Tab. 2.2).

Tab. 4.1 - Bits de controle dos pinos das PORTs.

DDxn	PORTxn	I/O	Pull-up	Comment
0	0	Input	No	Tri-state (Hi-Z)
0	1	Input	Yes	Pxn will source current if external pulled low.
0	1	Input	No	Tri-state (Hi-Z)
1	0	Output	No	Output Low (Sink)
1	1	Output	No	Output High (Source)

#### 4.1 LENDO UM BOTÃO E LIGANDO UM LED

O primeiro programa  $\mu$ controlado é o simples e clássico ligar um *led* e pressionar um botão. Entretanto, na prática, botões apresentam o chamado *bounce*, o ruído produzido pelo contato elétrico e liberação momentânea do botão que pode ocorrer ao ser pressionado ou solto. Este ruído produz uma oscilação na tensão proveniente do botão ocasionando sinais lógicos aleatórios. Se existente, geralmente esse ruído desaparece após aproximadamente 10 ms (este tempo pode ser medido na montagem do circuito).

O objetivo desta seção é produzir um programa que troque o estado de um *led* toda vez que um botão for pressionado e utilizar uma técnica para eliminar o eventual ruído do botão. A técnica que será empregada para eliminar tal ruído é a inserção de um atraso após o botão ser solto, para garantir que o programa não leia o botão novamente durante a possível duração do ruído (na prática o ruído também pode aparecer quando se pressiona o botão). Na Fig. 4.2 é apresentado o fluxograma do programa desenvolvido. O circuito é apresentado na Fig. 4.3.

#### FLUXOGRAMA (Saber Eletrônica, Jan/2001)

O fluxograma é uma representação gráfica das tarefas de um programa, por meio de símbolos que fornecem a seqüência de um processo. O fluxograma ajuda a organizar o programa, permitindo o seu delineamento e possibilitando seguir passo a passo o que será executado nele.

Existem estruturas, definições rigorosas das regras de um fluxograma e uma variedade de símbolos para a construção do mesmo, o que é interessante seguir, principalmente quando se trata de um projeto complexo (ver Anexo 2). Com um fluxograma bem estruturado fica fácil corrigir eventuais erros que possam surgir no decorrer do projeto.

O fluxograma não é um elemento indispensável ao desenvolvimento de um programa, porém, sem ele se torna mais difícil qualquer alteração ou correção. O fluxograma deve ser feito com a visualização do problema a ser resolvido passo a passo, e o programa deve ser feito por partes, e

testado a cada parte para que fique mais simples detectar e solucionar qualquer tipo de erro. As alterações devem ser bem analisadas porque poderão influenciar outras partes do programa.

Essencialmente, o desenvolvimento de um programa deve ter os seguintes procedimentos:

- Análise do problema.
- Elaboração do fluxograma.
- Escrita do programa em linguagem simbólica.
- Tradução do programa para linguagem de máquina.
- Teste e se necessário correções.

Na análise do problema, devem ser determinados de maneira bem clara os objetivos a serem alcançados, exatamente que tarefa deve ser realizada e definir todo o hardware do sistema. O fluxograma deve ser dividido em tarefas e a definição da sequência que estas tarefas devem executar.

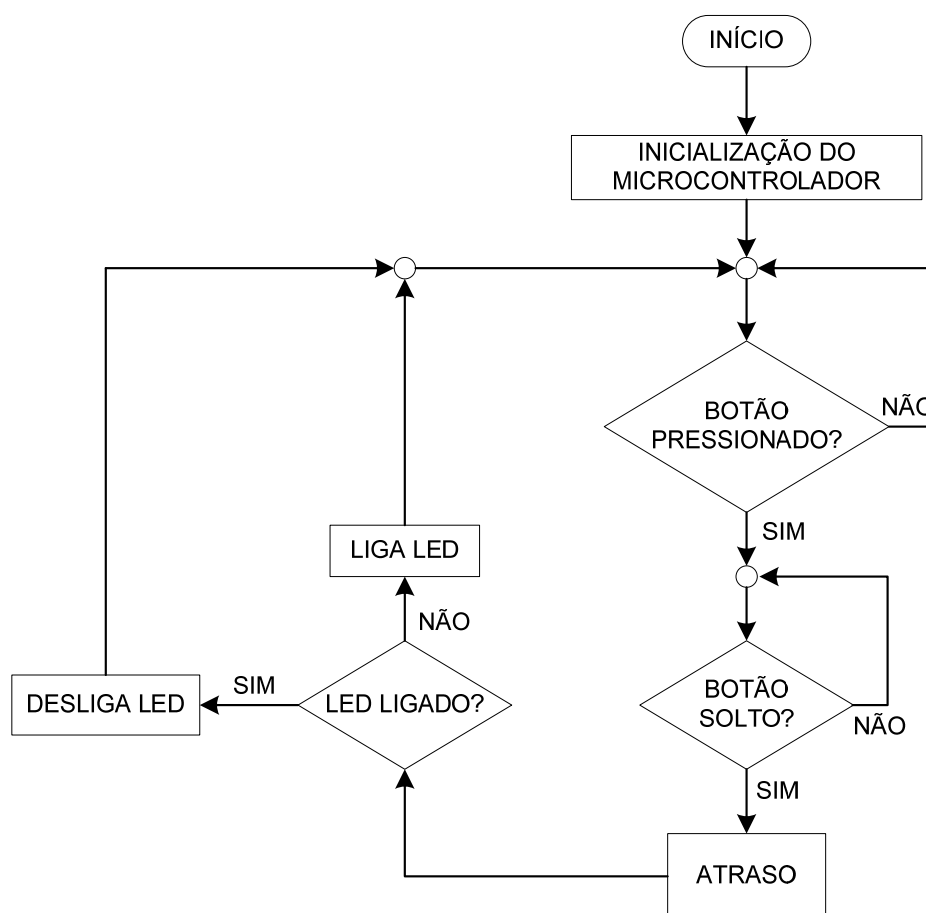


Fig. 4.2 – Fluxograma do programa para ligar e apagar um led com um botão.

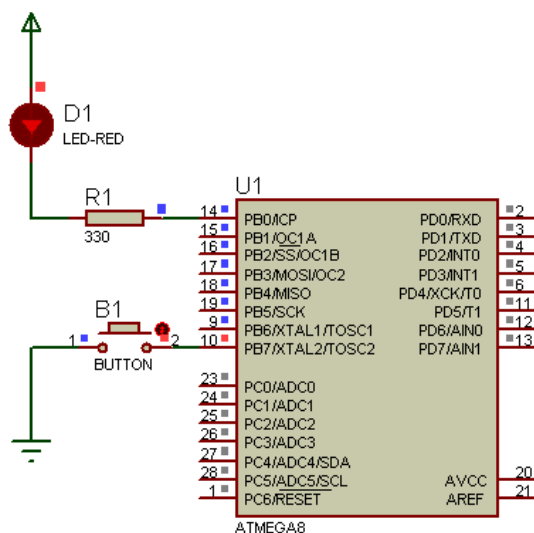


Fig. 4.3 – Circuito para ligar e apagar um *led* com um botão.

O programa em *assembly* é apresentado abaixo (ver Anexo 3 para detalhes das instruções em *assembly* do ATmega8). Após a compilação o tamanho do código foi de 50 bytes (25 instruções).

Todo bom programa deve ser adequadamente comentado e organizado. Códigos eficientes são resultantes de bons algoritmos, produzindo maior densidade de código.

```
//===== //
//      LIGANDO E DESLIGANDO UM LED QUANDO UM BOTÃO É PRESSIONADO      //
//===== //
.include "m8def.inc" //arquivo com as definições dos nomes dos bits e registradores do ATmega8

//DEFINIÇÕES
.equ LED      = PB0 //LED é o substituto de PB0 na programação
.equ BOTAO    = PB7 //BOTAO é o substituto de PB7 na programação
.def AUX      = R16; //R16 tem agora o nome de AUX (nem todos os 32 registradores de uso
                    //geral podem ser empregados em todas as instruções)

//-----
.ORG 0x000 //endereço de início de escrita do código
//após o reset o contador do programa aponta para cá

Inicializacoes:

    LDI  AUX,0x04 //inicializacao do Stack Pointer na posicao 0x045F da SRAM (Topo)
    OUT  SPH,AUX //registrador SPH = 0x04
    LDI  AUX,0x5F
    OUT  SPL,AUX //registrador SPL = 0x5F

    LDI  AUX,0b01111111 //carrega AUX com o valor 63 (1 saida 0 entrada)
    OUT  DDRB,AUX //configura porta B, PB0 .. PB6 saídas e PB7 entrada
    LDI  AUX,0b10000001 //habilita o pull-up para o botão e apaga o LED
    OUT  PORTB,AUX

    NOP //sincronização dos dados da porta para leitura imediata
//-----
//LAÇO PRINCIPAL
//-----
Principal:

    SBIC  PINB,BOTAO //verifica se o botao foi pressionado, senão
    RJMP  Principal //volta e fica preso no laço Principal

Esp_Soltar:

    SBIS  PINB,BOTAO //se o botão não foi solto, espera soltar
    RJMP  Esp_Soltar
    RCALL Atraso //após o botão ser solto gasta um tempo para eliminar
                //o ruído proveniente do mesmo
    SBIC  PINB,LED //se o LED estiver apagado liga e vice-versa
    RJMP  Liga
```

```

SBI    PORTB,LED    //apaga o LED
RJMP   Principal    //volta ler botão

Liga:

CBI    PORTB,LED    //liga LED
RJMP   Principal    //volta ler botão
//-----
//SUB-ROTINA DE ATRASO - Aprox. 25 ms a 8 MHz
//-----
Atraso:

DEC    R3            //decrementa R3, começa com 0x00
BRNE   Atraso        //enquanto R3 > 0 fica decrementando R3
DEC    R2
BRNE   Atraso        //enquanto R2 >0 volta decrementar R3
RET
//=====

```

Outra variante da leitura de botões é executar o que necessita ser feito imediatamente após o botão ser pressionado e avaliar se o botão já foi solto após isso. Quando se deseja incrementar uma variável enquanto o botão é mantido pressionado pode ser acrescida uma rotina de atraso adequada no laço de leitura do botão.

### ROTINA SIMPLES DE ATRASO

Rotinas de atraso são muito comuns na programação de  $\mu$ controladores. São realizadas fazendo-se a CPU gastar ciclos de máquina em alguns laços. Para o exemplo do programa acima, para se calcular o exato número de ciclos de máquina gastos na sub-rotina de atraso é necessário saber quantos ciclos cada instrução consome. A instrução DEC consome 1 ciclo, a instrução BRNE consome 2 ciclos e na última vez, quando não desvia mais, consome 1 ciclo. Como os registradores R3 e R2 possuem valor zero inicialmente e o decremento de R3 é repetido dentro do laço de R2 espera-se que haja 256 decrementos de R3 vezes 256 decrementos de R2. Se considerarmos 3 ciclos entre DEC e BRNE temos aproximadamente  $3 \times 256 \times 256$  ciclos, ou seja, 196.608 ciclos. Entretanto, BRNE não consome 2 ciclos no último decremento, assim, o cálculo preciso é mais complexo:  $((3 \text{ ciclos} \times 255) + 2 \text{ ciclos}) \times 255 + 769 \text{ ciclos} = 197.119 \text{ ciclos}$ . Para a melhor compreensão desse cálculo, observar a Fig. 4.4. Se considerarmos os ciclos gastos pela instrução RCALL (3 ciclos) e os ciclos da instrução RET (4 ciclos), temos um gasto total da chamada da sub-rotina até seu retorno de 197.126 ciclos.

$$\begin{array}{l}
 \text{Atraso:} \\
 \begin{array}{l}
 \text{DEC R3} \\
 \text{BRNE Atraso} \\
 \text{DEC R2} \\
 \text{BRNE Atraso}
 \end{array}
 \begin{array}{l}
 \left[ \begin{array}{l} +1 \text{ ciclo} \\ +2 \text{ ciclos} \end{array} \right] \\
 1 \text{ ciclo} \\
 2 \text{ ciclos}
 \end{array}
 \times 255 + \left[ \begin{array}{l} 1 \text{ ciclo} \end{array} \right] = 767 \text{ ciclos} \\
 \begin{array}{l} +1 \text{ ciclo} \\ +2 \text{ ciclos} \end{array} \times 255 + \left[ \begin{array}{l} 767 \text{ ciclos} \\ +1 \text{ ciclo} \\ +1 \text{ ciclo} \end{array} \right] = 197119 \text{ ciclos}
 \end{array}$$

Fig. 4.4 – Cálculo preciso da sub-rotina de atraso.



O tempo gasto pelo  $\mu$ controlador dependerá da frequência de trabalho utilizada. Como no AVR um ciclo de máquina equivale ao inverso da frequência do *clock* (período), o tempo gasto será:

$$Tempo\ Gasto = N^{\circ}\ de\ ciclos \times \frac{1}{Freq.de\ trabalho} \quad [4.1]$$

Logo, do nosso exemplo, para um *clock* de 8 MHZ (período de 0,125  $\mu$ s), da chamada da sub-rotina até seu retorno, resulta:

$$Tempo\ Gasto = 197.126 \times 0,125\ \mu s = 24,64075\ ms$$

### Exercícios:

- 4.1 – Elaborar um programa para ligar imediatamente o LED após o pressionar do botão, com uma rotina de atraso de 10 ms para eliminação do *bounce*.
- 4.2 – Elaborar um programa que troque o estado do LED se o botão continuar sendo pressionado. Utilize uma frequência que torne agradável o piscar do LED.
- 4.3 - Elaborar um programa em *assembly* que ligue 1 byte de leds (ver Fig. 4.5), com as seguintes funções:
  - 1.Ligue sequencialmente 1 led da esquerda para a direita.
  - 2.Ligue sequencialmente 1 led da direita para a esquerda.
  - 3.Ligue sequencialmente 1 led da direita para a esquerda e vice-versa (vai e volta).
  - 4.Ligue todos os leds e apague somente 1 led de cada vez da esquerda para a direita.
  - 5.Ligue todos os leds e apague somente 1 led de cada vez da direita para a esquerda.
  - 6.Ligue todos os leds e apague somente 1 led de cada vez da esquerda para a direita e vice-versa (vai e volta).
  - 7.Cada vez que um botão é pressionado (tecla de AJUSTE), os leds devem ligar realizando uma contagem crescente binária (início #00000000b).
  - 8.Cada vez que um botão é pressionado, os leds devem ser desligados realizando uma contagem decrescente binária (início #11111111b).

O circuito deve conter dois botões para controle das funções, um será o SET que quando pressionado permitirá que outro botão (AJUSTE) selecione a função desejada. Cada vez que o botão AJUSTE for pressionado um dos oito leds deverá acender para indicar a função escolhida, exemplo: 00000100 => led 3 ligado , função 3 selecionada. Quando o botão de SET for solto, o sistema começa a funcionar conforme função escolhida.

Desenvolva uma função por vez. Sempre faça o programa testando e reunindo as diferentes partes com cuidado. Não esqueça: um bom programa é bem comentado e organizado!

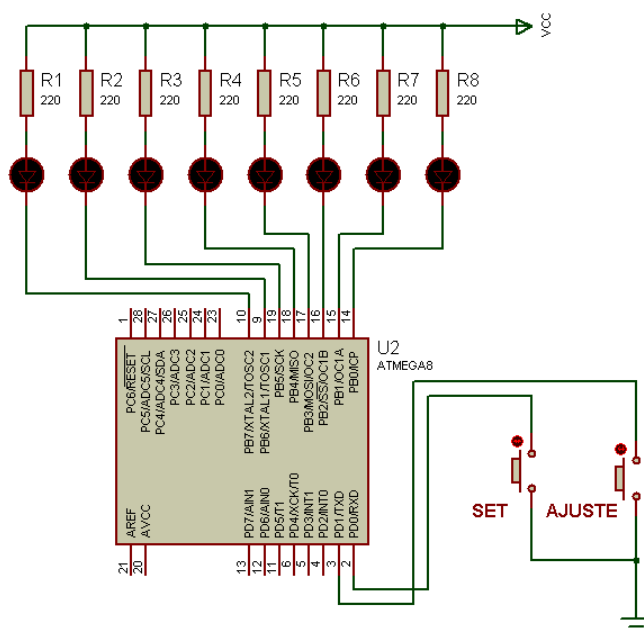


Fig. 4.5 – Sequencial de leds.

## 4.2 ACIONANDO DISPLAYs DE 7 SEGMENTOS

Um componente muito empregado no desenvolvimento de sistemas  $\mu$ controlados é o display de 7 segmentos. Estes displays geralmente são compostos por leds arranjados adequadamente em um encapsulamento, produzindo os dígitos numéricos que lhe são tão característicos. Na Fig. 4.6, é apresentado o diagrama esquemático para uso de um display de anodo comum e os caracteres mais comuns produzido por este. Nos displays catodo comum, o ponto comum dos leds é o terra e a tensão de alimentação deve ser aplicada individualmente em cada led do display.

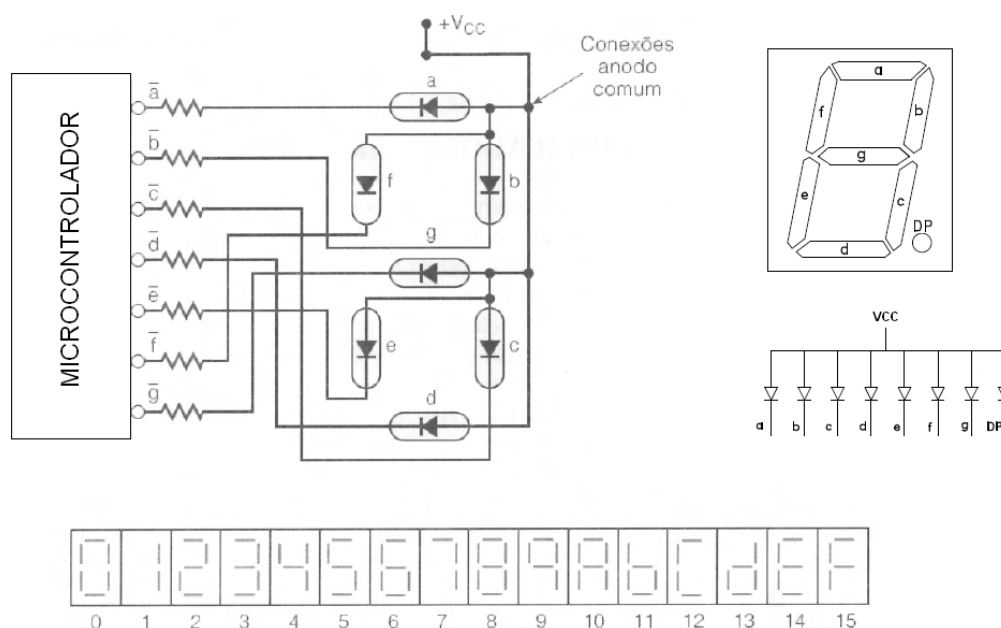


Fig. 4.6 – Display de 7 segmentos anodo comum.

Para o emprego de displays, é necessário decodificar (transformar em um número binário) o caractere que se deseja apresentar. A Tab. 4.2 apresenta o valor binário para os números hexadecimais de 0 até F, considerando o segmento a como sendo o bit menos significativo (LSB). Caso o display esteja ligado a uma porta de 8 bits, para ligar o ponto (DP) basta habilitar o 8º bit mais significativo (MSB).

Tab. 4.2 - Valores para a decodificação de display de 7 segmentos.

Dígito	Anodo comum		Catodo comum	
	gfedcba		gfedcba	
0	1000000b	0x40h	0111111b	0x3Fh
1	1111001b	0x79h	0000110b	0x06h
2	0100100b	0x24h	1011011b	0x5Bh
3	0110000b	0x30h	1001111b	0x4Fh
4	0011001b	0x19h	1100110b	0x66h
5	0010010b	0x12h	1101101b	0x6Dh
6	0000010b	0x02h	1111101b	0x7Dh
7	1111000b	0x78h	0000111b	0x07h
8	0000000b	0x00h	1111111b	0x7Fh



```

ADD  ZL,AUX          //soma posição de memória correspondente ao nr a apresentar na parte
                      //baixa do endereço
BRCC le_tab          //se houve Carry, incrementa parte alta do endereço, senão lê
                      //diretamente a memória
INC  ZH

le_tab:
LPM  R0,Z             //lê valor em R0
OUT  DISPLAY,R0       //mostra no display
RET

//-----
//Tabela p/ decodificar o display: como cada endereço da memória flash é de 16 bits, acessa-se a
//parte baixa e alta na decodificação
//-----
Tabela: .dw 0x7940, 0x3024, 0x1219, 0x7802, 0x1800, 0x0308, 0x2146, 0x0E06
//          1 0      3 2      5 4      7 6      9 8      B A      D C      F E
//=====

```

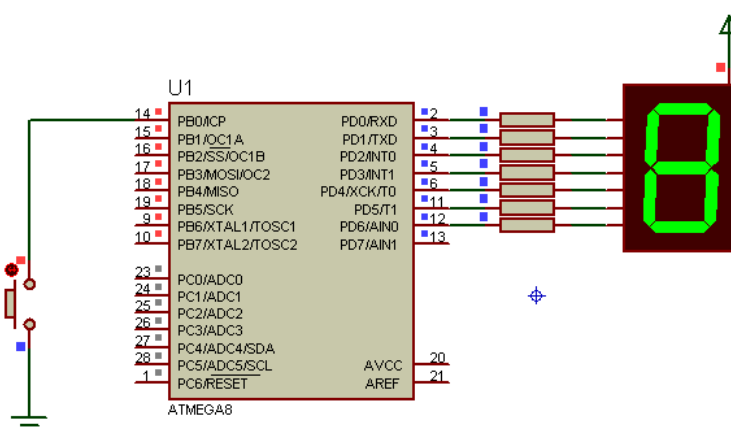


Fig. 4.7 – Circuito para acionamento de um display de 7 segmentos anodo comum.

O emprego de tabelas é muito usual e poderoso para a programação de  $\mu$ controladores. Elas devem ser armazenadas na memória de programa para evitar o desperdício da memória RAM e criação de código desnecessário na memória de programa. A gravação de dados na memória RAM aumenta o tamanho do código porque os valores movidos para as posições da RAM são gravados na memória de programa. Isto significa que é duplamente prejudicial empregar a RAM para armazenar dados que não serão alterados durante o programa. Este problema pode passar despercebido quando se programa em linguagem C.

É importante salientar que cada posição de memória *flash* do AVR contempla 16 bits. Entretanto, o hardware permite o acesso a dados gravados por bytes individualmente. O bit 0 do registrador ZL informa se deve ler a parte baixa ou alto do endereço. Para isto é preciso concatenar 2 bytes para cada posição de memória (ver a tabela do programa acima).

### **Exercício:**

**4.4** – Elaborar um programa em *assembly* para apresentar em um display de 7 segmentos um número aleatório entre 1 e 6 quando um botão é pressionado, ou seja, crie um dado eletrônico. Empregue o mesmo circuito da Fig. 4.7.

### 4.3 ACIONANDO LCDs 16x2

Os módulos LCDs são interfaces de saída muito úteis em sistemas  $\mu$ controlados. Estes módulos podem ser gráficos ou a caractere. Os LCDs comuns (tipo caractere) são especificados em número de linhas por colunas, sendo os mais usuais 16x2, 16x1, 20x2, 20x4, 8x2. Os módulos podem ser encontrados com *backlight* (leds para iluminação de fundo), facilitando leituras em ambientes escuros. Os LCDs mais comuns empregam o controlador Hitachi HD44780 com interface paralela, todavia, existem LCDs com controle serial e a gama de novos LCDs aumenta constantemente.

A seguir, iremos trabalhar com o LCD de 16 caracteres por 2 linhas, o uso de qualquer outro baseado no controlador HD44780 é direto. Na Fig. 4.8, é apresentado o circuito  $\mu$ controlado com um display de 16x2.

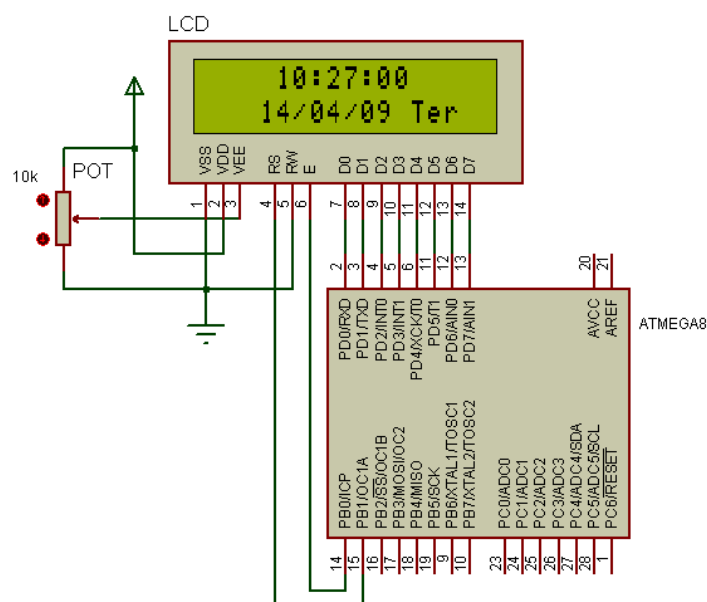


Fig. 4.8 – Circuito para acionamento de um LCD 16x2.

### INTERFACE DE DADOS DE 8 BITS

Para ler ou escrever no display LCD é necessário seguir a seguinte sequência de comandos:

1. Levar o pino R/W (*Read/Write*) para 0 lógico se a operação for de escrita e 1 lógico se for de leitura. Aterra-se o pino se não há necessidade de monitorar a resposta do LCD.
2. Levar o pino RS (*Register Select*) para o nível lógico 0 ou 1 (instrução ou caractere).
3. Transferir os dados para a via de dados (pode ser 8 ou 4 bits).
4. Levar o pino E (*Enable*) para 1 lógico.
5. Levar o pino E para 0 lógico.
6. Intercalar uma rotina de atraso entre as instruções ou fazer a leitura do *Busy Flag* (o bit 7 da linha de dados) antes do envio da instrução e enviá-la somente quando ele for 0 lógico.

Toda vez que a alimentação do módulo é ligada, o LCD executa a rotina de inicialização. Durante a execução dessa rotina o *Busy Flag* está em 1 lógico. Este estado “ocupado” dura por 10 ms após a tensão de alimentação atingir 4,5 V. Muitas vezes este detalhe é esquecido e o LCD não funciona adequadamente quando o  $\mu$ controlador manda informações antes do tempo. Para corrigir esse problema, basta colocar uma rotina de atraso no início do programa, só depois, então, as rotinas para o LCD.

As instruções executadas na inicialização do LCD são (detalhes no Anexo 5):

- Limpa display (0x01).
- Fixa modo de utilização (0x20), com DL = 1 (interface de 8 bits, N = 0 (linha única) e F = 0 (matriz 5x7).
- Controle ativo/inativo do display (0x08), com D = 0 (mensagem não aparente), C = 0 (cursor inativo) e B = 0 (função intermitente inativa).
- Fixa modo de operação (0x06), com I/D = 1 (modo incremental) e S = 0 (deslocamento do display inativo).

## PROGRAMAÇÃO C

As vantagens do uso do C comparado ao *assembly* são numerosas: redução do tempo de desenvolvimento, facilidade de re-uso do código e facilidade de manutenção e portabilidade. Os problemas podem ser: códigos compilados grandes e, conseqüentemente, redução da velocidade de processamento. Ressaltando-se que com o *assembly* (se bem programado) se consegue o máximo desempenho do  $\mu$ controlador.

Como os compiladores C são eficientes para arquitetura do AVR, a programação é feita em C, só em casos extremos existe a necessidade de se programar puramente em *assembly*. O bom é que os compiladores C aceitam trechos de código em *assembly* quando este for imprescindível.

A partir de agora será empregada exclusivamente a linguagem C para a programação do AVR. Detalhes do compilador WinAVR e suas bibliotecas podem ser encontrados no *help* do AVR *Studio*. No Anexo 4 encontram-se dicas para uma programação C eficiente. Abaixo se encontram as rotinas de escrita no LCD em conjunto com um programa exemplo que escreve na linha superior do LCD: “ABCDEFGHJKLMNPO”; e na linha inferior: “QRSTUVWXYZ 123456”, (circuito da Fig. 4.8).

```

//===== //
//      ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2      //
//      //                                                    //
//      Interface de dados de 8 bits                        //
//===== //
#define F_CPU 1000000UL      //define a frequência do ucontrolador 1 MHz (necessário para usar as
                             //rotinas de atraso)
#include <avr/io.h>          //definições do componente especificado
#include <util/delay.h>      //biblioteca para o uso das rotinas de atraso
#include <avr/pgmspace.h>    //uso de funções para salvar dados na memória de programa

//Definições de macros
#define set_bit(adress,bit) (adress|=(1<<bit))    //facilidades para o acesso aos pinos de I/O
#define clr_bit(adress,bit) (adress&=~(1<<bit))    // (PORTx,Pxx)
#define tst_bit(adress,bit) (adress&(1<<bit))      // leitura dos I/Os (PINx, Pxx)
#define cpl_bit(adress,bit) (adress^=(1<<bit))     //complementa o valor do bit

#define DADOS_LCD PORTD      //8 bits de dados do LCD na porta B
#define RS PB1              //pino de instrução ou dado para o LCD
#define E PB0               //pino de enable do LCD

const unsigned char msgl[ ] PROGMEM = "ABCDEFGHJKLMNOPS"; //mensagem 1 armazenada na memória flash
//-----
//Sub-rotina para enviar comandos ao LCD
//-----
void cmd_LCD(unsigned char c, char cd)
{
    DADOS_LCD = c;

    if(cd==0)
        clr_bit(PORTB,RS);    //RS = 0
    else
        set_bit(PORTB,RS);    //RS = 1

    set_bit(PORTB,E);          //E = 1
    clr_bit(PORTB,E);          //E = 0
    _delay_us(45);

    if((cd==0) && (c<127))    //se for instrução espera tempo de resposta do display
        _delay_ms(2);
}
//-----
//Sub-rotina de inicializacao do LCD
//-----
void inic_LCD(void)    //envio de instrucoes para o LCD
{
    cmd_LCD(0x38,0);    //interface 8 bits, 2 linhas, matriz 7x5 pontos
    cmd_LCD(0x0C,0);    //mensagem aparente cursor ativo nao piscando
    cmd_LCD(0x01,0);    //limpa todo o display
    cmd_LCD(0x80,0);    //escreve na primeira posicao a esquerda - 1a linha
}
//-----
//Sub-rotina de escrita no LCD
//-----
void escreve_LCD(char *c)
{
    for (; *c!=0;c++) cmd_LCD(*c,1);
}
//-----
int main( )
{
    unsigned char i;

    DDRB = 0xFF;        //porta B como saída
    DDRD = 0xFF;        //porta D como saída

    inic_LCD( );        //inicializa o LCD

    for(i=0;i<16;i++)    //enviando caractere por caractere
        cmd_LCD(pgm_read_byte(&msgl[i]),1); //lê na memória flash e usa cmd_LCD

    cmd_LCD(0xC0,0);    //desloca o cursor para a segunda linha do LCD

    escreve_LCD("QRSTUVWXYZ 123456");    //segunda mensagem para o LCD
    //mostra a versatilidade da rotina escreve_LCD
    for(;;);            //laço infinito
}
//=====

```

## INTERFACE DE DADOS DE 4 BITS

Utilizar a interface de dados de 8 bits para o LCD 16x2 com não tem sido mais empregado em projetos de hardware devido ao uso excessivo de pinos para o acionamento do display (10 ou 11). Empregar 4 bits de dados libera 4 pinos do  $\mu$ controlador para outras atividades, além de diminuir a complexidade da placa de circuito impresso. O custo é um pequeno aumento na complexidade do programa de controle do LCD, com alguns bytes a mais de programação.

Na Fig. 4.9 é apresentado o circuito  $\mu$ controlado do LCD 16x2 com via de dados de 4 bits. Na seqüência, o programa de controle do LCD.

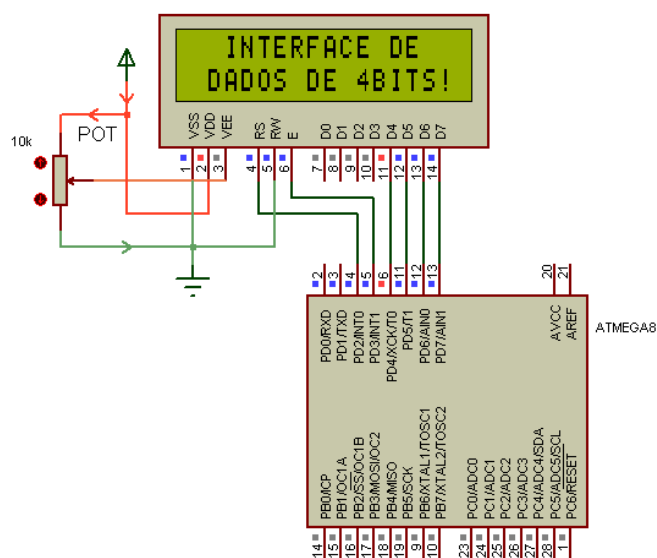


Fig. 4.9 – Circuito para acionamento de um LCD 16x2 com interface de dados de 4 bits.

```
//===== //
//      ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2      //
//      //                                                    //
//      Interface de dados de 4 bits                          //
//===== //
#define F_CPU 1000000UL //define a frequência do ucontrolador
#include <avr/io.h> //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de atraso
#include <compat/deprecated.h> //biblioteca para tratamento dos pinos de I/Os

#define DADOS_LCD PORTD //8 bits de dados do LCD na porta B
#define RS PD2 //pino de instrução ou dado para o LCD
#define E PD3 //pino de enable do LCD

unsigned char i=1; //variável para inicializar corretamente o LCD
//-----
//Sub-rotina para enviar comandos ao LCD com dados de 4 bits
//-----
void cmd_LCD(unsigned char c, char cd)
{
    DADOS_LCD = c;//primeiro os 4 MSB (PD4 - PD5 - PD6 - PD7) -> (D4 - D5 - D6 - D7 LCD)

    for(;i!=0;i--)
    {
        if(cd==0)
            cbi(PORTD,RS);
        else
            sbi(PORTD,RS);

        sbi(PORTD,E);
        cbi(PORTD,E);
        _delay_us(45);
    }
}
```



```

        if((cd==0) && (c<127))          //se for instrução espera tempo de resposta do display
            _delay_ms(2);

        DADOS_LCD = c<<4;                //4 bits menos significativos (nibble)
    }
    i = 2;
}
//-----
//Sub-rotina de inicializacao do LCD
//-----
void inic_LCD(void) //envio de instrucoes para o LCD
{
    // Como o LCD inicia no modo de 8 bits e somente 4 são conectados
    // o primeiro comando deve ser mandado duas vezes
    // a primeira para chavear para o modo de 4 bits (os bits menos
    // significativos não são vistos
    // a segunda para enviá-lo como 2 nibbles, então os menos
    // significativos são reconhecidos

    cmd_LCD(0x28,0); //interface de 4 bits
    cmd_LCD(0x28,0); //interface de 4 bits 2 linhas (aqui serao habilitadas as 2 linhas)
    cmd_LCD(0x0C,0); //mensagem aparente cursor ativo piscando
    cmd_LCD(0x01,0); //limpa todo o display
    cmd_LCD(0x80,0); //escreve na primeira posição a esquerda - 1ª linha
}
//-----
//Sub-rotina de escrita no LCD
//-----
void escreve_LCD(char *c)
{
    for (; *c!=0;c++) cmd_LCD(*c,1);
}
//-----
int main( )
{
    DDRD = 0xFF;                //porta D como saída
    inic_LCD( );                //inicializa o LCD
    escreve_LCD(" INTERFACE DE");
    cmd_LCD(0xC0,0);             //desloca cursor para a segunda linha
    escreve_LCD(" DADOS DE 4BITS!");

    for(;;);                    //laço infinito
}
//=====

```

O programa acima utiliza a porta D do ATmega sem se preocupar com os dois pinos PD0 e PD1 dessa porta. Caso se deseje empregá-los, o programa precisa ser alterado.

## CRIANDO CARACTERES NOVOS

Os códigos dos caracteres recebidos pelo LCD são armazenados em uma RAM chamada DDRAM (*Data Display RAM*) transformados em um caractere no padrão matriz de pontos e apresentados em na tela LCD (ver tabela no Anexo 5 para o conjunto de caracteres do 44780). Para produzir os caracteres nesse padrão, o módulo LCD incorpora uma CGROM (*Character Generator ROM*). Os LCDs também possuem uma CGRAM (*Character Generator RAM*) para a gravação de caracteres especiais. Oito caracteres programáveis estão disponíveis na CGRAM e utilizam os códigos 0x00 a 0x07. Os dados da CGRAM são apresentados em um mapa de bits de 8 bytes, dos quais se utilizam 5 bits (qualquer caractere é representado por uma matriz de pontos de 7x5), ver Fig. 4.10. Como há espaço para 8 caracteres, estão disponíveis 64 bytes nos seguintes endereços base: 0x40, 0x48, 0x50, 0x58, 0x60, 0x68, 0x70 e 0x78.









































Endereço da CGRAM	Mapa de Bits					Dado
50h						0Eh
51h						11h
52h						10h
53h						10h
54h						15h
55h						0Eh
56h						10h
57h						00h

Fig. 4.10 – Gravação do Ç na CGRAM, matriz 5x7. Esse caractere será selecionado pelo código 0x02.

Abaixo está o programa que cria dois caracteres novos, um no endereço 0x40 e o outro no 0x50. Após criados, os mesmos possuem o código 0x00 e 0x02, respectivamente. Na Fig. 4.11 é apresentado o resultado visual do LCD.

```
//===== //
//      CRIANDO CARACTERES PARA O LCD 16x2      //
//      Via de dados de 4 bits                    //
//===== //
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <compat/deprecated.h>
#include <avr/pgmspace.h>

#define DADOS_LCD PORTD
#define RS PD2
#define E PD3

unsigned char i=1;

//informação para criar o caractere Delta
const unsigned char carac1[] PROGMEM = {0b00100, //matriz de pontos
0b00100,
0b01010,
0b01010,
0b10001,
0b11111,
0b00000};

//informação para criar o caractere cedilha
const unsigned char carac2[] PROGMEM = {0b01110,
0b10001,
0b10000,
0b10000,
0b10101,
0b01110,
0b10000};

//-----
//Sub-rotina para enviar comandos ao LCD com dados de 4 bits
//-----
void cmd_LCD(unsigned char c, char cd)
{
    DADOS_LCD = c;

    for(;i!=0;i--)
    {
        if(cd==0)
            cbi(PORTD,RS);
        else
            sbi(PORTD,RS);

        sbi(PORTD,E);
        cbi(PORTD,E);
        _delay_us(45);

        if((cd==0) && (c<127))
            _delay_ms(2);
    }
}
```

```

        DADOS_LCD = c<<4;
    }
    i = 2;
}
//-----
//Subrotina de inicializacao do LCD
//-----
void inic_LCD(void)    //envio de instrucoes para o LCD
{
    cmd_LCD(0x28,0);    //interface de 4 bits
    cmd_LCD(0x28,0);    //interface de 4 bits 2 linhas (aqui se habilita as 2 linhas)
    cmd_LCD(0x0C,0);    //mensagem aparente cursor ativo nao piscando
    cmd_LCD(0x01,0);    //limpa todo o display
}
//-----
int main()
{
    unsigned char k;

    DDRD = 0xFF;        //porta D como saída
    inic_LCD();          //inicializa o LCD

    cmd_LCD(0x40,0);    //endereço base para gravar novo segmento, 0x40
    for(k=0;k<7;k++)
        cmd_LCD(pgm_read_byte(&carac1[k]),1); //grava 7 bytes na CGRAM começando no end. 0x40

    cmd_LCD(0x50,0);    //endereço base para gravar novo segmento 0x50
    for(k=0;k<7;k++)
        cmd_LCD(pgm_read_byte(&carac2[k]),1); //grava 7 bytes na CGRAM começando no end. 0x50

    cmd_LCD(0x80,0);    //endereço a posição para escrita dos caracteres no LCD
    cmd_LCD(0x00,1);    //apresenta primeiro caractere 0x00
    cmd_LCD(0x02,1);    //apresenta segundo caractere 0x02

    for(;;);            //laço infinito
}
//=====

```

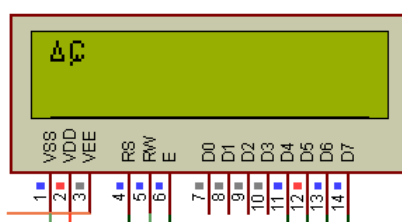
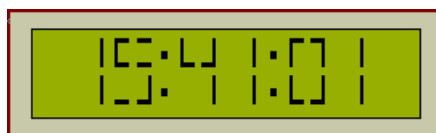


Fig. 4.11 - Dois caracteres novos: Δ e Ç.

O código acima apresenta didaticamente a forma de criar caracteres individualmente. Em uma programação mais eficiente, uma única matriz de informação pode ser utilizada. Da mesma forma, funções já desenvolvidas não necessitam serem reescritas; podem ser associadas em um único arquivo e disponibilizadas numa biblioteca.

## **Exercícios:**

- 4.5 – Crie oito caracteres novos para o LCD 16x2. Comece a criar sua própria biblioteca de funções.  
 4.6 – Usando as duas linhas do LCD crie números ou letras grandes (4 caracteres por dígito). A Fig. 4.12 exemplifica esta ideia.

Fig. 4.12 - *BIG NUMBER*.

## 5. INTERRUPÇÕES

As interrupções são muito importantes no trabalho com  $\mu$ controladores porque permitem simplificar o código e tornar mais eficiente o desempenho do processamento. O exemplo clássico é o teclado de computador: o processador não fica monitorando se alguma tecla foi pressionada, quando isto 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 nem de processamento.

As interrupções no AVR são “vetoradas”, significando que cada interrupção tem uma posição fixa na memória de programa. A Tab. 5.1 apresenta o endereço de cada interrupção. O ATmega8 apresenta um total de 19 tipos diferentes de interrupções. Quando o  $\mu$ controlador é inicializado, o contador de programa começa a rodar do endereço de *reset*, isto só não é válido quando se utiliza o *boot loader*.

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

Tab. 5.1 – Interrupções do ATmega8 e seus endereços.

Vector No.	Program Address	Source	Interrupt Definition
1	0x000	RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	TIMER2 COMP	Timer/Counter2 Compare Match
5	0x004	TIMER2 OVF	Timer/Counter2 Overflow
6	0x005	TIMER1 CAPT	Timer/Counter1 Capture Event
7	0x006	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	0x007	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	0x008	TIMER1 OVF	Timer/Counter1 Overflow
10	0x009	TIMER0 OVF	Timer/Counter0 Overflow
11	0x00A	SPI, STC	Serial Transfer Complete
12	0x00B	USART, RXC	USART, Rx Complete
13	0x00C	USART, UDRE	USART Data Register Empty
14	0x00D	USART, TXC	USART, Tx Complete
15	0x00E	ADC	ADC Conversion Complete
16	0x00F	EE_RDY	EEPROM Ready
17	0x010	ANA_COMP	Analog Comparator
18	0x011	TWI	Two-wire Serial Interface
19	0x012	SPM_RDY	Store Program Memory Ready



Tab. 5.2 – Bits de configuração da forma das interrupções nos pinos INT1 e INT0.

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

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

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	–	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

O registrador GIFR (*General 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.

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	–	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

Na Fig. 5.1 é apresentado um circuito exemplo para as duas interrupções externas, uma por nível e outra por transição. 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.

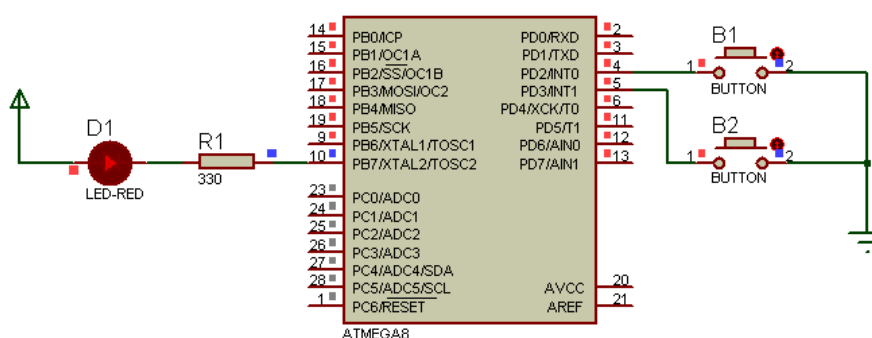


Fig. 5.1 – Circuito para emprego das interrupções externas.

```

//===== //
//      HABILITANDO AS INTERRUPÇÕES INT0 e INT1 POR TRANSIÇÃO E NÍVEL      //
//===== //
#define F_CPU 1000000UL

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

//Definições de macros
#define set_bit(adress,bit)  (adress|=(1<<bit))
#define clr_bit(adress,bit)  (adress&=~(1<<bit))
#define tst_bit(adress,bit)  (adress&(1<<bit))

unsigned char flags;          //empregado so 1 bit do byte flag

#define LED PB7;

//-----
ISR(INT0_vect) //interrupção externa 0, quando o botão é pressionado o led troca de estado
{
    if(tst_bit(flags,0))      //usa bit 0 do flag para poder trocar o estado do led
    {
        clr_bit(PORTB,PB7);    //liga led
        clr_bit(flags,0);      //permite apagar o led
    }
    else
    {
        set_bit(PORTB,PB7);    //apaga o led
        set_bit(flags,0);      //permite ligar o led
    }
}
//-----
ISR(INT1_vect) //interrupção externa 1, mantendo o botão pressionado o led pisca
{
    if(tst_bit(flags,0)) //código reescrito é mais eficiente, neste caso. Menos bytes compilados
    {
        clr_bit(PORTB,PB7);
        clr_bit(flags,0);
    }
    else
    {
        set_bit(PORTB,PB7);
        set_bit(flags,0);
    }
    _delay_ms(100);
}
//-----
int main()
{
    DDRD = 0x00;          //porta D entrada
    PORTD = 0xFF;          //pull-ups habilitados na porta D
    DDRB = 0xFF;          //port B saída
    PORTB = 0xFF;

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

    for(;;)               //executa tudo dentro das interrupções
}
//=====

```

Poderia ter sido empregada a macro `cpl_bit(adress,bit) (adress^=(1<<bit))` para trocar o valor do bit ao invés de uma função com *flag*. O exemplo acima é apenas uma forma didática para o uso de *flags* e uma compilação com menos bytes.

## 6. GRAVANDO A EEPROM

O ATmega8 possui 512 bytes de memória de dados EEPROM, a qual é organizada em um espaço de memória separado, nos quais bytes individuais podem ser lidos e escritos. A EEPROM suporta pelo menos 100.000 ciclos de escrita e apagamento. O acesso entre a EEPROM e a CPU é feito pelos registradores de endereço, de dados e de controle da EEPROM.

O tempo de escrita de um dado é de 8,5 ms, tempo que pode ser longo demais para certas aplicações. Caso esse tempo seja crítico, pode-se empregar a interrupção da EEPROM para avisar quando uma escrita foi completada, liberando o programa dessa monitoração.

Os seguintes passos devem ser seguidos para a escrita na EEPROM (a ordem dos passos 3 e 4 não é importante):

1. Esperar até que o bit EEWB do registrador EECR fique zero;
2. Esperar até que o bit SPMEN no registrador SPMCR fique zero;
3. Escrever o novo endereço da EEPROM no registrador EEAR (opcional);
4. Escrever o dado a ser gravado no registrador EEDR (opcional);
5. Escrever 1 lógico no bit EEMWE enquanto escreve um 0 no bit EEWB do registrador EECR;
6. Dentro de quatro ciclos de *clock* após setar EEMWE, escrever 1 lógico no bit EEWB.

Os exemplos de códigos a seguir assumem que nenhuma interrupção irá ocorrer durante a execução das funções.

### Código em Assembly:

EEPROM\_escrita:

```

SBIC EECR,EEWB      ;Espera completar um escrita prévia
RJMP EEPROM_escrita
OUT  EEARH, R18      ;Escreve o endereço (R18:R17) no registrador de endereço
OUT  EEARL, R17
OUT  EEDR,R16        ;Escreve o dado (R16) no registrador de dado
SBI  EECR,EEMWE      ;Escreve um lógico em EEMWE
SBI  EECR,EEWB      ;Inicia a escrita setando EEWB
RET

```

EEPROM\_leitura:

```

SBIC EECR,EEWB      ;Espera completar um escrita prévia
RJMP EEPROM_leitura
OUT  EEARH, R18      ;Escreve o endereço (R18:R17) no registrador de endereço
OUT  EEARL, R17
SBI  EECR,EERE      ;Inicia leitura escrevendo em EERE
IN   R16,EEDR        ;Lê dado do registrador de dados em R16
RET

```



## Código em C:

```
void EEPROM_escrita(unsigned int endereco, unsigned char dado)
{
    while(EECR & (1<<EWE));    //Espera completar um escrita prévia
    EEAR = endereco;           //Escreve enderecos de escrita dados nos registr.
    EEDR = dado;
    EECR |= (1<<EEMWE);        //Escreve um lógico em EEMWE
    EECR |= (1<<EWE);          //Inicia a escrita setando EWE
}

unsigned char EEPROM_leitura(unsigned int endereco)
{
    while(EECR & (1<<EWE));    //Espera completar um escrita prévia
    EEAR = endereco;           //Escreve endereco de leitura
    EECR |= (1<<EERE);         //Inicia a leitura setando EERE
    return EEDR;                //retorna o valor lido do registrador de dados
}
```

Para trabalhar no AVR *Studio*, basta incluir a biblioteca <avr/eeprom.h> que possui funções prontas para serem empregadas com a EEPROM, conforme exemplo a seguir.

```
//=====//
//          GRAVANDO E LENDO A EEPROM          //
//=====//
#include <avr/io.h>
#include <avr/eeprom.h>

unsigned char dado1 = 0x13, dado2;
unsigned int endereco = 0x02;
//-----
int main()
{
    _EEPUT(endereco, dado1);    //gravando 0x13 no endereco 0x02 da EEPROM
    _EGET(dado2, endereco);     //lendo o endereco 0x02 para a variavel dado2

    for(;;);
}
//=====
```

## 7. TECLADO MATRICIAL

Uma forma muito comum de entrada de dados em um sistema  $\mu$ controlado é através de teclas (botões). Quando o número delas é pequeno, cada uma pode ser associada a um pino de entrada do  $\mu$ controlador. Entretanto, quando o número de teclas se torna grande não é recomendado desperdiçar pinos de entrada. Um teclado convencional emprega 3x4 teclas (12) ou 4x4 teclas (16), ver Fig. 7.1. Ao invés de se empregar 12 ou 16 pinos para leitura dos referidos teclados, emprega-se 7 ou 8 pinos, respectivamente.

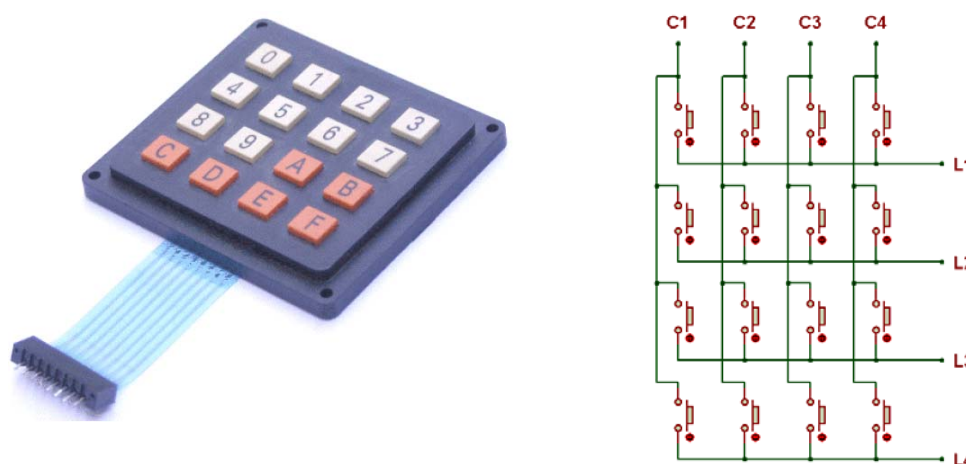


Fig. 7.1 – Teclado matricial hexa: 4x4.

Para a leitura de teclados com várias teclas é necessário empregar o conceito de matriz, em que os botões são arranjados em colunas e linhas (ver Fig. 7.1) e o teclado é lido por varredura, através de um CI decodificador ou um  $\mu$ controlador. O pressionar de uma tecla produzirá um curto entre uma coluna e uma linha e o sinal da coluna se refletirá na linha ou vice-versa. A ideia é trocar sucessivamente o nível lógico das colunas e verificar se esse nível lógico aparece nas linhas (ou vice-versa). Se por exemplo, houver alteração no sinal lógico de alguma linha significa que alguma tecla foi pressionada e, então, com base na coluna habilitada e na linha, se sabe qual tecla foi pressionada. Nesse tipo de varredura é fundamental o emprego de resistores de *pull-up* ou *pull-down* nas linhas e colunas porque elas sempre têm que estar em um nível lógico conhecido. No ATmega isto é fácil pois existem resistores de *pull-up* habilitáveis em todas as portas. Assim, um possível circuito para trabalho com teclado é apresentado na Fig. 7.2. Um código exemplo com uma função para leitura desse teclado é dado a seguir.

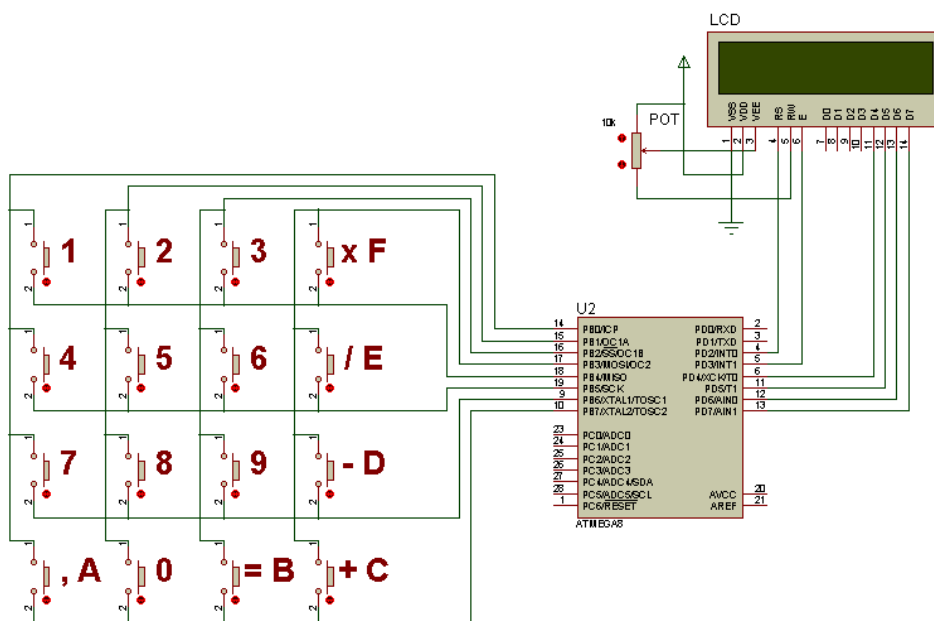


Fig. 7.2 – Teclado 4x4 controlado pelo ATmega8.

```
//===== //
//                                LEITURA DE UM TECLADO 4x4                                //
//===== //
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

//Define macros
#define set_bit(address,bit) (address|=(1<<bit))
#define clr_bit(address,bit) (address&=~(1<<bit))
#define tst_bit(address,bit) (address&(1<<bit))

//matriz com as informações para decodificação do teclado
const unsigned char teclado[4][4] PROGMEM = {{0x01, 0x02, 0x03, 0x0F},
                                              {0x04, 0x05, 0x06, 0x0E},
                                              {0x07, 0x08, 0x09, 0x0D},
                                              {0x0A, 0x00, 0x0B, 0x0C}};

unsigned char nr;
//-----
//Sub-rotina para leitura do teclado
//-----
unsigned char ler_teclado()
{
    unsigned char n, j, tecla, linha;

    for(n=0;n<4;n++)
    {
        clr_bit(PORTB,n); //apaga o bit da coluna (varredura)
        _delay_ms(10); //atraso para uma varredura mais lenta
        linha = PINB>>4; //lê o valor das linhas

        for(j=0;j<4;j++) //testa as linhas
        {
            if(!tst_bit(linha,j)) //se foi pressionada alguma tecla, decodifica
                tecla = pgm_read_byte(&teclado[j][n]);
        }
        set_bit(PORTB,n); //seta o bit zerado anteriormente
    }
    return tecla; //retorna o valor pressionado
                  //se nenhuma tecla for pressionada, tecla tem o ultimo valor lido
                  //inicialização por conta do usuário
}
//-----
int main()
```

```

{
    DDRB = 0b00001111; //definições das entradas e saídas para o teclado
    PORTB= 0xFF;        //habilita os pull-ups da porta B

    while(1)
    {
        nr = ler_teclado();    //Lê constantemente o teclado

        //aqui vai o seu código
    }
}
//=====

```

Quando se faz a varredura do teclado, é importante utilizar uma frequência adequada. Se esta for muito alta pode resultar em ruídos espúrios, se for baixa, a tecla pode ter sido solta e o sistema não detectá-la. No exemplo acima, um atraso de 10 ms é suficiente, além de fazer o *debounce*.

## Exercícios:

**7.1** – Elaborar um programa para controle de acesso por senha numérica. A senha pode conter 3 dígitos. Toda vez que uma tecla for pressionada, um pequeno alto-falante deve ser acionado. Quando a senha for correta, o relé deve ser ligado por um pequeno tempo (utilize um led de sinalização). Preveja que a senha possa ser alterada e salva na EEPROM. O circuito abaixo exemplifica o hardware de controle.

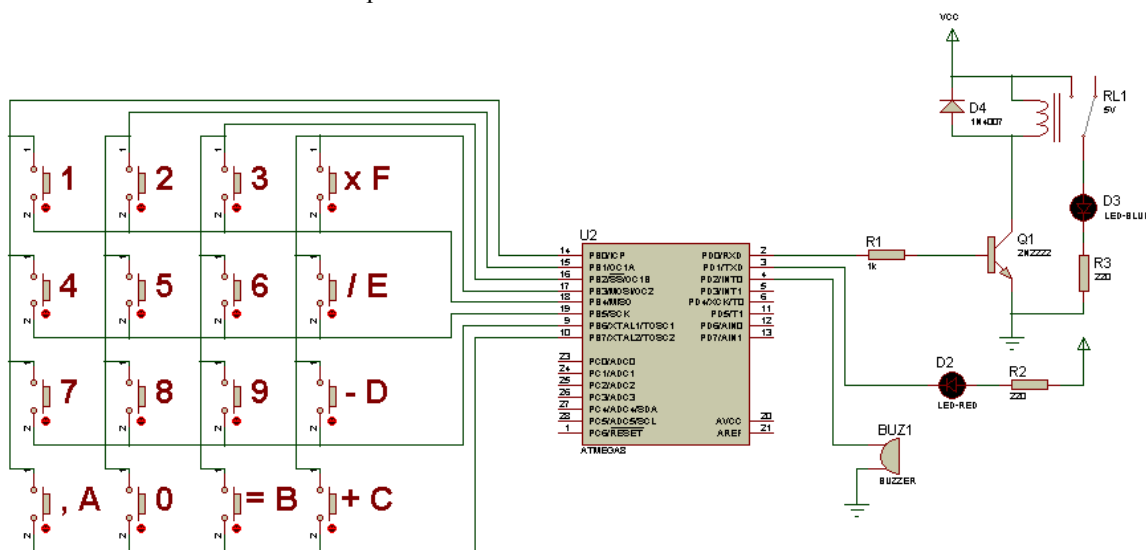


Fig. 7.3 – Controle de acesso por senha numérica.

**7.2** – Elaborar um programa para ler um teclado alfa-numérico, similar ao usado nos telefones celulares.

**7.3** – Elaborar um programa para executar as funções matemáticas básicas de uma calculadora, conforme Fig. 37. Abaixo é apresentada uma função auxiliar para converter um número decimal em seus dígitos individuais.

```

//-----
unsigned char digitos[8];          //variável global para no máximo 8 dígitos
                                   //inicialização por conta do usuário
//-----
void ident_num(unsigned int valor) //converte um nr. decimal nos seus dígitos individuais
{
    //o tipo da variável valor pode ser alterado
    unsigned short k=0;

    do{
        digitos[k] = valor%10;      //pega o resto da divisao/10 e salva no dígito correspondente
        valor /=10;                 //pega o inteiro da divisao/10
        k++;
    }while (valor!=0);
}
//-----

```

## 8. TEMPORIZADORES/CONTADORES

O ATmega8 possui dois temporizadores/contadores de 8 bits e um de 16bits.

O Temporizador/Contador 0 (T/C0) é um módulo de propósito geral de 8 bits. Suas principais características são:

- Contador simples.
- Gerador de frequência.
- Contador de eventos externos.
- *Prescaler* de 10 bits (divisor do incremento para o contador).

O Temporizador/Contador 2 (T/C2) é um módulo de propósito geral de 8 bits. Suas características principais são:

- Contador simples.
- Limpeza do temporizador quando houver uma igualdade de comparação (recarga automática).
- PWM com correção de fase e sem ruído (*glitch*).
- Gerador de frequência.
- *Prescaler* de 10 bits.
- Fontes de interrupção no estouro e na igualdade de comparação (TOV2 e OCF2).
- Permite o uso de cristal externo de 32.768 Hz para contagem precisa de 1s.

O Temporizador/Contador 1 (T/C1) permite a execução precisa de temporização, geração de onda e medição de períodos de tempo. Suas características principais são:

- PWM de 16 bits.
- Duas unidades independentes de comparação de saída.
- Duplo *buffer* comparador de saída.
- Uma unidade de captura de entrada.
- Cancelamento do ruído da entrada de captura.
- Limpeza do temporizador quando houver uma igualdade de comparação (recarga automática).
- PWM com correção de fase e sem ruído (*glitch*).
- Período do PWM variável.
- Gerador de frequência.
- Contador de eventos externos.
- Quatro fontes independentes de interrupção (TOV1, OCF1, OCF1B e ICF1).

Os registradores comuns aos 3 T/Cs são o TIMSK (*Timer/Counter Interrupt Mask Register*) e o TIFR (*Timer/Counter Interrupt Flag Register*). O primeiro habilita as interrupções individuais e o segundo possui os bits (*flags*) de sinalização das interrupções, conforme descrito abaixo:

Bit	7	6	5	4	3	2	1	0
<b>TIMSK</b>	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

**BIT 0 – TOIE0: Timer/Counter0, Overflow Interrupt Enable**

A escrita de 1 neste bit seta a interrupção do T/C0.

**BIT 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable,**

A escrita de 1 neste bit seta a interrupção do T/C1.

**BIT 6 – TOIE2: Timer/Counter2, Overflow Interrupt Enable**

A escrita de 1 neste bit seta a interrupção do T/C2.

**BIT 3 – OCIE1B: Timer/Counter1, Output B Match Interrupt Enable**

A escrita de 1 neste bit seta a interrupção por comparação de igualdade da saída B do T/C1.

**BIT 4 – OCIE1A: Timer/Counter1, Output A Match Interrupt Enable**

A escrita de 1 neste bit seta a interrupção por comparação de igualdade da saída A do T/C1.

**BIT 5 – TICIE1: Timer/Counter1, Interrupt Capture Enable**

A escrita de 1 neste bit seta a interrupção por captura do T/C1.

**BIT 7 – OCIE2: Timer/Counter2, Output Match Interrupt Enable**

A escrita de 1 neste bit seta a interrupção por comparação de igualdade da saída do T/C2.

Bit	7	6	5	4	3	2	1	0
<b>TIFR</b>	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	–	TOV0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

**BIT 0 – TOV0: Timer/Counter0, Overflow Flag**

Este bit é colocado em 1 quando um estouro do T/C0 ocorre.

**BIT 2 – TOV1: Timer/Counter1, Overflow Interrupt Enable,**

Este bit é colocado em 1 quando um estouro do T/C1 ocorre.

**BIT 6 – TOV2: Timer/Counter2, Overflow Interrupt Enable**

Este bit é colocado em 1 quando um estouro do T/C2 ocorre.

**BIT 3 – OCF1B: Timer/Counter1, Output Compare B Match Flag**

Este bit é colocado em 1 quando o valor da contagem (TCNT1) é igual ao valor do registrador de comparação de saída B (OCR1B) do T/C1.

**BIT 4 – OCF1A: Timer/Counter1, Output Compare A Match Flag**

Este bit é colocado em 1 quando o valor da contagem (TCNT1) é igual ao valor do registrador de comparação de saída A (OCR1A) do T/C1.

**BIT 5 – ICF1: Timer/Counter1, Input Capture Flag**

Este bit é colocado em 1 quando um evento de captura ocorre no pino ICP1 do T/C1.

**BIT 7 – OCF2: Timer/Counter2, Output Compare Flag**

Este bit é colocado em 1 quando o valor da contagem (TCNT2) é igual ao valor do registrador de comparação de saída (OCR2) do T/C2.

As interrupções individuais dependem da habilitação das interrupções globais pelo bit I do SREG.

## 8.1 TEMPORIZADOR/CONTADOR 0

O T/C0 é incrementado a cada ciclo de *clock*, que pode ser da fonte interna ou de uma fonte externa. O registrador de contagem TCNT0 (8 bits) pode ser acessado a qualquer momento pela CPU e uma escrita nele tem prioridade sobre as operações de contagem e limpeza (um valor novo para contagem pode ser escrito a qualquer momento). A contagem se dá de 0x00 até 0xFF, quando a contagem passa de 0xFF para 0x00 ocorre o estouro e é gerada a interrupção.

O controle da forma de operação do T/C0 é feito no registrador TCCR0 (*Timer/Counter Control Register*), conforme bits CS02:0:

Bit	7	6	5	4	3	2	1	0
<b>TCCR0</b>	–	–	–	–	–	CS02	CS01	CS00
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Tab. 8.1 – Bit dos modos de seleção para operação do T/C0.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	clk <sub>I/O</sub> /(No prescaling)
0	1	0	clk <sub>I/O</sub> /8 (From prescaler)
0	1	1	clk <sub>I/O</sub> /64 (From prescaler)
1	0	0	clk <sub>I/O</sub> /256 (From prescaler)
1	0	1	clk <sub>I/O</sub> /1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

O *prescaler* é o número de divisões para incremento do T/C, por exemplo, se ele for 8, após 8 ciclos de relógio o T/C incrementa 1 unidade. Fonte de relógio externas não tem *prescaler* e o fabricante recomenda que a máxima frequência delas seja a da fonte dividida por 2,5 (clk/2,5). Abaixo é apresentado um código exemplo empregando o relógio interno.

```
//===== //
//      HABILITANDO A INTERRUPÇÃO POR ESTOURO DO T/C0      //
//===== //
#define F_CPU 1000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

//declaração de variáveis, etc., aqui
//-----
ISR(TIMER0_OVF_vect) //interrupção do T/C0
{
    //Aqui vai o código que roda após o estouro do T/C0
}
//-----
int main()
{
    //inicializações ficam aqui

    TCCR0 = (1<<CS01) | (1<<CS00); //T/C0 com prescaler de 64, a 1 MHz gera uma interrupção
                                   //a cada 16,384 ms

    TIMSK = 1<<TOIE0; //habilita a interrupção do T/C0
```

```

sei(); //habilita interrupções globais

for(;;)
{
    //Aqui vai o código, a cada estouro do T/C0 o programa desvia para ISR((TIMER0_OVF_vect)
};
}
//=====

```

## 8.2 TEMPORIZADOR/CONTADOR 2

O T/C2 pode ser incrementado pelo *clock* interno, via *prescaler*, ou assincronamente através dos pinos TOSC1/2 (cristal de 32.768 Hz), quando o bit AS2 do registrador ASSR for colocado em 1. No modo assíncrono o T/C2 pode contar precisamente 1s funcionando como RTC (*Real Time Counter*). A contagem é realizada no registrador TCNT2.

O registrador de comparação de saída OCR2 é comparado com o TCNT2 a todo instante. O resultado da comparação pode ser usado pelo gerador de onda para gerar um sinal PWM ou uma frequência variável de saída no pino OC2 (*Output Compare Pin*).

O contador alcança o valor de topo de contagem (TOP) quando alcança o maior valor de contagem (0xFF) ou quando alcança o valor armazenado no registrador OCR2. Isto dependerá do modo de operação do T/C2, conforme descrito a seguir.

### MODO NORMAL

É o modo mais simples de operação, onde a contagem é sempre crescente e nenhuma limpeza de contagem é realizada, passando de 0xFF para 0x00 na ocorrência do estouro. Um novo valor de contagem pode ser escrito a qualquer momento no registrador TCNT2.

### MODO LIMPEZA DO CONTADOR NA IGUALDADE DE COMPARAÇÃO – CTC

No modo CTC (*Clear Timer on Compare*) o registrador OCR2 é usado para manipular a resolução do T/C2. Neste modo o contador é zerado quando o valor de TCNT2 é igual ao valor de OCR2 (o valor TOP da contagem). Isto permite o controle da frequência de comparação e simplifica a contagem de eventos externos. O diagrama de tempo do modo CTC é mostrado na Fig. 8.1. Uma interrupção pode ser gerada cada vez que o contador atinge o valor TOP.



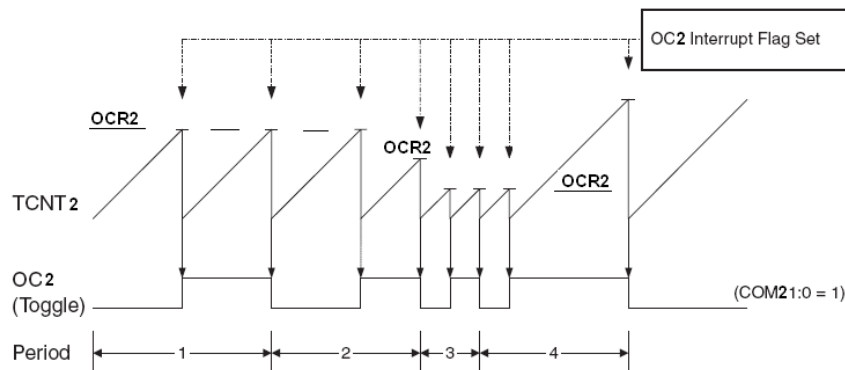


Fig. 8.1 – Modo CTC.

Para gerar uma onda de saída nesse modo, a saída OC2 pode ser ajustada para trocar de estado em cada igualdade de comparação ajustando os bits do modo de comparação de saída (COM21:0=1). Lembrando que OC2 deve ser habilitado como pino de saída. A frequência no modo CTC é definida por:

$$f_{OC2} = \frac{f_{clk}}{2N(1+OCR2)} \quad [\text{Hz}] \quad (8.1)$$

onde:  $f_{clk}$  é a frequência de operação do  $\mu$ controlador,  $OCR2$  tem um valor entre 0 e 255, e  $N$  é o fator do *prescaler* (1, 8, 32, 64, 128, 256 ou 1024).

### MODO PWM RÁPIDO

O modo de modulação por largura de pulso rápido permite a geração de um sinal PWM de alta frequência. O contador conta de zero até o valor máximo e volta ao zero. No modo de comparação de saída não-invertido, OC2 (pino) é zerado na igualdade entre TCNT2 e OCR2 e colocado em 1 no valor mínimo do contador. No modo de comparação de saída invertido o processo é inverso: OC2 é setado na igualdade e zerado no valor mínimo. A alta frequência do PWM o torna adequado para aplicações de regulação de potência, retificação e outras usando D/As. A Fig. 8.2 apresenta o diagrama de tempo para o modo PWM rápido para saída invertida e não-invertida.

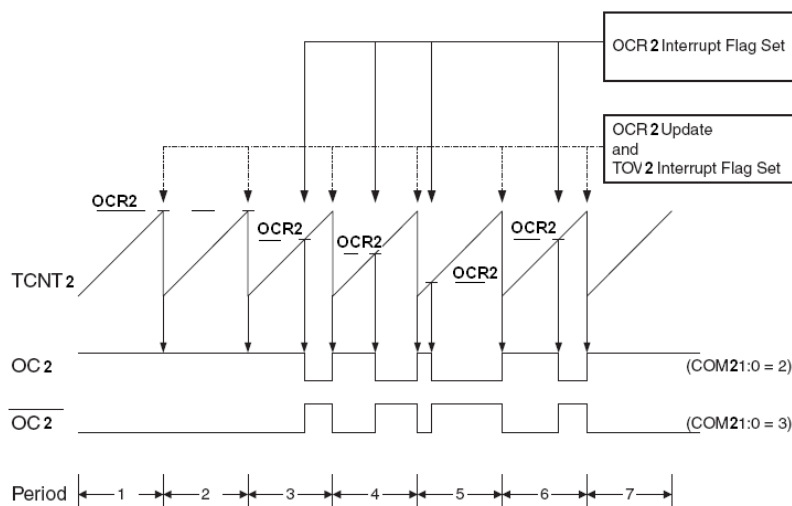


Fig. 8.2 – Modo PWM rápido.

Cada vez que o contador chega ao valor máximo de contagem, pode-se empregar a interrupção (se habilitada), atualizando o valor de comparação OCR2 e, assim, ter o PWM variável.

A frequência de saída do PWM é calculada como:

$$f_{OC2\_PWM} = \frac{f_{clk}}{256N} \quad [\text{Hz}] \quad (8.2)$$

onde:  $f_{clk}$  é a frequência de operação do  $\mu$ controlador e  $N$  é o fator do *prescaler* (1, 8, 32, 64, 128, 256 ou 1024).

Uma frequência com ciclo ativo do PWM de 50% pode ser conseguida fazendo com que OC2 mude seu estado em cada igualdade de comparação (COM21:0=1). A máxima frequência é obtida quando OCR2 é zero ( $f_{OC2} = f_{clk}/2$ ).

### MODOS PWM COM FASE CORRIGIDA

O modo PWM com fase corrigida permite gerar sinais PWM com alta resolução de fase. É baseado na contagem crescente e decrescente do TCNT2, que conta repetidamente do mínimo para o máximo e vice-versa, a resolução do PWM é de 8 bits. No modo de comparação de saída não invertido, o pino OC2 é zerado na igualdade entre TCNT2 e OCR2 quando a contagem é crescente e colocado em 1 quando a contagem é decrescente. No modo de comparação invertido, o processo é contrário. Como o processo de comparação ocorre em duas contagens, a frequência é menor que a do modo anterior do T/C2. Devido a sua característica simétrica, o modo PWM com correção de fase é preferido nas aplicações para controle de motores. A Fig. 8.3 apresenta o diagrama de tempo para o modo PWM com fase corrigida para saída invertida e não-invertida.

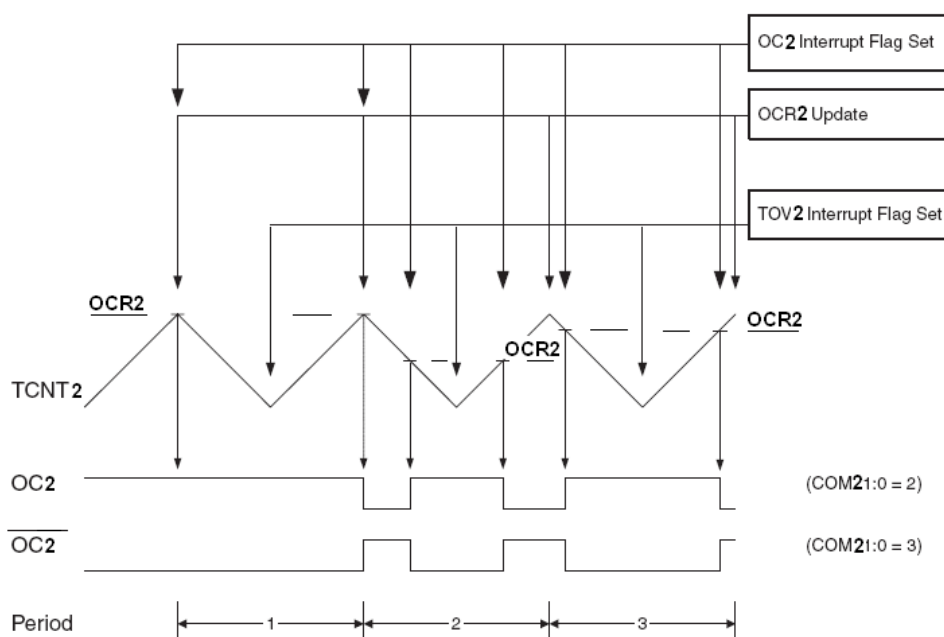


Fig. 8.3 – Modo PWM com correção de fase.

A frequência de saída do PWM é calculada como:

$$f_{OC2\_PWM} = \frac{f_{clk}}{510N} \quad [\text{Hz}] \quad (8.3)$$

onde:  $f_{clk}$  é a frequência de operação do  $\mu$ controlador e  $N$  é o fator do *prescaler* (1, 8, 32, 64, 128, 256 ou 1024).

## REGISTRADORES PARA TRABALHO COM O T/C2

O controle do modo de operação do T/C2 é feito no registrador TCCR2 (*Timer/Counter Control Register 2*).

Bit	7	6	5	4	3	2	1	0
<b>TCCR2</b>	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

### BIT 7 – FOC2: *Force Output Compare*

O bit FOC2 está ativo somente quando os bits WGM especificam um modo que não seja PWM. Deve ser colocado em zero quando operando em algum modo PWM. Colocado em 1, é forçada uma comparação no módulo gerador de onda. A saída OC2 é alterada de acordo com os bits COM21:0.

### BIT 6, 3 – WGM21:0: *Wave Form Generation Mode*

Estes bits controlam a sequência de contagem do contador, a fonte do valor máximo para contagem (TOP) e o tipo de forma de onda a ser gerada, conforme Tab. 8.2.

Tab. 8.2 – Bits para configurar o modo de operação do T/C2.

Mode	WGM21	WGM20	Timer/Counter Mode of Operation	TOP	Update of OCR2	TOV2 Flag Set
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR2	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

### BIT 5,4 – COM21:0: *Compare Match Output Mode*

Estes bits controlam o comportamento do pino OC2. Se um ou ambos bits forem colocados em 1, a funcionalidade normal do pino é alterada. Entretanto, o correspondente bit do registrador DDR deve estar ajustado para habilitar o *drive* de saída. A funcionalidade dos bits COM21:0 depende do ajuste dos bits WGM21:0. As Tabs. 8.3-5 mostram as possíveis configurações.

Tab. 8.3 – Modo CTC (não PWM).

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected.
0	1	Toggle OC2 on Compare Match
1	0	Clear OC2 on Compare Match
1	1	Set OC2 on Compare Match

Tab. 8.4 – Modo PWM rápido.

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected.
0	1	Reserved
1	0	Clear OC2 on Compare Match, set OC2 at BOTTOM, (non-inverting mode)
1	1	Set OC2 on Compare Match, clear OC2 at BOTTOM, (inverting mode)

Tab. 8.5 – Modo PWM com fase corrigida.

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected.
0	1	Reserved
1	0	Clear OC2 on Compare Match when up-counting. Set OC2 on Compare Match when downcounting.
1	1	Set OC2 on Compare Match when up-counting. Clear OC2 on Compare Match when downcounting.

**BIT 2:0 – CS22:0: Clock Select**

Bits para seleção da fonte de *clock* para o T/C2, conforme Tab. 8.6.

Tab. 8.6 – Seleção do *clock* para o T/C2.

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$\text{clk}_{T2S}/(\text{No prescaling})$
0	1	0	$\text{clk}_{T2S}/8$ (From prescaler)
0	1	1	$\text{clk}_{T2S}/32$ (From prescaler)
1	0	0	$\text{clk}_{T2S}/64$ (From prescaler)
1	0	1	$\text{clk}_{T2S}/128$ (From prescaler)
1	1	0	$\text{clk}_{T2S}/256$ (From prescaler)
1	1	1	$\text{clk}_{T2S}/1024$ (From prescaler)

**Operação Assíncrona do T/C2**

O oscilador do T/C2 para sinal externo é otimizado para o cristal de relógio de 32.768 Hz. Outro sinal de relógio aplicado ao pino TOSC1 pode resultar em operação incorreta. A frequência da CPU deve ser pelo menos 4 vezes maior que a frequência do oscilador (>131 kHz). Existe uma série de detalhes que devem ser considerados: quando se escreve nos registradores do T/C2; quando se entra em modos de economia de energia e quando se altera entre a operação síncrona e assíncrona (ver folha de dados do componente).

Para habilitar o uso desse cristal externo, é preciso ajustar o bit AS2 (*Asynchronous Timer/Counter2*) do registrador ASSR (*Asynchronous Status Register*).

## Programas Exemplos do uso do T/C2

```
//===== //
//      T/C2 COM CRISTAL EXTERNO DE 32,768kHz      //
//      Rotina para implementar um relógio e piscar um led      //
//===== //
#include <avr/io.h>
#include <avr/interrupt.h>

#define cpl_bit(adress,bit) (adress^=(1<<bit))      //complementa o valor do bit
#define LED PB0

unsigned char segundos, minutos, horas; //variáveis com os valores para o relógio
//precisam ser tratadas em um código adequado
//-----
ISR(TIMER2_OVF_vect)//entrada aqui a cada 1s
{
    cpl_bit(PORTB,LED);//pisca LED

    //rotina para contagem das horas, minutos e segundos

    segundos++;

    if(segundos==60)
    {
        segundos=0;
        minutos++;

        if (minutos==60)
        {
            minutos=0;
            horas++;

            if (horas==24)
                horas=0;
        }
    }
}
//-----
int main()
{
    //inicializacoes vao aqui
    DDRB = 0xFF;      //crítal externo, não importa como configura os pinos TOSC1 e TOSC2
    PORTB= 0xFF;

    ASSR = 1<<AS2;      //habilita o cristal externo para o contador de tempo real
    TCCR2 = (1<<CS22)|(1<<CS20); //prescaler = 128, resultando em um estouro a cada 1s
    TIMSK = 1<<TOIE2;      //habilita interrupção do T/2

    sei();      //habilita interrupção global

    while(1)
    {
        //código principal (display, ajuste de hora, etc..)
    };
}
//=====

//===== //
//      PWM COM FASE CORRIGIDA      //
//===== //
#include <avr/io.h>
#include <avr/interrupt.h>
//-----
int main()
{
    DDRB = 1<<PB3;      //pino OC2 como saída do sinal PWM
    PORTB= 0xFF;

    TCCR2 = (1<<WGM20)|(1<<COM21)|(1<<CS02); //PWM com correção de fase, prescaler=1 (liga T/C2)
    //limpa OC2 na igualdade quando a contagem é crescente
    //seta OC2 na igualdade quando a contagem é decrescente
    OCR2 = 0xAB;      //no registrador OCR2 se ajusta o valor do ciclo ativo do PWM.

    for(;;);
}
//=====
```

## Exercício:

**8.1**– Elaborar um programa para controlar, empregando um sinal PWM, um motor DC com 256 níveis de velocidade utilizando o circuito da Fig. 8.4. O nível da velocidade selecionado deve ser apresentado no display e armazenado na memória EEPROM.

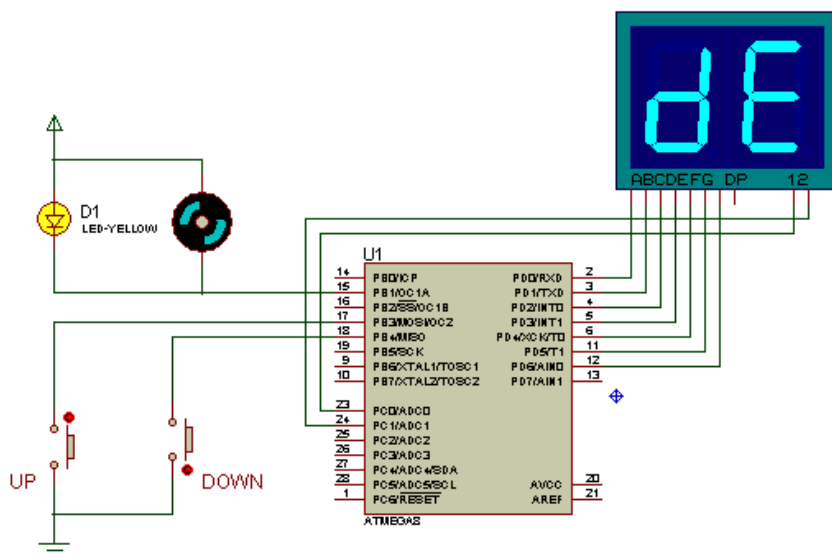


Fig. 8.4 – Controle de um motor DC com um sinal PWM.

## 8.3 TEMPORIZADOR/CONTADOR 1

O T/C1 é um contador de 16 bits com contagem no par de registradores TCNT1H e TCNT1L (contador TCNT1), e que possui dois registradores de controle: TCCR1A e TCCR1B. O T/C1 é incrementado via *clock* interno com ou sem *prescaler* ou por *clock* externo ligado ao pino T1. Os registradores de comparação de saída OCR1A e OCR1B são constantemente comparados com o valor de contagem. O resultado da comparação pode ser usado para gerar sinais PWM ou com frequência variável nos pinos de saída de comparação (OC1A/OC1B). O registrador de captura de entrada pode capturar o valor do contador em qualquer evento externo que ocorrer no pino ICP1 ou nos pinos do comparador analógico. A entrada de captura inclui um filtro digital para filtrar eventuais ruídos.

O T/C1 conta de 0x0000 até 0xFFFF ou até o valor de topo (TOP) que pode ser definido nos registradores OCR1A, ICR1 ou por um conjunto fixo de valores (0x00FF, 0x01FF ou 0x3FF). Quando o valor de OCR1A é usado como TOP para o modo PWM, o registrador OCR1A não pode ser utilizado para gerar uma saída PWM (o valor de topo pode ser alterado a qualquer momento). Se o valor de topo for fixo, o registrador ICR1 pode ser empregado alternativamente liberando OCR1A para gerar uma saída PWM.

O TCNT1 pode ser escrito a qualquer momento pelo programa e tem prioridade sobre a contagem e a limpeza do contador. A sequência de contagem é determinada pelos bits do modo de geração de onda WGM13:0 localizados nos registradores de controle TCCR1A/B. Há uma relação muito próxima na forma como o contador conta e como as onda são geradas nos pinos de comparação de saída OC1A/B.

Os modos de operação do T/C1 são definidos pela combinação dos bits do modo gerador de forma de onda (WGM13:0) e os bits do modo de comparação de saída (COM1x1:0, x = A ou B). Os bits do modo de comparação de saída não afetam a sequência de contagem enquanto os bits do modo gerador de forma de onda o fazem. Os bits COM1x1:0 ajustam a saída PWM (invertida ou não). Para os modos não-PWM, esses bits controlam se a saída deve ser zerada, colocada em um ou alterada numa igualdade de comparação.

### MODO DE CAPTURA DE ENTRADA

O modo captura de entrada permite medir o período de um sinal digital externo. Lê-se o valor do T/C1 quando ocorrer a interrupção; após duas interrupções, é possível calcular o tempo decorrido entre elas. O programa deve lidar com estouros do T/C1 se ocorrerem. No modo de captura de entrada se escolhe qual o tipo de borda do sinal gerará a interrupção (descida ou subida). Para medir o tempo em que um sinal permanece em alto ou baixo é preciso alterar o tipo de borda desse sinal após cada captura. O filtro contra ruído de entrada do T/C1 pode ser habilitado ajustando-se o bit ICNC1 do registrador TCCR1B.

### MODO NORMAL

É o modo simples de operação do contador (WGM13:0=0). Neste modo a contagem é sempre crescente e nenhuma limpeza do contador é realizada. O contador conta de 0 até 65535 e volta a zero, gerando um estouro e um pedido de interrupção.

### MODO LIMPEZA DO CONTADOR NA IGUALDADE DE COMPARAÇÃO – CTC

No modo CTC (WGM13:0 = 4 ou 12) os registradores OCR1A ou ICR1 são utilizados para mudar a resolução do contador (valor de topo). Neste modo, o contador é limpo (zerado) quando o valor do TCNT1 é igual ao OCR1A (WGM13:0=4) ou igual ao ICR1 (WGM13:0=12). Este modo permite grande controle na comparação para a frequência de saída e também simplifica a operação de contagem de eventos externos. O diagrama temporal do modo CTC é apresentado na Fig. 8.5.

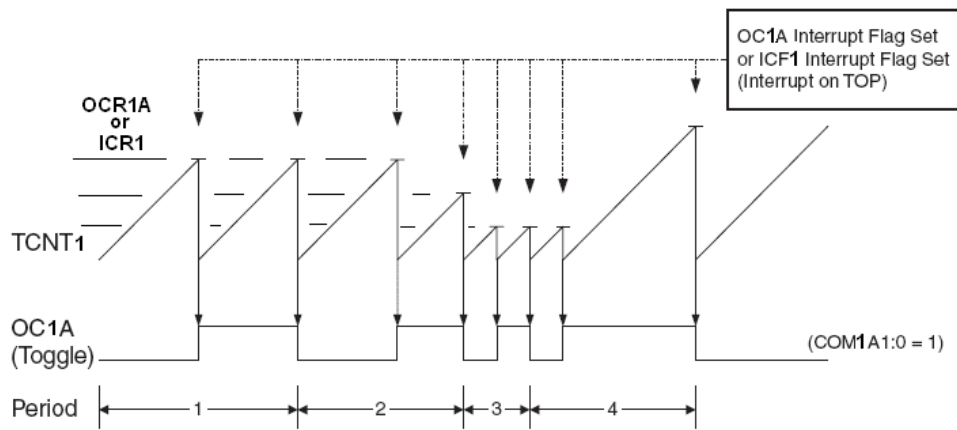


Fig. 8.5 – Diagrama de tempo para o modo CTC.

Para gerar uma onda de saída no modo CTC, o pino OC1A de saída pode ser ajustado para trocar de nível em cada igualdade de comparação (COM1A1:0=1). O valor de OC1A não será visível no pino, a menos que a direção do pino seja configurada como saída (DDR\_OC1A = 1). A máxima frequência que a forma de onda pode ter é metade da frequência de trabalho da CPU. A frequência da forma de onda é definida por:

$$f_{OC1A} = \frac{f_{clk}}{2N(1+OCR1A)} \quad [Hz] \quad (8.4)$$

onde:  $f_{clk}$  é a frequência de operação do  $\mu$ controlador,  $OCR1A$  tem um valor entre 0 e 65535, e  $N$  é o fator do *prescaler* (1, 8, 64, 256 ou 1024).

### MODO PWM RÁPIDO

A modulação por largura de pulso rápida ou modo PWM rápido (WGM13:0=5, 6, 7, 14 ou 15) permite a geração de um PWM de alta frequência. O contador conta de 0 até o valor de topo e volta a zero. No modo de comparação de saída não invertido, a saída OC1x é limpa na igualdade entre TCNT1 e OCR1x e colocado em 1 no valor 0. No modo de comparação de saída invertido o processo é contrário.

A resolução para o PWM rápido pode ser fixa para 8, 9, 10 bits ou definida pelo registrador ICR1 ou OCR1A. A resolução mínima é 2 bits (ICR1 ou OCR1A = 3) e a máxima é 16 bits (ICR1 ou OCR1A = 65535). A resolução do PWM pode ser calculada como:

$$R_{PWM\_Rápido} = \frac{\log(TOP + 1)}{\log(2)} \quad [bits] \quad (8.5)$$

Nesse modo o contador é incrementado até o contador encontrar um dos valores fixos 0x00FF, 0x01FF ou 0x03FF (WGM13:0=5,6 ou 7), o valor do ICR1 (WGM13:0=14), ou o valor de OCR1A (WGM13:0=15). O diagrama de tempo para o PWM rápido é mostrado na Fig. 8.6.



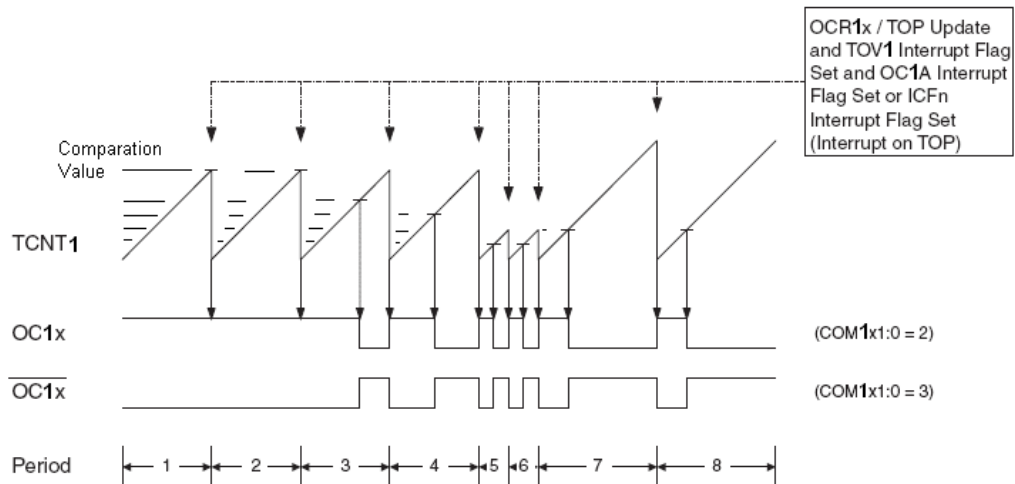


Fig. 8.6 – Diagrama temporal para o modo PWM rápido.

A frequência de saída do PWM rápido pode ser calculada com:

$$f_{OC1x\_PWM} = \frac{f_{clk}}{N.(1+TOP)} \quad [Hz] \quad (8.6)$$

onde:  $f_{clk}$  é a frequência de operação do  $\mu$ controlador e  $N$  é o fator do *prescaler* (1, 8, 64, 256 ou 1024). Uma frequência com ciclo ativo de 50% pode ser conseguida ajustando OC1A para mudar de nível lógico em cada igualdade de comparação (COM1A1:0=1). Isto implica que somente OCR1A é usado para definir o valor TOP (WGM13:0=15). A frequência máxima gerada será a metade da frequência de trabalho da CPU quando OCR1A = 0.

### MODO PWM COM FASE CORRIGIDA

O modo PWM com fase corrigida (WGM13:0=1, 2, 3, 10 ou 11) permite gerar um PWM com fase precisa. O contador conta repetidamente de 0 até o valor TOP e então do TOP até o zero. No modo de comparação de saída não invertida, OC1x é limpo na igualdade entre TCNT1 e OCR1x na contagem crescente e colocado em 1 na contagem decrescente. No modo com saída não invertida o processo é contrário. A resolução do PWM é a mesma do modo PWM rápido (Eq. 8.5).

No modo PWM com fase corrigida o contador incrementa até encontrar um dos valores fixos 0x00FF, 0x01FF ou 0x03FF (WGM13:0=1, 2 ou 3), o valor do ICR1 (WGM13:0=10), ou o valor de OCR1A (WGM13:0=11). Quando o contador atinge o valor de topo, ele muda a direção de contagem (permanece 1 ciclo de clock no valor TOP). O diagrama de tempo para o PWM com correção de fase é mostrado na Fig. 8.7, quando OCR1A e ICR1 definem o valor de topo.

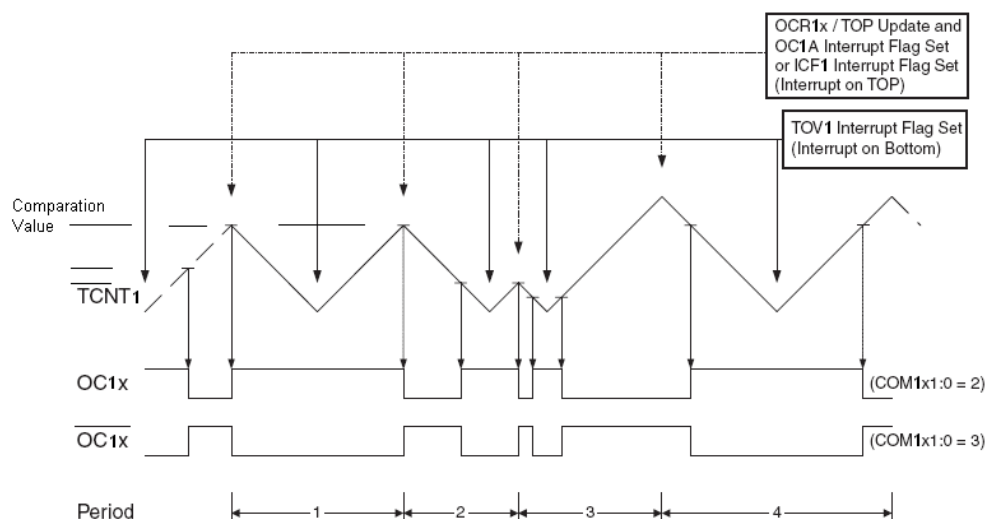


Fig. 8.7 – Diagrama de tempo para o PWM com fase corrigida.

A frequência de saída do PWM com fase corrigida pode ser calculada com:

$$f_{OC1x\_PWM} = \frac{f_{clk}}{2N.TOP} \quad [Hz] \quad (8.7)$$

onde:  $f_{clk}$  é a frequência de operação do  $\mu$ controlador e  $N$  é o fator do *prescaler* (1, 8, 64, 256 ou 1024). Se OCR1A definir o valor TOP (WGM13:0=11) e COM1A1:0 = 1, a saída OC1A terá um ciclo ativo de 50%.

### MODOS PWM COM FASE E FREQUÊNCIA CORRIGIDA

Neste modo, o PWM apresenta uma alta resolução de frequência e fase (WGM13:0=8 ou 9). É similar ao modo PWM com fase corrigida. A diferença principal é o tempo de atualização do registrador OCR1x (ver Fig. 8.7 e 8.8). A resolução em bits é igual ao modo PWM rápido (Eq. 8.5), e o cálculo da frequência é igual ao modo PWM com fase corrigida (Eq. 8.7).

No modo PWM com fase e frequência corrigida o contador incrementa até encontrar o valor do ICR1 (WGM13:0=8), ou o valor de OCR1A (WGM13:0=9). Quando o contador atinge o valor de topo, ele muda a direção de contagem (permanece 1 ciclo de clock no valor TOP). O diagrama de tempo para o PWM com correção de fase e frequência é mostrado na Fig. 8.8, quando OCR1A e ICR1 definem o valor de topo.

Se OCR1A definir o valor TOP (WGM13:0=9) e COM1A1:0 = 1, a saída OC1A terá um ciclo ativo de 50%.

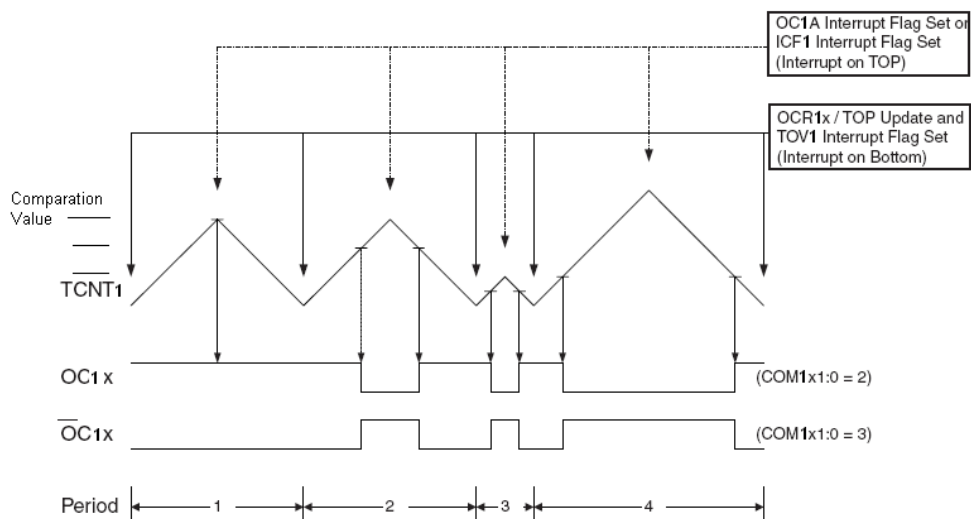


Fig. 8.8 – Diagrama temporal para o PWM com fase e frequência corrigidos.

## REGISTRADORES PARA TRABALHO COM O T/C1

O controle do modo de operação do T/C1 é feito no registrador TCCR1A e TCCR1B (*Timer/Counter Control Register*).

Bit	7	6	5	4	3	2	1	0
<b>TCCR1A</b>	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

**BIT 7, 6 – COM1A1:0: Compare Output Mode for channel A**

**BIT 5, 4 – COM1B1:0: Compare Output Mode for channel B**

COM1A1:0 e COM1B1:0 controlam o comportamento dos pinos OC1A e OC1B, respectivamente. Se o valor 1 for escrito nesses bits a funcionalidade dos pinos é alterada (os bits correspondentes no registrador DDR devem ser colocado em 1s para habilitar o *drive* de saída). Quando OC1A e OC1B forem empregados, a funcionalidade dos bits COM1x1:0 dependerá do ajuste dos bits WGM13:0. As Tabs. 8.7-9 apresentam as configurações para os bits COM1x1:0 para os diferentes modos de operação do T/C1.

Tab. 8.7– Modo não PWM (normal e CTC).

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on Compare Match
1	0	Clear OC1A/OC1B on Compare Match (Set output to low level)
1	1	Set OC1A/OC1B on Compare Match (Set output to high level)

Tab. 8.8 – Modo PWM rápido.

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM, (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM, (inverting mode)

Tab. 8.9 – Modo PWM com correção de fase e correção de fase e frequência.

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 9 or 14: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match when up-counting. Set OC1A/OC1B on Compare Match when downcounting.
1	1	Set OC1A/OC1B on Compare Match when up-counting. Clear OC1A/OC1B on Compare Match when downcounting.

**BIT 3 – FOC1A: Force Output Compare for channel A****BIT 2 – FOC1B: Force Output Compare for channel B**

Os bits FOC1A/B só estão ativos quando os bits WGM13:0 especificam um modo não PWM.

**BIT 1, 0 – WGM11:0: Waveform Generation Mode**

Combinado com os bits WGM13:2 do registrador TCCR1B, esses bits controlam a forma de contagem do contador, a fonte para o valor máximo (TOP) e qual tipo de forma de onda gerada será empregada, ver Tab. 8.10.

Tab. 8.10 – Descrição dos bits para os modos de geração de formas de onda.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation <sup>(1)</sup>	TOP	Update of OCR1x	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

### Registrador de controle TCCR1B

Bit	7	6	5	4	3	2	1	0
<b>TCCR1B</b>	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

#### BIT 7 – ICNC1: *Input Capture Noise Canceler*

Colocando este bit em 1, o filtro de ruído do pino de captura ICP1 é habilitado. Este filtro requer 4 amostras sucessivas iguais para o ICP1 mudar sua saída. Assim, a captura de entrada é atrasada por 4 ciclos de *clock*.

#### BIT 6 – ICES1: *Input Capture Edge Select*

Este bit seleciona qual borda no pino de entrada de captura (ICP1) será usado para disparar o evento de captura (ICES1=0 na transição de 1 para 0, ICES1=1 na transição de 0 para 1). Quando uma captura ocorre o valor do contador é copiado no registrador ICR1.

#### BIT 4, 3 – WGM13:2: *Waveform Generation Mode*

Ver Tab. 8.10.

#### BIT 2:0 – CS12:0: *Clock Select*

Existem 3 bits para a escolha do *clock* para o T/C1, ver Tab. 8.11.

Tab. 8.11 – Descrição dos bits para seleção do *clock* para o T/C1.

CS12	CS11	CS10	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$\text{clk}_{I/O}/1$ (No prescaling)
0	1	0	$\text{clk}_{I/O}/8$ (From prescaler)
0	1	1	$\text{clk}_{I/O}/64$ (From prescaler)
1	0	0	$\text{clk}_{I/O}/256$ (From prescaler)
1	0	1	$\text{clk}_{I/O}/1024$ (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Os outros registradores importantes são os de 16 bits, compostos por uma parte alta e uma parte baixa. No uso na programação em linguagem C não existe a preocupação da forma de acesso a essas partes, sendo acessadas com um único registrador. Assim, tem-se TCNT1 (*Timer/Counter1*), OCR1A (*Output Compare Register 1A*), OCR1B (*Output Compare Register 1B*), e ICR1 (*Input Capture Register 1*), que são compostos respectivamente pelos registradores de 8 bits: TCNT1H – TCNT1L, OCR1AH-OCR1AL, OCR1BH-OCR1BL e ICR1H-ICR1L.

## Programas Exemplo do uso do T/C1 como gerador de onda

```
//=====
//          T/C1 MODO CTC - Gerando uma onda no pino OC1A
//=====
#include <avr/io.h>
//-----
int main()
{
    DDRB = 0b0000010;    //pino OC1A como saída
    TCCR1A = 1<<COM1A0;   //modo CTC, mudança do nível lógico na igualdade de comparação
                        //entre TCNT1 e OCR1A
    TCCR1B = (1<<WGM12)|(1<<CS10); //CTC com carga pelo OCR1A e sem prescaler
    OCR1A = 0x1FFF;       //frequência do sinal de saída é alterada aqui. Com este valor = 61Hz

    while(1);
}
//=====
```

## Exercícios:

**8.2** – Elaborar um programa para controlar o motor de passo unipolar da Fig. 8.9. Dois botões controlam a direção de rotação e outros dois a velocidade.

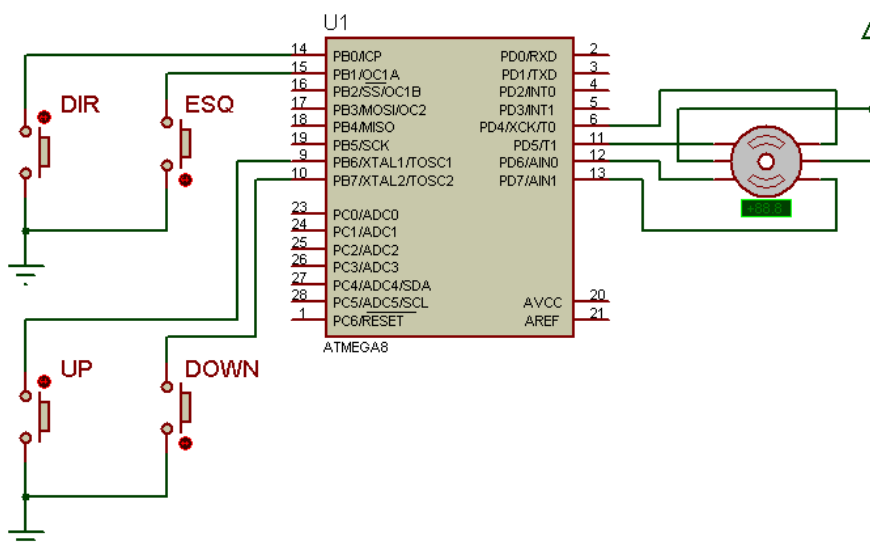


Fig. 8.9 – Controle de um motor de passo unipolar.

**8.3** – Elaborar um programa para que o ATmega8 funcione como frequencímetro digital, apresentando o valor lido em um LCD 16x2. Empregue um circuito adequado para a simulação.

**8.4** – Utilizando o modo de geração de frequência do ATmega8 faça um programa para gerar as notas musicais básicas de um piano (1 oitava). Empregue um circuito adequado para a simulação, incluindo teclas.

**8.5** – Como pode ser gerada um pequena música pelo ATmega8?

## 9. MULTIPLEXAÇÃO (VARREDURA DE DISPLAYs)

Quando são necessários vários pinos de I/O para acionamento de um determinado circuito e o  $\mu$ controlador não os dispõe, é fundamental o emprego da multiplexação: técnica para transitar com vários dados em uma mesma via, ou barramento. A multiplexação também é empregada para diminuir o número de vias e a complexidade física das placas de circuito impresso.

As melhores multiplexações empregam o mínimo número possível de componentes externos para cumprir as funções que devem ser desempenhadas pelo hardware. Uma dessas tarefas é o acionamento de vários displays de 7 segmentos, conforme exemplificado pela Fig. 9.1.

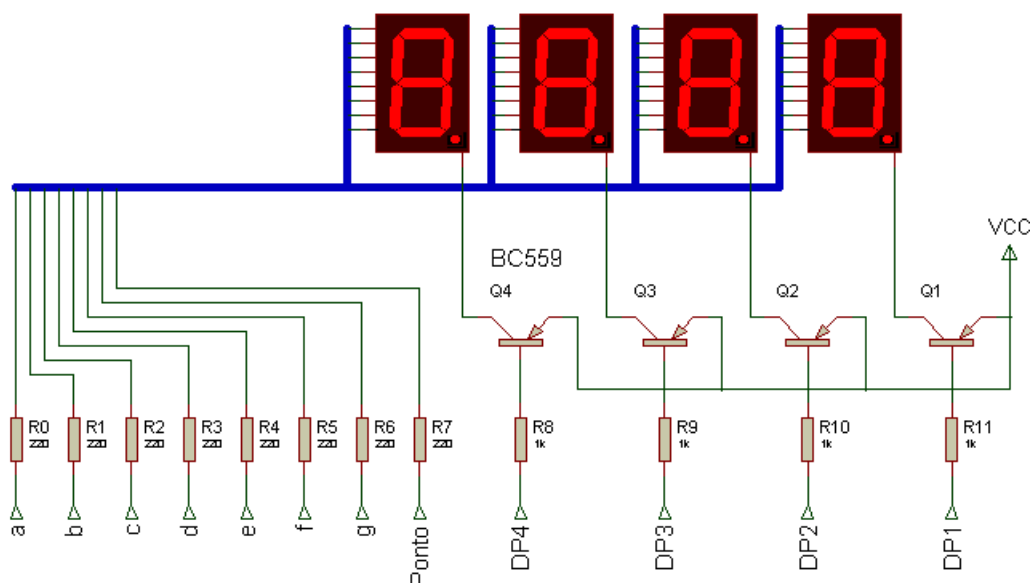


Fig. 9.1 - Acionamento de 4 displays de 7 segmentos.

Para ligar diretamente 4 displays de 7 segmentos, seriam necessários  $8 \times 4 = 32$  pinos do  $\mu$ controlador (considerando o ponto do display). O problema, além do grande número de pinos, seria a corrente máxima que poderia chegar a 320 mA, no caso de 10 mA por segmento.

Uma característica da visão humana, chamada persistência, pode ser usada aqui. Os olhos são incapazes de sentir a alteração de um sinal luminoso com frequência acima de 24 Hz, isto é, se um led piscar 24 vezes em um segundo, ele aparecerá aos olhos humanos como ligado constantemente. Este é o princípio utilizado na varredura pelos elétrons nos tubos de raios catódicos das antigas TVs e faz com que a velocidade de filmagem e projeção seja padronizada em 24 fotogramas por segundo.

Num sistema  $\mu$ controlado, a varredura de displays da Fig. 9.1 seria feita da seguinte maneira:

1. Os displays estão apagados, os transistores estão desabilitados.
2. O dado referente ao primeiro display é colocado no barramento.
3. O display referente ao dado é ligado, o transistor adequado é acionado.
4. Espera-se o tempo necessário para o acionamento do próximo display
5. Apaga-se o display ligado.

6. Coloca-se o dado referente ao próximo display da sequência (no caso da esquerda para a direita).
7. O display é ligado, o transistor adequado é acionado.
8. O processo se repete para cada display.

Para que a persistência da visão funcione, o tempo de varredura (tempo ligado de cada display) deve ser de aproximadamente 10 ms ( $24 \text{ Hz} \times 4 = 96 \text{ Hz}$ ,  $T \cong 10 \text{ ms}$ ).

Na programação do  $\mu$ controlador, a forma mais elegante de apresentar a mensagem num conjunto de displays é fazer a varredura dentro da rotina de interrupção de algum *timer*, liberando o programa principal para outras atividades e simplificando o código.

## Exercícios:

- 9.1** – Elaborar um programa para que o hardware da Fig. 9.2 funcione como relógio 24 h. A entrada de sinal para contagem dos segundos é de 60 Hz. O ajuste do horário deve ser feito nos botões específicos.

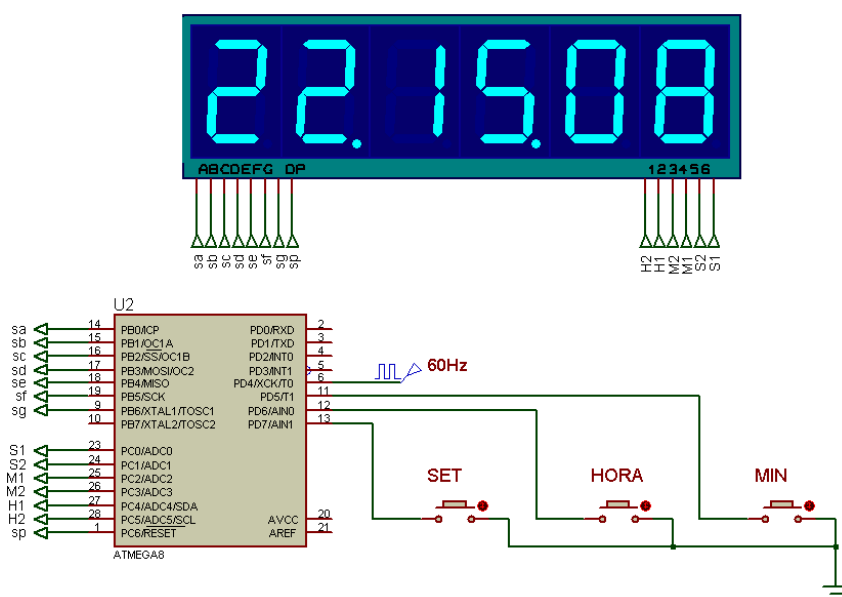


Fig. 9.2 – Relógio com contagem baseado na frequência da rede elétrica.

- 9.2** – Um sistema muito comum para divulgação visual de mensagens é a matriz de leds, onde um conjunto organizado de leds, formando um painel de pixel (1 Led = 1 pixel), é comandado por um sistema  $\mu$ controlado. A quantidade de informação para formar uma imagem exige a multiplexação de dados e conseqüentemente algum sistema de varredura é empregado. A Fig. 9.3 apresenta um sistema para o controle de uma matriz de 8x24 leds (192). O sistema alimenta uma linha por vez. Os dados correspondentes a cada coluna, incluindo qual linha será alimentada, são fornecidos por um conjunto de registradores de deslocamento (*shift register* – 4094, conversor serial-paralelo). Para controlar todo o sistema bastam somente 3 saídas do  $\mu$ controlador!

- \* Faça um programa para apresentar uma mensagem estática na matriz de leds da Fig. 9.3.
- \* Como o programa pode apresentar mensagens em movimento?

Obs.: o circuito da Fig. 9.3 é apenas para simulação, faltam os *drivers* para fornecer a corrente adequada aos leds.



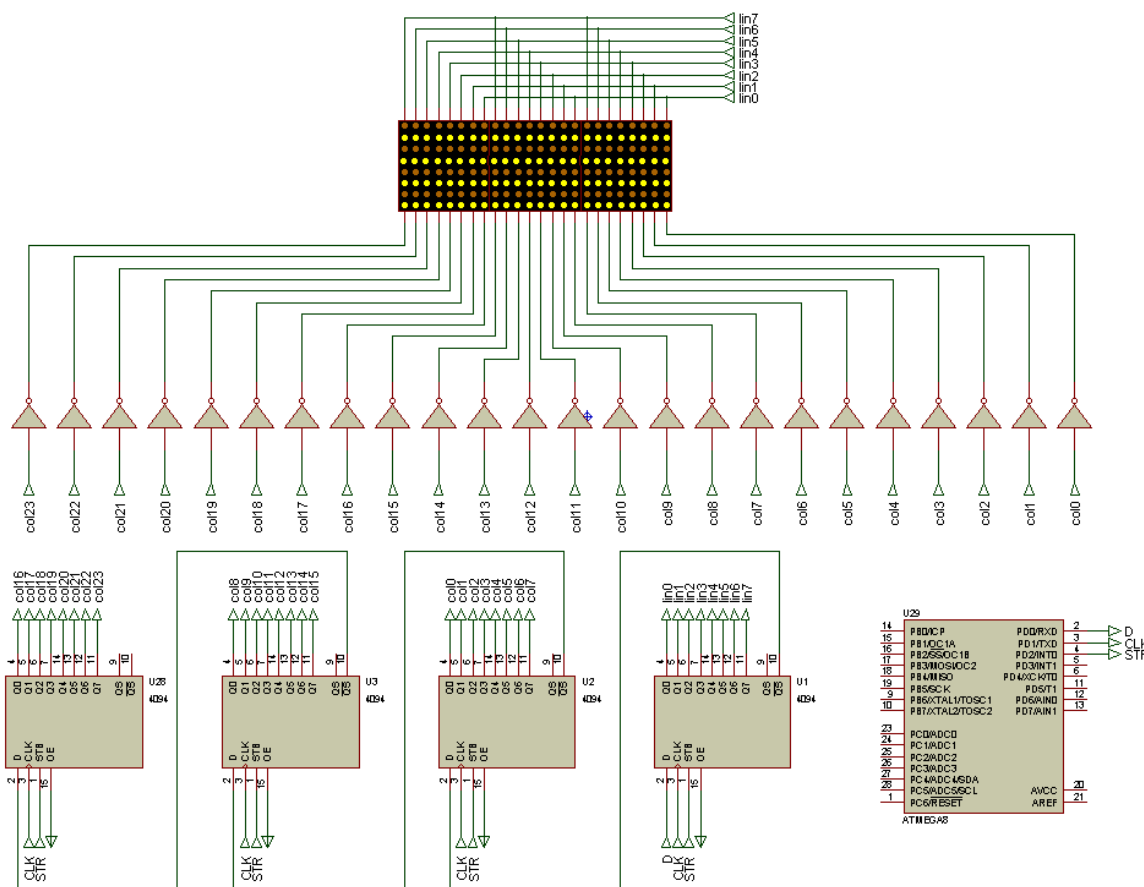


Fig. 9.3 – Matriz de leds e o 4094.

**9.3** – Baseado no processo de multiplexação projete um CLP (Controlador Lógico Programável) básico com 32 saídas e 32 entradas digitais empregando o ATmega8 (circuito para simulação).

**9.4** – Baseado em dois displays de 7 segmentos e um botão, desenvolva um programa e circuito de simulação para o sorteio de números de 1 até 60 (Mega Sena). O número sorteado não deve voltar ao sorteio.

## 10. DISPLAY GRÁFICO (128x64 pontos)

Um display gráfico é uma matriz de pontos visíveis pela aplicação de uma tensão elétrica sobre o cristal líquido de cada um desses pontos (pixel). Este tipo de LCD é empregado para representar os mais diversos tipos de caracteres e figuras, cuja definição dependerá da quantidade de pontos que o LCD pode representar.

Um tipo de LCD gráfico muito utilizado com  $\mu$ controladores é o de 128x64 pontos, baseado nos CIs controladores KS108B e KS107 ou similares. Os pontos não são ligados todos ao mesmo tempo, sendo a varredura do display realizada automaticamente pelo circuito de controle. A Fig. 10.1 apresenta o diagrama do circuito de controle com esses CIs. O display necessita de uma fonte de alimentação negativa para o contraste do LCD e vários modelos possuem internamente o circuito que gera esta tensão (Fig. 10.1a). Entretanto, existem LCDs que necessitam de uma fonte negativa externa (Fig. 10.1b).

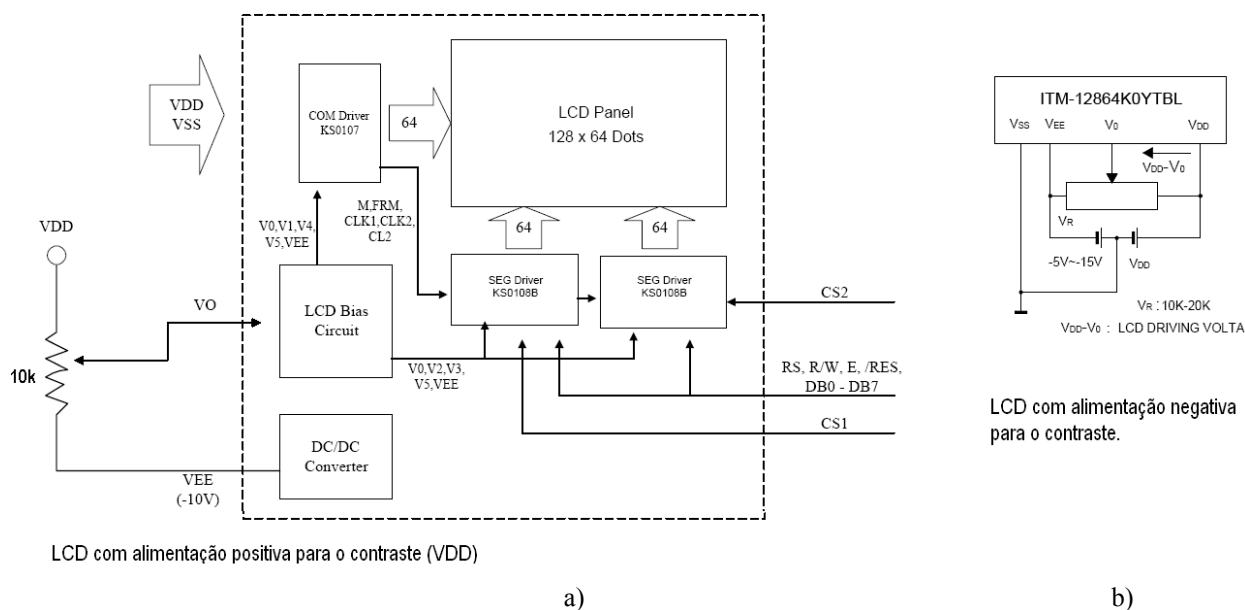


Fig. 10.1 – Diagrama esquemático de um LCD de 128x64.

A descrição dos pinos de um LCD gráfico de 128x64 pontos é apresentada na Tab. 10.1. Os pinos CS1 e CS2 podem ser ativos com diferentes níveis lógicos, dependendo do fabricante, bem como os pinos podem ter diferentes posições incluindo ou não pinos de *backlight*.

Tab. 10.1 – Pinagem de um LCD gráfico de 128x64 pontos.

ITEM	SYMBOL	DESCRIPTION
1	VSS	Ground
2	VDD	Power Supply for Logic
3	V0	Operating Voltage for LCD(Variable)
4	D/I	Data/Instruction Select Signal
5	R/W	Read/Write Select Signal
6	E	Enable Signal
7	DB0	Data bus [0~7]. There state I/O common terminal.
8	DB1	
9	DB2	
10	DB3	
11	DB4	
12	DB5	
13	DB6	
14	DB7	
15	CS1	Chip Select Signal IC 1
16	CS2	Chip Select Signal IC 2
17	RST	Reset Signal
18	VEE	Power Supply Voltage for LCD
19	A	Anode of LED
20	K	Cathode of LED

O LCD gráfico possui uma memória RAM, na qual basta escrever a informação que se deseja apresentar, sendo a metade do display controlada por um KS108B e a outra metade por outro. As posições de escrita seguem o padrão apresentado na Fig. 10.2. As metades do LCD são separadas em 8 páginas, sendo cada página composta por 64 bytes. Cada metade do LCD é controlada individualmente pelos pinos CS1 e CS2. As colunas da esquerda para a direita são numeradas respectivamente de 0 até 63. Uma imagem completa no LCD consumirá 1 kbyte de informação,

$(128 \text{ bits} \times 64 \text{ bits})/8 = 1024 \text{ bytes}$ . A escrita no LCD é feita por bytes e não se pode ligar pontos individuais diretamente.

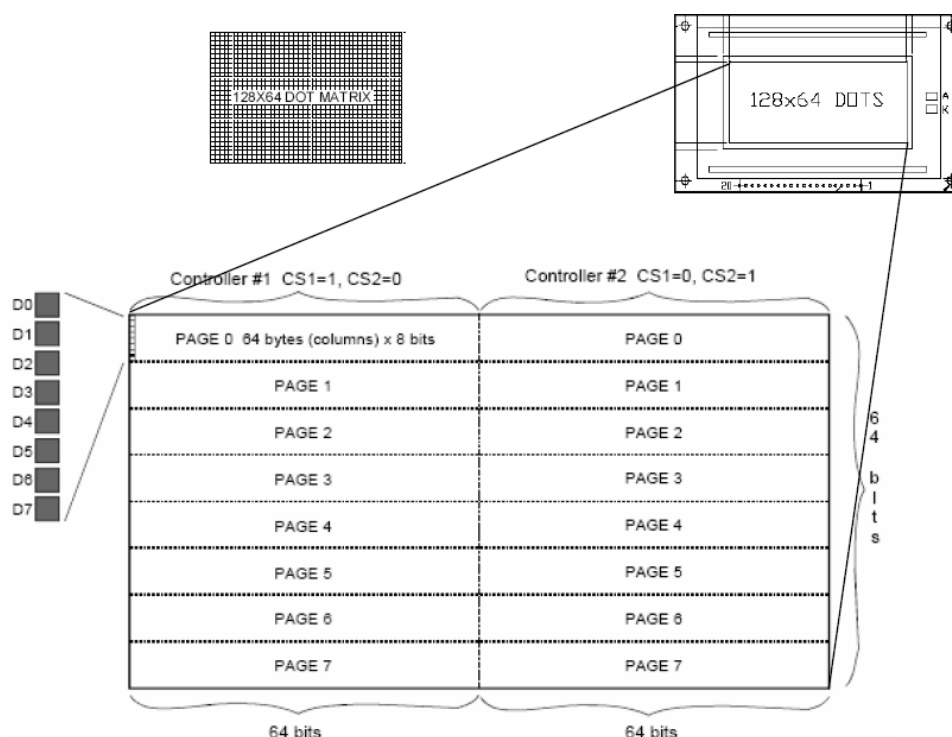


Fig. 10.2 – Organização da memória de pontos do LCD gráfico.

O correto funcionamento do LCD depende da inicialização do circuito de controle. O *reset* pode ser realizado externamente por uma rede RC ou controlado diretamente pelo  $\mu$ controlador. Os dados são transferidos após um pulso de *enable*. Para o correto funcionamento, os tempos de resposta de cada modelo de LCD devem ser consultados no catálogo do fabricante. Na Tab. 10.2 são apresentadas as instruções de controle do LCD.

Tab. 10.2 – Instruções de controle para o CI KS108B.

Instruction	D/I	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Function
Display ON/OFF	L	L	L	L	H	H	H	H	H	L/H	Controls the display on or off. Internal status and display RAM data is not affected. L: OFF, H: ON
Set Address	L	L	L	H	Y address (0~63)						Sets the Y address in the Y address counter.
Set Page (X address)	L	L	H	L	H	H	H	Page (0~7)			Sets the X address at the X address register.
Display Start Line	L	L	H	H	Display start line (0~63)						Indicates the display data RAM displayed at the top of the screen.
Status Read	L	H	B U S Y	L	O N / O F F	R E S E T	L	L	L	L	Read status. BUSY L: Ready H: In operation ON/OFF L: Display ON H: Display OFF RESET L: Normal H: Reset
Write Display Data	H	L	Write Data								Writes data (DB0:7) into display data RAM. After writing instruction, Y address is increased by 1 automatically.
Read Display Data	H	H	Read Data								Reads data (DB0:7) from display data RAM to the data bus.

## MODO DE OPERAÇÃO

1. O controle do LCD é realizado entre a habilitação e desabilitação do mesmo (pulso de *enable*). Se CS1 ou CS2 não estiverem ativos, instruções de entrada ou saída de dados não podem ser executadas.
2. Registradores de entrada no LCD armazenam dados temporariamente até serem escritos na sua RAM. Quando CS1 ou CS2 estiverem ativos, R/W e D/I selecionam o registrador de entrada.
3. Registradores de saída armazenam temporariamente os dados da RAM. Quando CS1 ou CS2 estiverem ativos e for realizada uma operação de leitura, o bit de *status* de ocupado (*busy*) pode ser lido. Para ler o conteúdo da RAM, são necessárias duas operações de leitura, uma para colocar o dado no registrador de saída e outro para lê-lo.
4. O sistema pode ser inicializado colocando o pino de *reset* em nível baixo durante a energização do circuito (pode ser feito por software). No *reset*, o display é desligado e ajustada a linha de início de escrita para a posição zero. Somente a instrução de leitura pode ser realizada (teste do bit de ocupado (*busy flag*) DB7 e de *reset* DB4).
5. O bit de ocupado (*busy flag*) indica se o KS108B está ocupado ou não. Quando estiver em nível alto o KS108B está realizando alguma operação interna, quando em nível baixo o KS108B aceita dados ou instruções. Em escritas rápidas do LCD, este *flag* precisa ser monitorado, caso contrário, apenas se emprega um atraso adequado na escrita de um novo dado.
6. O display pode ser ligado e desligado, este processo apenas desliga a alimentação do cristal líquido, o conteúdo da RAM permanece inalterado.
7. O registrador de página X indica qual página de dados da RAM será acessada. Cada troca de página precisa ser realizada por software.
8. O contador de endereço Y indica qual coluna será escrita. É incrementado automaticamente após cada escrita ou leitura, da esquerda para a direita. Detalhe: o posicionamento de escrita não segue o sistema cartesiano, aqui o Y representaria o *x* do sistema cartesiano e a página X representa o *y* (o valor 7 seria a origem).
9. A memória RAM do display indica o estado de cada ponto da matriz de pontos. Um bit igual a 1 indica que o ponto estará ligado; para se apagar o ponto, o bit correspondente deve ser zerado.
10. O registrador de linha de início indica que os dados da memória RAM devem ser apresentados na linha superior do LCD. É empregado quando se deseja rotacionar alguma figura do LCD.

A Fig. 10.3 indica o processo de escrita do dado 0xAB (0b10101011) em uma coluna do LCD. Primeiramente, o LCD deve ser inicializado; o display é *resetado*, ligado com CS1 ou CS2 habilitados, escolhe-se a coluna de escrita e a página. Após o dado ser colocado no barramento, avisa-se que será realizada uma operação de escrita e que o valor do barramento é um dado. Então, aplica-se um pulso de *enable* para a transferência dos dados.

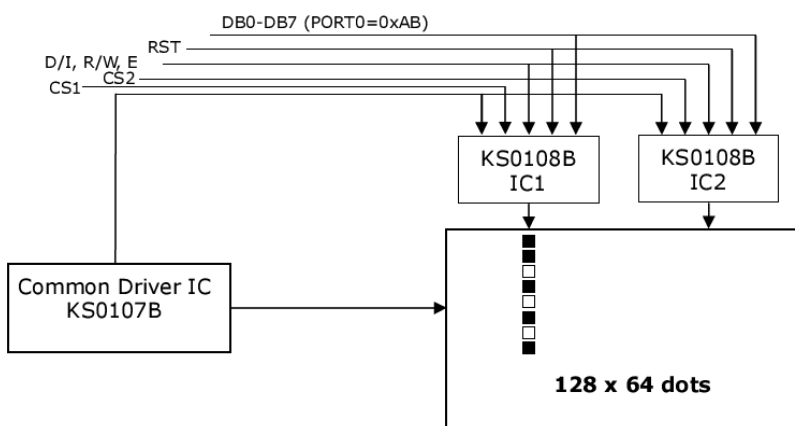


Fig. 10.3 – Escrita do dado 0xAB no LCD (fora de escala).

A Fig. 10.4 apresenta a alteração de um único ponto do LCD. Para tal, primeiro é necessário uma leitura da posição que se deseja alterar, salvando a informação do byte cuja posição será escrita e então, alterando-se somente o bit desejado desse byte e reescrevendo-o na posição lida. No exemplo da Fig. 10.4 o bit DB2 foi alterado.

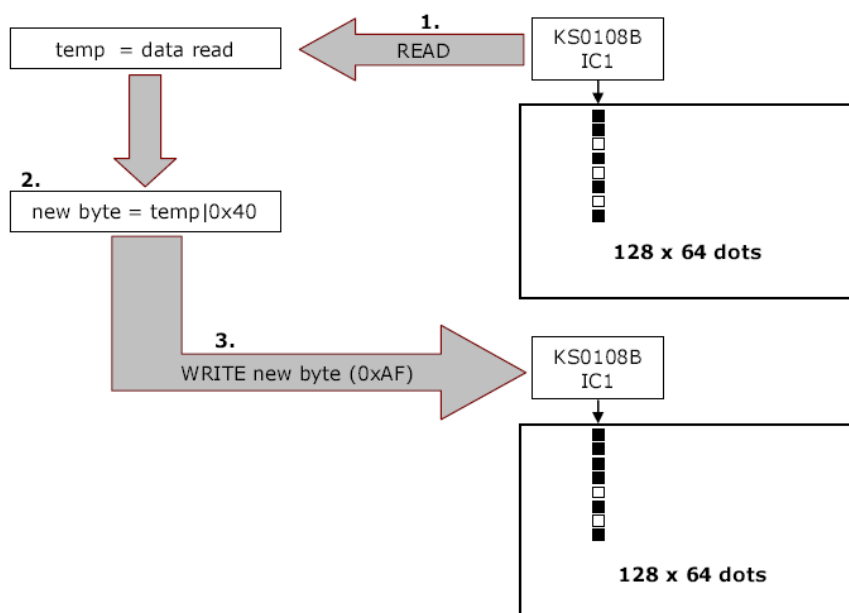


Fig. 10.4 – Escrevendo em um único ponto do LCD.

A seguir, é apresentado um código simples para escrita em um LCD de 128x64 empregando o ATmega. Não é apresentada uma função de leitura do LCD. O código apenas ilustra o emprego dos pinos de controle do LCD.

Rotinas elaboradas permitem escrever caracteres em qualquer posição do display. Programas gratuitos permitem converter figuras em tabela de dados que podem ser acessados diretamente pelo programa para envio ao display (por exemplo, *FastLCD* e *Bitmap2LCD*). O único trabalho é adequar a tabela criada com a forma de escrita da memória RAM do LCD.

```
//===== //
//          DISPLAY GRÁFICO 128x64 PONTOS          //
//===== //
#include <avr/io.h>
#include <util/delay.h>

#define      set_bit(address,bit) (address|=(1<<bit))
#define      clr_bit(address,bit) (address&=~(1<<bit))
#define      tst_bit(address,bit) (address&(1<<bit))

#define DADOS  PORTD

#define CS1    PC0          //ativo em alto
#define CS2    PC1          //ativo em alto
#define DI     PC2          //DI=1 dados, DI=0 instrucao
#define RW     PC3          //RW=0 escrita, RW=1 leitura
#define EN     PC4          //ativo em alto
#define RST    PC5          //ativo em baixo

#define LIGA_LCD  0x3F      //codigos de instrucao
#define DESL_LCD  0x3E
#define LIN_INIC  0xC0
#define Y_INIC    0x40
#define PAG_INIC  0xB8

#define set_escrita  clr_bit(PORTC,RW)      //facilitando a escrita do codigo
#define set_leitura  set_bit(PORTC,RW)
#define set_dado     set_bit(PORTC,DI)
#define set_instrucao clr_bit(PORTC,DI)
#define set_CS1      set_bit(PORTC,CS1)
#define set_CS2      set_bit(PORTC,CS2)
#define clr_CS1      clr_bit(PORTC,CS1)
#define clr_CS2      clr_bit(PORTC,CS2)
#define set_enable   set_bit(PORTC,EN)
#define clr_enable   clr_bit(PORTC,EN)
//-----
void reset_LCDG();
void enable_LCDG();
void escreve_LCDG(unsigned char dado, unsigned char coluna, unsigned char pagina);
//-----
int main()
{
    DDRC = 0xFF;          //ajustando os pinos como saída, somente escrita
    PORTD = 0xFF;
    PORTC = 0b00101000;    //CS1 e CS2 = 1, habilitados. Reset ativo e EN = 0.

    set_instrucao;//instrucao
    set_escrita;

    reset_LCDG();

    DADOS = LIGA_LCD;
    enable_LCDG();

    //DADOS = LIN_INIC;
    //enable_LCDG();

    DADOS = Y_INIC;
    enable_LCDG();

    DADOS = PAG_INIC;
    enable_LCDG();

    escreve_LCDG(0xAA, 125, 2);//exemplo da escrita de 0xAA na coluna 125 (60 CS2), pagina 2

    //seu codigo vai aqui
}
//-----
void reset_LCDG()      //reset
{
    clr_bit(PORTC,RST);
    _delay_ms(1);
    set_bit(PORTC,RST);
    _delay_ms(1);
}
//-----
void enable_LCDG()     //pulso de enable
{
    _delay_us(3);       //ajustar este tempo na prática ou ler o busy flag para escrita rápida
    set_enable;
}
```

```

    _delay_us(3);
    clr_enable;
}
//-----
void escreve_LCDG(unsigned char dado, unsigned char coluna, unsigned char pagina)
{
    set_instrucao;          //instrucao

    if (coluna>63)           //coluna 0-127
    {
        clr_CS1;
        set_CS2;
        DADOS = Y_INIC + coluna - 64;
    }
    else
    {
        clr_CS2;
        set_CS1;
        DADOS = Y_INIC + coluna;
    }

    enable_LCDG();
    DADOS = PAG_INIC + pagina; //pagina 0 - 7
    enable_LCDG();

    set_dado;                //escreve dado
    DADOS = dado;
    enable_LCDG();
}
//=====

```

## Exercício:

10.1 – Elaborar um programa para gerar uma animação gráfica com 5 quadros para o circuito da Fig. 10.5a.

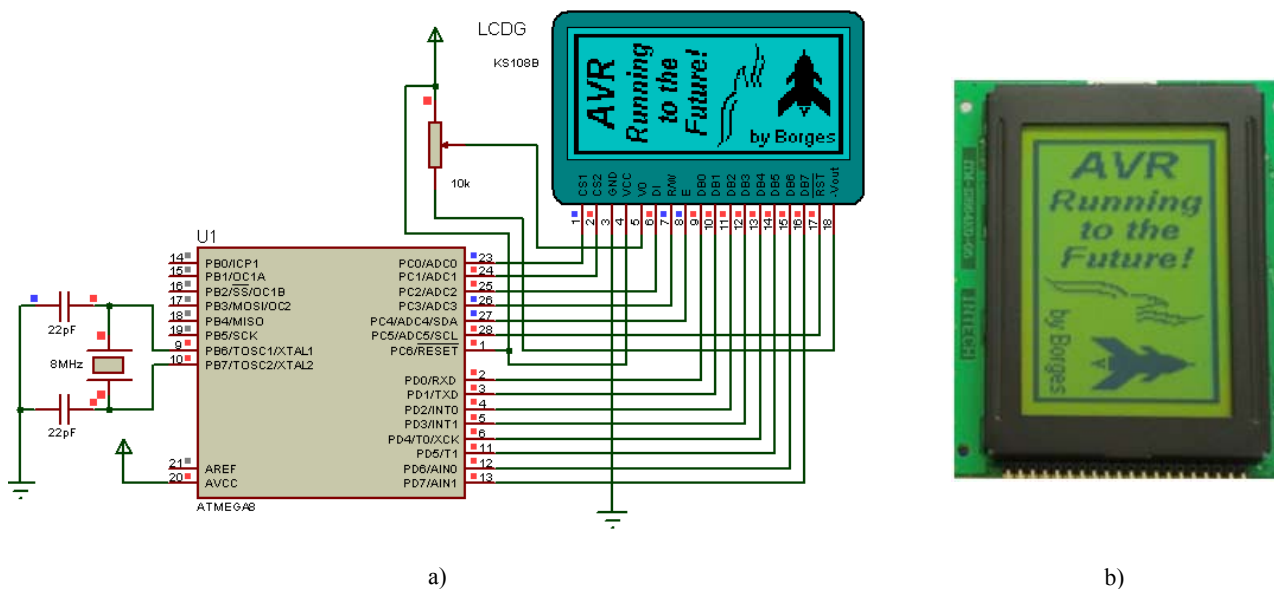


Fig. 10.5 – a) circuito para simulação; b) exemplo prático para 1 quadro.

## 11. GERANDO FORMAS DE ONDA

### 11.1 DISCRETIZANDO UM SINAL

Para transformar um sinal analógico em digital tanto o tempo quanto a amplitude devem ser discretizados, o que se chama amostragem e quantização, respectivamente. Para a amostragem, define-se um período (intervalo de tempo entre cada amostra) e, conseqüentemente, uma frequência de amostragem (número de amostras por segundo). Para um sinal senoidal, por exemplo:

Sinal Contínuo:

$$y(t) = A \cdot \text{seno}(2 \cdot \pi \cdot f \cdot t) \quad [\text{s}] \quad (11.1)$$

Sinal Discreto:

$$Y[n] = A \cdot \text{seno}(2 \cdot \pi \cdot f \cdot n \cdot \Delta t), \quad (11.2)$$

onde,  $A$  = amplitude do sinal,  $f$  = frequência do sinal,  $t$  = tempo,  $n$  = índice do vetor (inteiro positivo, 0, 1, ... N) e  $\Delta t = 1/fs$ , ( $fs$  é a frequência de amostragem do sinal - *sampling frequency*). A frequência de amostragem deve ser no mínimo duas vezes maior que a maior frequência presente no sinal a ser amostrado (Teorema de Nyquist).

A Fig. 11.1 ilustra a amostragem de um sinal senoidal representado por:

$$y(t) = 3 \cdot \text{seno}(2 \cdot \pi \cdot 1 \cdot t)$$

Na Fig. 11.1a tem-se uma frequência de amostragem de 10 Hz (10 amostras por segundo), na Fig. 11.1b, essa frequência é de 20 Hz. Considerando  $n$  começando em zero, tem-se, respectivamente, 11 e 21 amostras no intervalo de tempo de 1 s (um período de amostragem do sinal). As equações que resultariam nos pontos amostrados da figura 11.1 são representadas por:

$$Y_a[n] = 3 \cdot \text{seno}(2 \cdot \pi \cdot n \cdot 0,1)$$

$n = 0, 1, 2, \dots 10$ . O que resulta em:

$$Y_a = [0 \ 1,76 \ 2,85 \ 2,85 \ 1,76 \ 0 \ -1,76 \ -2,85 \ -2,85 \ -1,76 \ 0]$$

e

$$Y_b[n] = 3 \cdot \text{seno}(2 \cdot \pi \cdot n \cdot 0,05)$$

$n = 0, 1, 2, \dots 20$ . O que resulta em:

$$Y_b = [0 \ 0,93 \ 1,76 \ 2,43 \ 2,85 \ 3 \ 2,85 \ 2,43 \ 1,76 \ 0,93 \ 0 \ -0,93 \ -1,76 \ -2,43 \ -2,85 \ -3 \ -2,85 \ -2,43 \ -1,76 \ -0,93 \ 0]$$

Se for usado um conversor analógico-digital, o valor é mantido constante na saída do mesmo até que um novo valor seja convertido (operação de amostragem e retenção - *sample and hold*). O valor é quantizado de acordo com o número de bits de resolução do A/D.



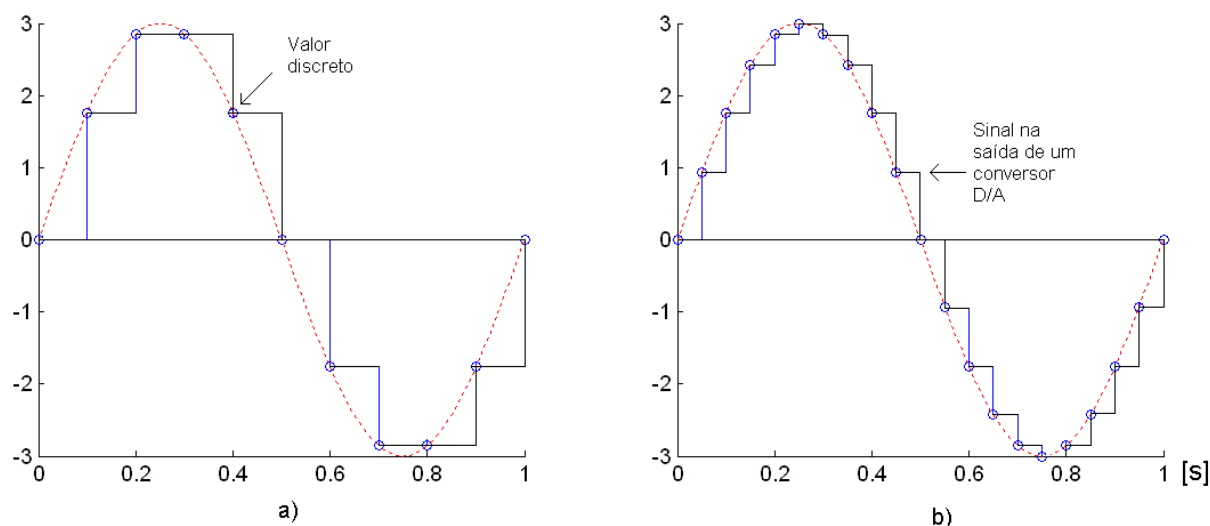


Fig. 11.1 – Amostragem de um sinal senoidal de 1 Hz: a)  $f_s = 10$  Hz e b)  $f_s = 20$  Hz.

Em sistemas  $\mu$ controlados, informações referentes aos pontos de uma forma onda podem ser armazenadas em memória. Após, os valores desses pontos podem ser enviados, a uma taxa pré-definida e na sequência adequada, a um conversor digital-analógico para a recomposição da forma de onda.

### Exercício:

**11.1** – Elaborar um programa para gerar 3 formas de onda diferentes: senoidal, triangular e dente de serra, de acordo com o circuito da Fig. 11.2. A frequência do sinal pode ser alterada por botões.

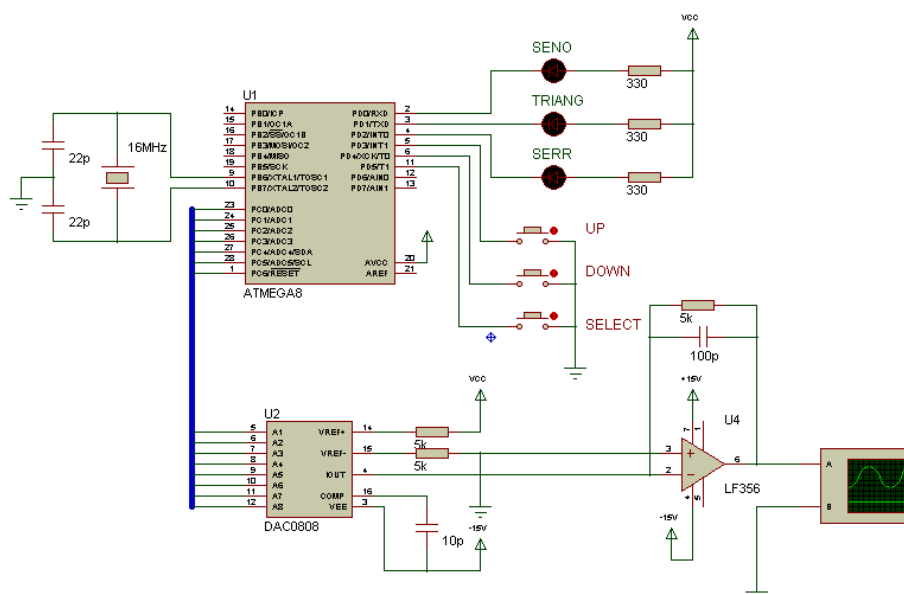


Fig. 11.2 – Gerador de funções.

## 11.2 MODULAÇÃO POR LARGURA DE PULSO – PWM

O PWM é baseado no conceito de valor médio de uma forma de onda periódica. Digitalmente, o valor médio de uma forma de onda é controlado pelo tempo em que o sinal fica em nível lógico alto durante um determinado intervalo de tempo. No PWM esse tempo é chamado de ciclo ativo (*Duty Cycle*). A Fig. 11.3 apresenta formas de onda PWM em que, a largura do pulso (*Duty Cycle*) é variável de 0 até 100%, com incremento de 25%.

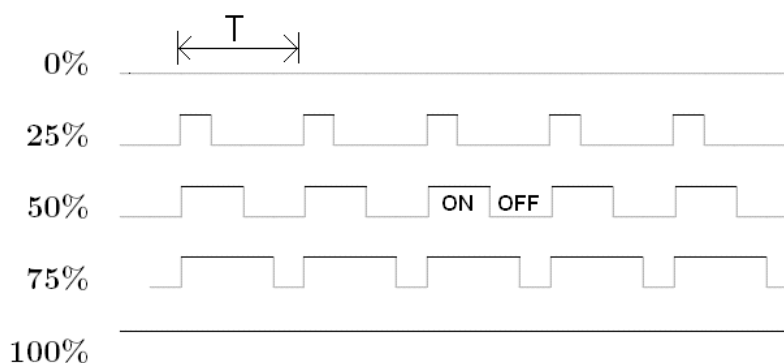


Fig. 11.3 – PWM com período T e ciclo ativo de 0%, 25%, 50%, 75% e 100%.

O cálculo do valor médio de um sinal digital é dado por:

$$V_{\text{médio}} = \frac{\text{Amplitude Máxima}}{\text{Período}} \cdot (\text{Tempo Ativo no Período}) \quad (11.3)$$

Assim, se o sinal digital tem variação de 0 a 5 V, um ciclo ativo de 50% corresponde a um valor médio de 2,5 V, enquanto um ciclo ativo de 75% corresponderia a 3,75 V. Ao se alterar o ciclo útil do PWM, altera-se o seu valor médio. A Fig. 11.4 ilustra a variação de um PWM digital de 0 a 100% do seu ciclo útil com o passar do tempo.

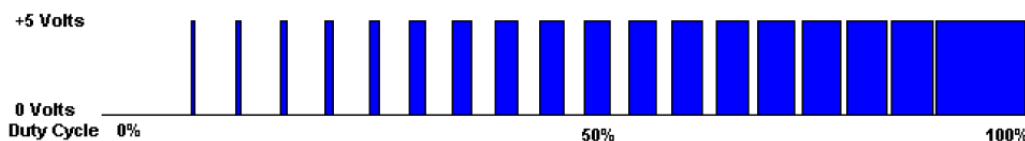


Fig. 11.4 – Variação de um PWM em um intervalo de tempo.

A resolução do PWM em um  $\mu$ controlador é dada em bits e indica quantos diferentes ciclos ativos podem ser gerados, por exemplo, um PWM de 10 bits pode gerar 1024 larguras diferentes de pulsos.

Baseado no valor médio de um sinal PWM, muitos dispositivos eletrônicos podem ser desenvolvidos, entre os quais pode-se citar: controle de motores, brilho de lâmpadas – leds, fontes chaveadas e circuitos inversores. Por exemplo, através de um PWM pode-se controlar uma chave transistorizada empregando-se um filtro adequado, ver Fig. 11.5.

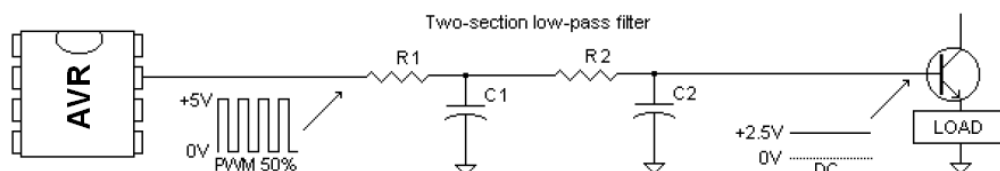


Fig. 11.5 – Chave transistorizada controlada por PWM.

### Conversor D/A com PWM

O valor médio de um sinal PWM pode ser obtido com o emprego de um filtro. Isto permite criar um conversor digital/analógico simples, de baixa precisão. Os circuitos da Fig. 11.5 ou 11.6 podem ser empregados para esta função.

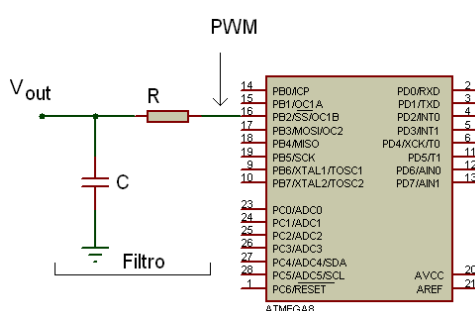


Fig. 11.6 – Conversor DA empregando PWM.

O circuito acima utiliza um filtro passa baixa de primeira ordem, cuja frequência de corte é dada por:

$$f_c = \frac{1}{2\pi R C} \quad (11.4)$$

Importante que a frequência de corte seja pelo menos 10 vezes maior que a frequência do sinal PWM:

$$f_c > \frac{f_{PWM}}{10} \quad (11.5)$$

A obtenção de um sinal filtrado com pouca distorção pode ser conseguida com o uso de filtros de ordem superior, empregando-se para tal, um número adequado de elementos passivos (resistores, capacitores e/ou indutores) ou ativos (amplificadores).

### Exercícios:

**11.2** – Elaborar um circuito para simulação que funcione como um conversor A/D para um sinal PWM. Desenvolva um programa para teste.

**11.3** - Baseado no conceito de valor médio, empregue o PWM do ATmega 8 para criar um sinal senoidal com frequência de 10 Hz. Obs.: para uma precisão adequada, a frequência do sinal PWM deve ser no mínimo 10 vezes maior que a frequência do sinal gerado.

Como alterar a frequência do sinal gerado?

**11.4** - Como se pode controlar motores monofásicos e trifásicos com sinais PWM (inversores de frequência)?

## 12. SPI

A Interface Serial Periférica (SPI – *Serial Peripheral Interface*) permite grande velocidade (até  $f_{osc}/2$ ) na transferência síncrona de dados entre o ATmega8 e outros CIs ou entre AVR's. A SPI do ATmega8 inclui as seguintes características:

- *Full-duplex* (envia e recebe dados ao mesmo tempo), transferência síncrona de dados com 3 fios;
- Operação Mestre ou Escravo;
- Escolha do bit a ser transferido primeiro: bit mais significativo (MSB) ou menos significativo (LSB);
- Várias taxas programáveis de bits;
- Sinalizador de interrupção no final da transmissão;
- Sinalizador para proteção de colisão de escrita;
- *Wake-up* do modo *Idle*;
- Velocidade dupla no modo Master ( $clk/2$ ).

A conexão entre mestre e escravo é apresentada na Fig. 12.1. O sistema é composto por dois registradores de deslocamento e gerador mestre de *clock*. O mestre SPI inicializa o ciclo de comunicação quando coloca a pino  $\overline{SS}$  (*Slave Select*) em nível baixo do desejado escravo. Mestre e escravo preparam o dado a ser enviado nos seus respectivos registradores de deslocamento, e o mestre gera os pulsos necessários de *clock* no pino SCK para a troca de dados (um bit é enviado e um bit é recebido ao mesmo tempo). Os dados são sempre deslocados do mestre para o escravo no pino MOSI (*Master Out – Slave In*), e do escravo para o mestre no pino MISO (*Master In – Slave Out*). Após cada pacote de dados, o mestre sincroniza o escravo colocando o pino  $\overline{SS}$  em nível alto.

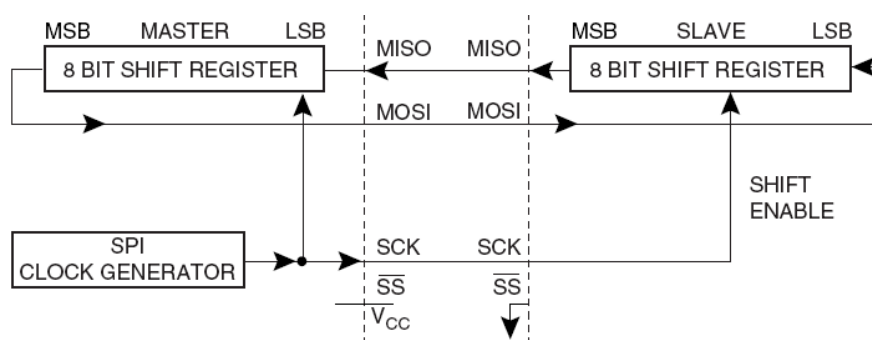


Fig. 12.1 – Conexão entre mestre e escravo na SPI.

Quando configurada com mestre, a interface SPI não tem controle automático sobre o pino  $\overline{SS}$ . Isto deve ser feito por software antes da comunicação começar. Quando isto é feito, escrever um byte no registrador de dados do SPI inicializa a geração do clock, e o hardware envia os oito bits para o escravo. Após este envio, o *clock* do SPI pára, setando o aviso de final de transmissão (*flag SPIF*). Se o bit de interrupção da SPI for habilitado (*SPIE*) no registrador *SPCR*, uma interrupção será gerada. O mestre pode continuar a enviar o próximo byte escrevendo-o no registrador *SPDR*,

ou sinalizar o final da transmissão ( $\overline{SS}$  em nível alto). O último byte de chegada é mantido no registrador de *buffer*.

O sistema possui um registrador de *buffer* na direção de transmissão e um duplo na direção de recepção. Isto significa que o byte a ser transmitido não pode ser escrito no registrador de dados SPI antes do final da transmissão do byte. Quando um dado é recebido, ele deve ser lido do registrador SPI antes do próximo dado ser completamente recebido, caso contrário, o primeiro dado é perdido.

No modo escravo, para garantir a correta amostragem do sinal de *clock*, o período mínimo e máximo devem ser maiores que 2 ciclos de *clock* da CPU.

Quando o modo SPI é habilitado, a direção dos pinos MOSI, MISO, SCK e  $\overline{SS}$  é ajustada conforme Tab. 12.1.

Tab. 12.1 – Direção dos pinos para a SPI quando habilitada.

Pin	Direction, Master SPI	Direction, Slave SPI
MOSI	User Defined	Input
Pin	Direction, Master SPI	Direction, Slave SPI
MISO	Input	User Defined
SCK	User Defined	Input
$\overline{SS}$	User Defined	Input

O código a seguir mostra como inicializar a SPI como mestre e como executar uma simples transmissão.

```
//===== //
//          Funções para inicializar a SPI no modo Mestre e transmitir um dado //
//===== //
void SPI_Mestre_Inic( )
{
    DDRB = (1<<PB5)|(1<<PB3);           //Ajusta MOSI e SCK como saída, demais pinos entrada
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0); //Habilita SPI, Mestre, taxa de clock ckl/16
}
//-----
void SPI_Mestre_Transmit(char dado)
{
    SPDR = dado;           //Inicia a transmissão
    while(!(SPSR & (1<<SPIF))); //Espera a transmissão ser completada
}
//=====
```

O código a seguir mostra como inicializar a SPI como escravo e como executar uma simples recepção.

```
//===== //
//          Funções para inicializar a SPI no modo Escravo e receber um dado //
//===== //
void SPI_Escravo_Inic( )
{
    DDRB = (1<<PB4); //Ajusta o pino MISO como saída, demais entrada
    SPCR = (1<<SPE); //Habilita SPI
}
//-----
```

```

char SPI_Escravo_Recebe( )
{
    while(!(SPSR & (1<<SPIF)));    //Espera a recepção estar completa
    return SPDR;                    //Retorna o registrador de dados
}
//=====

```

### Registrador de controle SPI

Bit	7	6	5	4	3	2	1	0
<b>SPCR</b>	<b>SPIE</b>	<b>SPE</b>	<b>DORD</b>	<b>MSTR</b>	<b>CPOL</b>	<b>CPHA</b>	<b>SPR1</b>	<b>SPR0</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

#### BIT 7 – SPIE: *SPI Interrupt Enable*

Quando este bit estiver em 1 a interrupção do SPI é executada se o bit SPIF do registrador SPSR também estiver em 1. O bit I do SREG deve estar habilitado.

#### BIT 6 – SPE: *SPI Enable*

Bit para habilitar a SPI.

#### BIT 5 – DORD: *Data Order*

Bit = 1 transmite primeiro o LSB, em zero transmite o MSB.

#### BIT 4 – MSTR: *Master/Slave Select*

Bit = 1 seleciona o modo mestre, contrário, o modo escravo. Se o pino  $\overline{SS}$  é configurado como entrada e estiver em nível zero enquanto MSTR estiver setado, MSTR será limpo, e o bit SPIF do SPSR será setado. Então será necessário setar MSTR para re-habilitar o modo mestre.

#### BIT 3 – CPOL: *Clock Polarity*

Quando este bit é setado o pino SCK é mantido em nível alto quando inativo e vice-versa.

#### BIT 2 – CPHA: *Clock Phase*

Este bit determina se o dado será coletado na subida ou descida do sinal de *clock*.

#### BIT 1,0 – SPR1, SPR0: *SPI Clock Rate Select 1 e 0*

Estes dois bits controlam a taxa do SCK quando o  $\mu$ controlador é configurado como mestre, sem efeito quando configurado como escravo, ver Tab. 12.2.

Tab. 12.2 – Seleção da frequência de operação para o modo mestre ( $f_{osc}$  = freq. De operação da CPU)

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

### Registrador de *status* SPI

Bit	7	6	5	4	3	2	1	0
<b>SPSR</b>	<b>SPIF</b>	<b>WCOL</b>	–	–	–	–	–	<b>SPI2X</b>
Read/Write	R	R	R	R	R	R	R	R/W
Initial Value	0	0	0	0	0	0	0	0

#### **BIT 7 – SPIF: SPI Interrupt Flag**

Colocado em 1 quando uma transferência serial for completada.

#### **BIT 6 – WCOL: Write COLision Flag**

É setado se o registrador SPDR é escrito durante uma transmissão.

#### **BIT 0 – SPI2X: Double SPI Speed Bit**

Bit = 1 duplica a velocidade de transmissão quando no modo mestre. No modo escravo, a SPI trabalha garantidamente com  $f_{osc}/4$ .

### Registrador de Dados

Bit	7	6	5	4	3	2	1	0
<b>SPDR</b>	<b>MSB</b>							<b>LSB</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	X	X	X	X	X	X	X	X

Undefined

Escrever neste registrador inicia uma transmissão; ao lê-lo, o *buffer* do registrador de entrada é lido.

### Modos de operação

Existem 4 combinações de fase e polaridade de *clock* com respeito aos dados seriais, determinados pelos bits de controle CPHA e CPOL, conforme Tab. 12.3 e Figs. 12.2-3.

Tab. 12.3 – Funcionalidade dos bits CPOL e CPHA.

	Leading Edge	Trailing Edge	SPI Mode
CPOL = 0, CPHA = 0	Sample (Rising)	Setup (Falling)	0
CPOL = 0, CPHA = 1	Setup (Rising)	Sample (Falling)	1
CPOL = 1, CPHA = 0	Sample (Falling)	Setup (Rising)	2
CPOL = 1, CPHA = 1	Setup (Falling)	Sample (Rising)	3

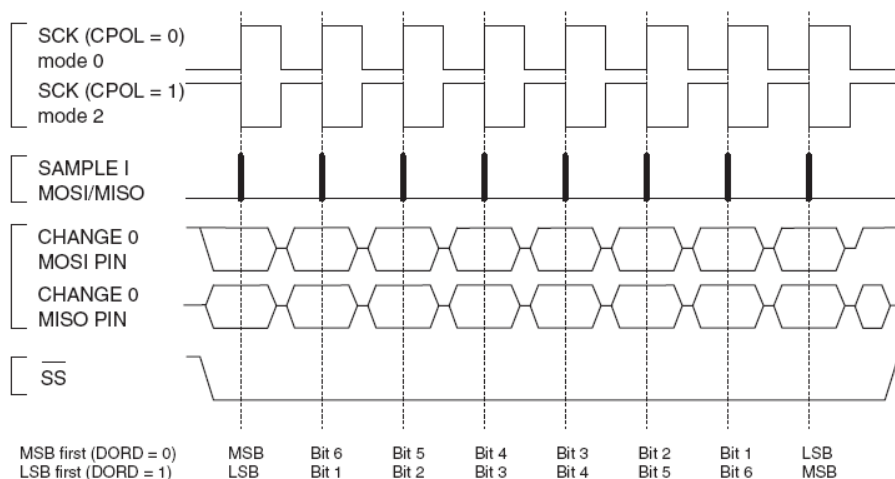


Fig. 12.2 – Formato da transferência SPI com CPHA=0.

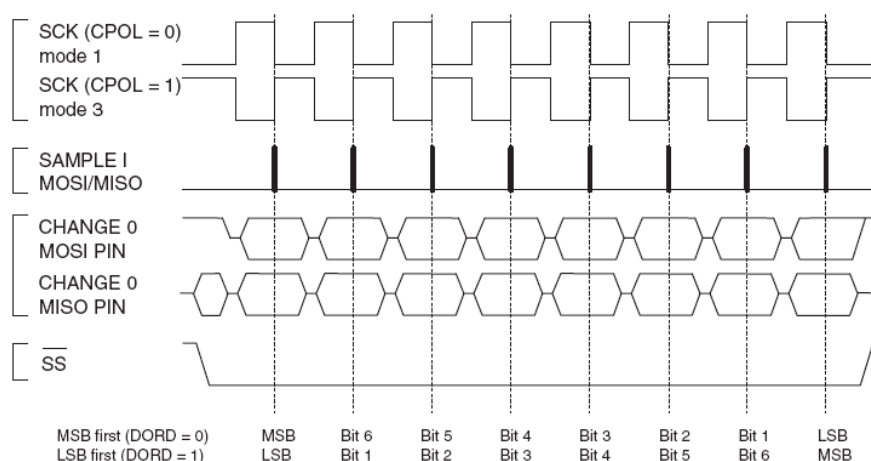
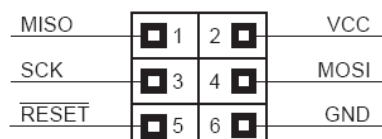


Fig. 12.3 – Formato da transferência SPI com CPHA=1.

## 12.1 GRAVAÇÃO IN-SYSTEM DO ATMEGA

A SPI também é empregada para programar a memória *flash* e ler ou escrever na EEPROM do AVR. No ATmega, a chamada programação ISP (*In-System Programming*) utiliza a interface SPI para permitir a gravação do  $\mu$ controlador sem retirá-lo do circuito de trabalho (de fundamental aplicação com componentes SMD). É de uso garantido no desenvolvimento do circuito de hardware. A Atmel recomenda a configuração de conector para a ISP dada abaixo. A Tab. 12.4 apresenta a funcionalidade dos referidos pinos.

Fig. 12.4 – Layout do conector para gravação *In-System*, vista aérea.

Tab. 12.4 – Descrição dos pinos para ISP.

Pin	Name	Comment
SCK	Serial Clock	Programming clock, generated by the In-System Programmer (Master)
MOSI	Master Out – Slave In	Communication line from In-System Programmer (Master) to target AVR being programmed (Slave)
MISO	Master In – Slave Out	Communication line from target AVR (Slave) to In-System Programmer (Master)
GND	Common Ground	The two systems must share the same common ground
RESET	Target AVR MCU Reset	To enable In-System Programming, the target AVR Reset must be kept active. To simplify this, the In-System Programmer should control the target AVR Reset
V <sub>CC</sub>	Target Power	To allow simple programming of targets operating at any voltage, the In-System Programmer can draw power from the target. Alternatively, the target can have power supplied through the In-System Programming connector for the duration of the programming cycle



Cuidados devem ser tomados quando os pinos do  $\mu$ controlador são empregados na programação *In-System* e também são utilizados pelo hardware para outras funções. Deve-se evitar que eles absorvam corrente do sistema de gravação. Um exemplo da conexão do ATmega8 para a ISP é apresentada na Fig. 12.5, o detalhe é que se perde o pino de reset como possível I/O. O circuito pode ser o mesmo para os diferentes AVRs.

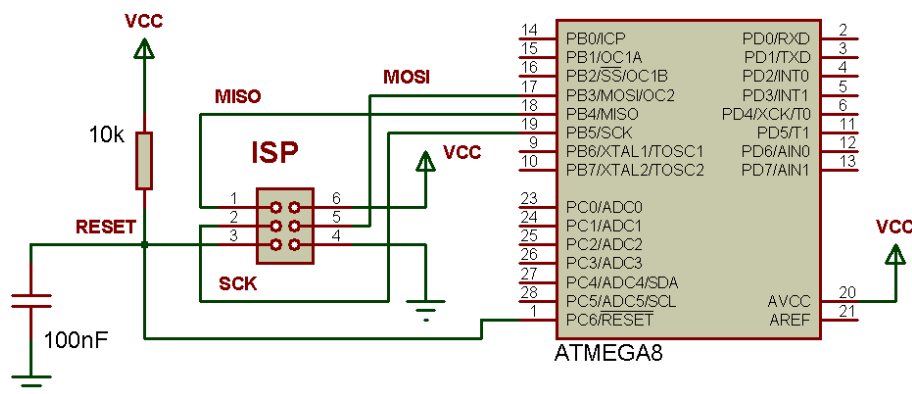


Fig. 12.5 – Conexão do ATmega8 para ISP.

## Exercícios:

**12.1** – Elaborar um programa para a leitura do sensor de temperatura MAX6675 do circuito da Fig. 12.6.

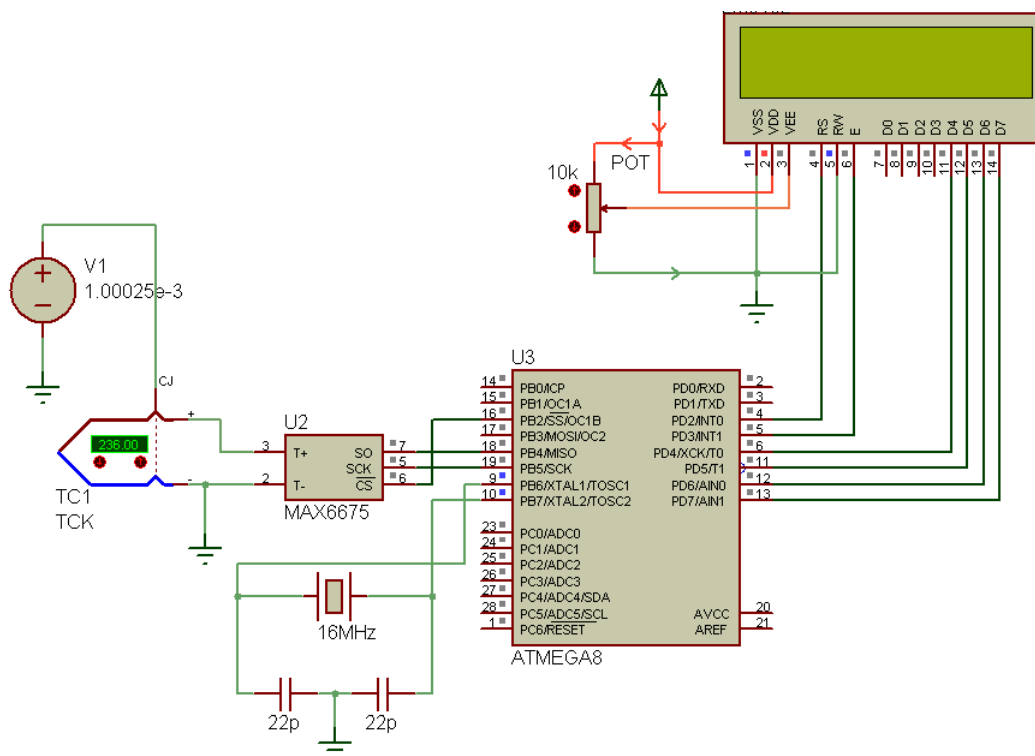


Fig. 12.6 – Usando o sensor MAX6675.

**12.2** – Elaborar um programa para a comunicação através da ISP de dois ATmega8. Faça o circuito de simulação e troque alguns dados entre os  $\mu$ controladores.

### 13. USART

A *Universal Synchronous and Asynchronous serial Receiver and Transmitter* é um dispositivo de comunicação serial muito flexível. Suas principais características são:

- Operação *Full Duplex* (independentes registradores de recepção e transmissão).
- Operação Síncrona ou Assíncrona.
- Operação Síncrona com *clock* mestre ou escravo.
- Gerador de taxa de transmissão de alta resolução (*Baud Rate Generator*).
- Suporta *frames* seriais com 5, 6, 7, 8 ou 9 bits de dados e 1 ou 2 bits de parada.
- Gerador de paridade par ou ímpar e conferência de paridade por hardware.
- Detecção de colisão de dados e erros de *frames*.
- Filtro para ruído incluindo falso bit de início e filtro digital passa-baixas.
- Três fontes separadas de interrupção (TX completo, RX completo e esvaziamento do registrador de dados TX).
- Modo de comunicação multi-processado.
- Modo de comunicação assíncrono com velocidade duplicável.

Para gerar a taxa de transmissão no modo mestre é empregado o registrador UBRR (*USART Baud Rate Register*). Um contador decrescente que roda na velocidade de *clock* da CPU é carregado com o valor de UBRR cada vez que chega a zero ou quando o UBRR é escrito. Assim, um pulso de *clock* é gerado cada vez que esse contador zera, determinando o *baud rate* ( $f = f_{osc}/(UBRR+1)$ ). O transmissor dividirá o *clock* de *baud rate* para 2, 8 ou 16 de acordo com o modo programado. A taxa de transmissão de saída é usada diretamente pela unidade de recepção e recuperação de dados. A Tab. 13.1 apresenta as equações para cálculo da taxa de transmissão (bits por segundo, bps) e para cálculo do valor de UBRR para cada modo de operação usando a fonte de *clock* interna.

Tab. 13.1 – Equações para calcular o ajuste do registrador de taxa de transmissão.

Operating Mode	Equation for Calculating Baud Rate	Equation for Calculating UBRR Value
Asynchronous Normal mode (U2X = 0)	$BAUD = \frac{f_{osc}}{16(UBRR + 1)}$	$UBRR = \frac{f_{osc}}{16BAUD} - 1$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \frac{f_{osc}}{8(UBRR + 1)}$	$UBRR = \frac{f_{osc}}{8BAUD} - 1$
Synchronous Master Mode	$BAUD = \frac{f_{osc}}{2(UBRR + 1)}$	$UBRR = \frac{f_{osc}}{2BAUD} - 1$

Para duplicar a taxa de transmissão basta setar o bit U2X do registrador UCSRA, tendo efeito somente para transmissão assíncrona. Para operação síncrona esse bit deve ser colocado em zero.

Quando se emprega *clock* externo, a frequência máxima deste está limitada pelo tempo de resposta da CPU, devendo seguir a seguinte equação:

$$f_{XCK} < \frac{f_{osc}}{4} \quad (13.1)$$

A frequência do sistema ( $f_{osc}$ ) depende da estabilidade da fonte de *clock*. É recomendado usar alguma margem de segurança para evitar possível perda de dados.

Quando o modo síncrono é usado (UMSEL=1), o pino XCK será empregado como *clock* de entrada (escravo) ou saída (mestre). O princípio básico é que o dado de entrada (no pino RxD) é amostrado na borda oposta do XCK quando a borda do dado de saída é alterada (pino TxD). O bit UCPOL do registrador UCRSC seleciona a borda usada para amostrar o dado e qual é usada para mudança de dado. A Fig. 13.1 apresenta o efeito do bit UCPOL.

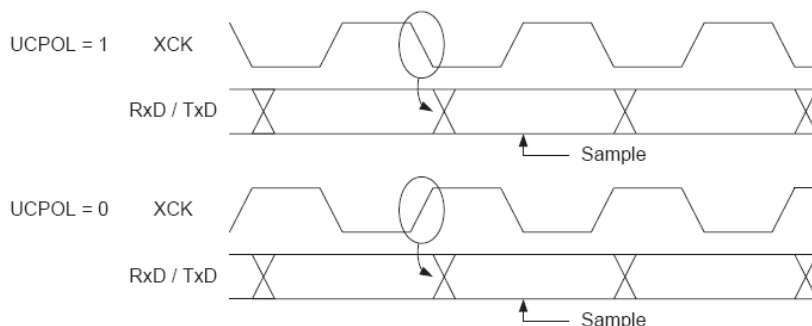
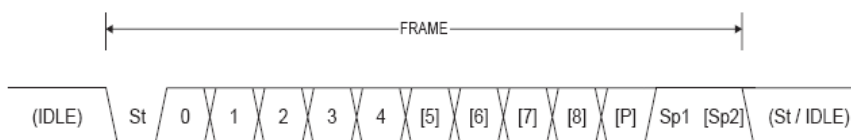


Fig. 13.1 – Efeito do bit UCPOL na amostragem dos dados.

A formatação dos bits na transmissão/recepção (*frame*) é definida por um caractere de dados com bits de sincronização (bits de início e parada) e opcionalmente um bit de paridade para conferência de erro. A USART aceita 30 combinações possíveis de formatos de dados:

- Um bit de início.
- 5, 6, 7, 8 ou 9 bits de dados.
- Bit de paridade par, ímpar ou nenhum.
- Um ou dois bits de parada.

Um *frame* inicia com um bit de início seguido pelo bit menos significativo (LSB). Seguem os outros bits de dados num total de até 9 bits, terminando com o bit mais significativo (MSB). Se habilitado, o bit de paridade é inserido após os bits de dados, antes dos bits de parada. Após a transmissão completa de um frame, pode-se seguir outra transmissão ou aguardar-se nova transmissão. A Fig. 13.2 ilustra o formato do *frame*.



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxD or TxD). An IDLE line must be high.

Fig. 13.2 – Formato do *frame* da USART.

O formato do *frame* é definido pelos bits UCSZ2:0, UPM1:0 e USBS nos registradores UCSRB e UCSRC. O transmissor e o receptor usam a mesma configuração.

Antes de qualquer comunicação, a USART deve ser inicializada. Os detalhes de configuração necessitam ser ajustados nos registradores específicos, detalhados posteriormente. A seguir são apresentados exemplos para trabalho com a USART, a partir de códigos extraídos do catálogo do fabricante.

```
//===== //
//          INICIALIZANDO A USART          //
//===== //
#define FOSC 1843200      //clock speed
#define BAUD 9600
#define MYUBRR FOSC/16/BAUD-1
//-----
void main( void )
{
    ...
    USART_Init (MYUBRR);
    ...
}
//-----
void USART_Init(unsigned int ubrr)
{
    UBRRH = (unsigned char)(ubrr>>8);      //Set baud rate
    UBRL = (unsigned char)ubrr;
    UCSRB = (1<<RXEN)|(1<<TXEN);           //Enable receiver and transmitter
    UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0); //Set frame format: 8data, 2stop bit
}
//=====

//===== //
//          ENVIANDO FRAMES COM 5 A 8 BITS   //
//===== //
void USART_Transmit( unsigned char data )
{
    while ( !( UCSRA & (1<<UDRE)) ); //Wait for empty transmit buffer

    UDR = data;                      //Put data into buffer, sends the data
}
//=====

//===== //
//          ENVIANDO FRAMES COM 9 BITS       //
//===== //
void USART_Transmit(unsigned int data )
{
    while(!(UCSRA &(1<<UDRE)));      //Wait for empty transmit buffer

    UCSRB &= ~(1<<TXB8);             //Copy ninth bit to TXB8

    if ( data & 0x0100 )
        UCSRB |= (1<<TXB8);

    UDR = data;                     //Put data into buffer, sends the data
}
//=====

//===== //
//          RECEBENDO FRAMES COM 5 A 8 BITS  //
//===== //
unsigned char USART_Receive( void )
{
    while (!(UCSRA & 1<<RXC));      //Wait for data to be received
    return UDR;                     //Get and return received data from buffer
}
//=====
```

```

//===== //
//          RECEBENDO FRAMES COM 9 BITS          //
//===== //
unsigned int USART_Receive( void )
{
    unsigned char status, resh, resl; //Wait for data to be received

    while(!(UCSRA & (1<<RXC)));        //Get status and ninth bit, then data from buffer

    status = UCSRA;
    resh = UCSRB;
    resl = UDR;

    if ( status & (1<<FE)|(1<<DOR)|(1<<PE) ) //If error, return -1
        return -1;

    resh = (resh >> 1) & 0x01;          //Filter the ninth bit, then return
    return ((resh << 8) | resl);
}
//=====

//===== //
//  LIMPANDO O REGISTRADOR DE ENTRADA (quando ocorre um erro p. ex.)  //
//===== //
void USART_Flush( void )
{
    unsigned char dummy;
    while(UCSRA & (1<<RXC)) dummy = UDR;
}
//=====

//===== //
//  ACESSANDO OS REGISTRADORES UBRRH/UCSRC - ESCRITA                  //
//===== //
...
UBRRH = 0x02; //Set UBRRH to 2
...
UCSRC = (1<<URSEL)|(1<<USBS)|(1<<UCSZ1); //Set the USBS and the UCSZ1 bit to one, and
//the remaining bits to zero.
...
//=====

//===== //
//  ACESSANDO OS REGISTRADORES UBRRH/UCSRC - LEITURA                //
//===== //
unsigned char USART_ReadUCSRC( void )
{
    unsigned char ucsrc;

    ucsrc = UBRRH; //Read UCSRC
    ucsrc = UCSRC;
    return ucsrc;
}
//=====

```

A faixa de operação do receptor é dependente do descasamento entre o bit recebido e a taxa de transmissão gerada internamente (*baud rate*). Se o transmissor estiver enviando bits muito rapidamente, ou muito devagar, ou ainda, se o *clock* gerado internamente no receptor não tiver a frequência base similar ao transmissor, o receptor não será capaz de sincronizar os frames relativos ao bit de início. Isto gerará um erro na taxa de recepção, que deve, segundo a Atmel, estar entre  $\pm 1$  a  $\pm 3\%$  (ver Anexo 6 para um detalhamento completo). O erro é calculado como:

$$Erro[\%] = \frac{Baud\ Rate_{closest\ Match}}{Baud\ Rate} \cdot 100\% \quad (13.2)$$

## REGISTRADORES DA USART

### USART I/O Data Register:

Bit	7	6	5	4	3	2	1	0	
<b>UDR</b>	RXB[7:0]								UDR (Read)
	TXB[7:0]								UDR (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Os registradores de recebimento e envio de dados dividem o mesmo endereço. Entretanto, uma leitura é feita no UDR de leitura e uma escrita no UDR de escrita. O hardware se encarrega da distinção. Para *frames* com 5, 6 ou 7 bits, estes serão ignorados e na recepção colocados em zero. O UDR só pode ser escrito quando o bit UDRE do registrador UCSRA estiver setado.

### USART Control and Status Register A:

Bit	7	6	5	4	3	2	1	0
<b>UCSRA</b>	RXC	TXC	UDRE	FE	DOR	PE	U2X	PMCM
Read/Write	R	R/W	R	R	R	R	R/W	R/W
Initial Value	0	0	1	0	0	0	0	0

#### **BIT 7 – RXC: USART Receive Complete**

Este bit é setado quando existe um dado não lido no buffer de recepção e é limpo quando lido.

#### **BIT 6 – TXC: USART Transmit Complete**

É setado quando um *frame* inteiro for enviado e não existe um novo para transmissão. É automaticamente limpo quando a interrupção de transmissão completa é executada, ou pode ser limpo gravando-se 1.

#### **BIT 5 – UDRE: USART Data Register Empty**

Em 1 indica se o registrador UDR está pronto para receber novo dado.

#### **BIT 4 – FE: Frame Error**

Indica se existe um erro no frame recebido. Este bit deve sempre ser zerado quando se escrever no registrador UCSRA.

#### **BIT 3 – DOR: Data OverRun**

Ocorre quando o registrador de entrada está cheio, não foi lido e um novo bit de início é detectado. Este bit deve sempre ser zerado quando se escrever no registrador UCSRA.

#### **BIT 2– PE: Parity Error**

Indica se existe um erro de paridade no dado recebido, fica setado até UDR ser lido. Este bit deve sempre ser zerado quando se escrever no registrador UCSRA.

#### **BIT 1 – U2X: Double the USART transmission speed**

Este bit só tem efeito no modo de operação assíncrona.

#### **BIT 0 – PMCM: Multi-processor Communication Mode**

Habilita a comunicação com vários processadores. Quando habilitado, todos os *frames* recebidos serão ignorados se não contiverem uma informação de endereço.

USART Control and Status Register B:

Bit	7	6	5	4	3	2	1	0
<b>UCSRB</b>	<b>RXCIE</b>	<b>TXCIE</b>	<b>UDRIE</b>	<b>RXEN</b>	<b>TXEN</b>	<b>UCSZ2</b>	<b>RXB8</b>	<b>TXB8</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Initial Value	0	0	0	0	0	0	0	0

**BIT 7 – RXCIE: RX Complete Interrupt Enable**

Este bit habilita a interrupção com o bit RXC. Uma interrupção de recepção será gerada somente se o bit RXCIE, o bit I (SREG) e o bit RXC estiverem setados.

**BIT 6 – TXCIE: TX Complete Interrupt Enable**

Este bit habilita a interrupção com o bit TXC. Uma interrupção de transmissão será gerada somente se o bit TXCIE, o bit I (SREG) e o bit TXC estiverem setados.

**BIT 5 – UDRIE: USART Data Register Empty Interrupt Enable**

Este bit habilita a interrupção com o bit UDRE. Uma interrupção de dado de registrador vazio será gerada somente se o bit UDRIE, o bit I (SREG) e o bit UDRE estiverem setados.

**BIT 4 – RXEN: Receiver Enable**

Este bit habilita a recepção da USART. O receptor irá alternar a operação do pino RxD. Desabilitando o receptor ocorrerá o esvaziamento do registrador de entrada, invalidando os bits FE, DOR e PE.

**BIT 3 – TXEN: Transmitter Enable**

Este bit habilita a transmissão da USART. O transmissor irá alternar a operação do pino TxD. A desabilitação do transmissor só terá efeito após as transmissões pendentes serem completadas.

**BIT 2 – UCSZ2: Character Size**

Este bit combinado com os bits UCSZ1:0 do registrador UCSRC ajustam o número de bits de dado do *frame*.

**BIT 1 – RXB8: Receive Data Bit 8**

É o nono bit de dados recebidos quando o frame for de 9 bits. Deve ser lido antes dos outros bits do UDR.

**BIT 0 – TXB8: Transmit Data Bit 8**

É o nono bit de dados a ser transmitido quando o frame for de 9 bits. Deve ser escrito antes dos outros bits no UDR.

USART Control and Status Register C:

Bit	7	6	5	4	3	2	1	0
<b>UCSRC</b>	<b>URSEL</b>	<b>UMSEL</b>	<b>UPM1</b>	<b>UPM0</b>	<b>USBS</b>	<b>UCSZ1</b>	<b>UCSZ0</b>	<b>UCPOL</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	1	0	0	0	0	1	1	0

O registrador UCSRC divide o mesmo endereço do registrador UBRRH.

**BIT 7 – URSEL: Register Select**

Este bit seleciona o acesso entre o registrador UCSRC ou UBRRH. Ele é lido como 1 quando se está lendo UCSRC. O URSEL deve ser 1 quando se estiver escrevendo em UCSRC.

**BIT 6 – UMSEL: USART Mode Select**

Este bit seleciona entre o modo assíncrono (UMSEL=0) ou síncrono (UMSEL=1).

**BIT 5,4 – UPM1:0: Parity Mode**

Estes bits habilitam e ajustam o gerador de paridade e de conferência. Se habilitado, o transmissor irá gerar e enviar automaticamente o bit de paridade em cada frame. O receptor irá gerar o valor de paridade para comparação. Se uma desigualdade for detectada, o bit PE é setado. A Tab. 13.2 apresenta as possíveis configurações para os bits IPM1:0.

Tab. 13.2 – Bits UPM1:0.

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

**BIT 3 – USBS: Stop Bit Select**

Este bit seleciona o número de bits de parada a serem inseridos pelo transmissor (USBS=0, 1 bit de parada; USBS=1, 2 bits de parada).

**BIT 2,1 – UCSZ1:0: Character Size**

Estes bits combinados com o bit UCSZ2 no registrador UCSRB ajustam o número de bits de dados no frame do transmissor e receptor, conforme Tab. 13.3.

Tab. 13.3 – Ajuste dos bits UCSZ1:0.

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

**BIT 0– UCPOL: Clock Polarity**

Este bit é usado somente para o modo síncrono. Deve ser zero quando o modo assíncrono é usado. Ele ajusta o sinal de *clock* (XCK) para amostragem e saída de dados conforme Tab. 13.4.

Tab. 13.4 – Ajustando a polaridade do *clock*.

UCPOL	Transmitted Data Changed (Output of TxD Pin)	Received Data Sampled (Input on RxD Pin)
0	Rising XCK Edge	Falling XCK Edge
1	Falling XCK Edge	Rising XCK Edge

**USART Baud Rate Register – UBRRL e UBRRH:**

Bit	15	14	13	12	11	10	9	8
<b>UBRRH</b>	URSEL	–	–	–	UBRR[11:8]			
<b>UBRRL</b>	UBRR[7:0]							
	7	6	5	4	3	2	1	0
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0



O registrador UBRRH divide o mesmo endereço que o registrador UCSRC.

### **BIT 15 – URSEL: Register Select**

Este bit seleciona o acesso ao registrador UBRRH ou UCSRC. É lido como zero quando se lê o registrador UBRRH. Deve ser zero quando se escreve em UBRRH.

### **BIT 11:0– UBRR11:0: USART Baud Rate Register**

Este é o registrador de 12 bits que contém o valor da taxa de comunicação. Qualquer transmissão em andamento será corrompida se houver mudança desse valor. Qualquer escrita aqui atualiza imediatamente a taxa de comunicação.

## **Exercício:**

- 13.1** – Elaborar um programa, empregando o terminal virtual do Proteus, que simule uma comunicação serial (RS232) entre o AVR e um computador, conforme Fig. 13.3. O programa deve escrever “TESTE SERIAL” na primeira linha do LCD e mandar uma mensagem ao terminal virtual. Os caracteres digitados neste terminal devem ser escritos na segunda linha do LCD. Quando a mesma estiver completa, deve ser apagada e a escrita recomeçada. Utilizar 2400bps, 8 bits de dados, sem paridade e 1 bit de parada.

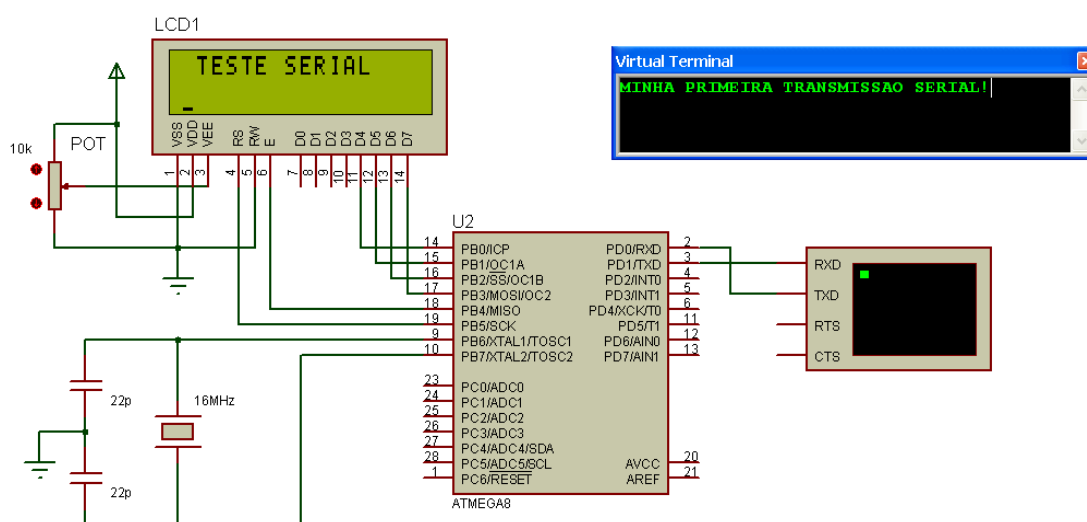


Fig. 13.3 – Comunicação serial: simulando no Proteus.

## 14. TWI (TWO WIRE SERIAL INTERFACE) – I2C

O I2C foi desenvolvido pela Philips e é baseado no protocolo de comunicação entre mestre/escravo empregando duas vias de comunicação. No ATmega foi denominado TWI e suas características são:

- Simples, mas poderosa e flexível interface de comunicação com barramento a dois fios (*clock* e dado).
- Operação como mestre ou escravo. Tanto mestre (controle) quando escravo podem transmitir ou receber dados.
- Endereçamento de 7 bits (permitindo até 128 diferentes dispositivos no mesmo barramento).
- Máxima transferência de dados de 400 kHz.
- *Drivers* de saída com limitação da taxa de subida (*Slew Rate*).
- Circuito supressor de ruído no barramento.
- Reconhecimento de endereço pode acordar o AVR do modo *Sleep*.

A diferença entre o I2C e o TWI é que o I2C permite endereçamento também de 10 bits (virtualmente até 1024 diferentes dispositivos no mesmo barramento!) e o TWI só permite o endereçamento de 7 bits.

Para evitar conflitos na via de dados (SDA), os pinos são chaveados como I/O dependendo das necessidades do protocolo (dreno aberto). O pino de *clock* (SCL) também é alterado como I/O. Em ambos pinos são necessários resistores de *pull-up* (10 k $\Omega$  e 1 k $\Omega$  – p/ 400 kHz). Assim, o valor padrão para SDA e SCL é o nível alto (1). Os bits são sempre lidos na borda de descida do SCL. Os dispositivos ligados ao barramento possuem endereços individuais. Os mecanismos inerentes ao trabalho com o I2C gerenciam o protocolo. A Fig. 14.1 ilustra o barramento I2C com vários dispositivos.

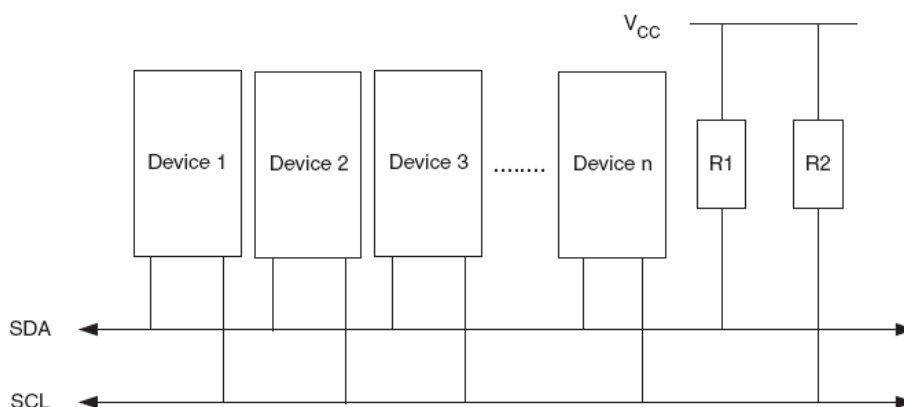


Fig. 14.1 – Barramento I2C – TWI.

O dispositivo mestre é que inicia e termina uma transmissão, gerando o sinal de *clock* SCL. O dispositivo escravo é endereçado pelo mestre. O transmissor é o dispositivo que coloca um dado no barramento e o dispositivo receptor é o que o lê.

Cada bit transferido para o barramento é acompanhado por um pulso da linha de *clock*. O nível lógico do dado deve estar estável quando a linha de *clock* for para o nível alto. A única exceção é quando se geram a condição de início e parada da comunicação. A Fig. 14.2 ilustra esse fato.

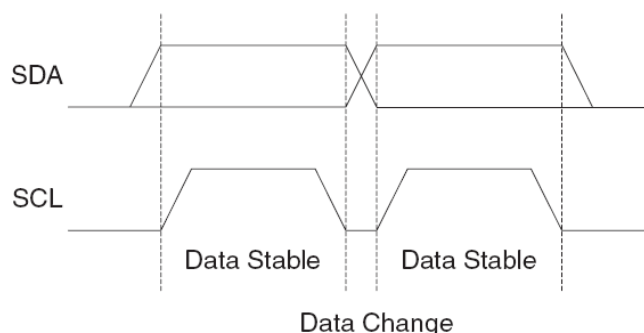


Fig. 14.2 – Validade dos dados no barramento I2C.

Os sinais possíveis no barramento são:

- **Condição de Início:** usada para informar à rede que uma transmissão irá ocorrer.
- **Condição de Parada:** usada para informar à rede que a transmissão acabou (necessária após o início para não travar o sistema).
- **Condição de Re-Início:** pode ser executada antes da parada, usada na comunicação com endereçamento de 10 bits, ou na necessidade de reenvio do primeiro byte antes de finalização com a parada.
- **Condição de Acknowledge (ACK):** resposta pelo mestre ou escravo, no lugar do 9º bit, informando se o dado foi corretamente recebido ou não.
- **Transmissão de endereço:** independentemente da informação a ser transmitida (endereço de 7 bits ou 10 bits e dados de 8 bits), a comunicação sempre é feita em pacotes de 8 bits e mais um retorno de *acknowledge*.
- **Transmissão de dados:** depois do término da transferência da parte relacionada ao endereço, mestre ou escravo deverá transmitir um ou mais bytes de dados.
- **Pausa:** Mantendo a linha de dados em nível baixo (ex. para processar a informação).

O mestre inicia e termina a transmissão de dados. A transmissão é iniciada quando o mestre executa uma condição de início no barramento, e termina quando o mestre executa uma condição de parada. Entre o início e a parada, o barramento é considerado ocupado e nenhum outro mestre deve tentar controlar o barramento. Um caso especial ocorre quando um novo início ocorre entre uma condição de início e parada. Isto é conhecido como re-início e é realizado quando o mestre deseja iniciar uma nova transmissão sem perder o controle sobre o barramento. Após o re-início, o barramento é considerado ocupado até a próxima parada. A Fig. 14.3 ilustra esse fato.

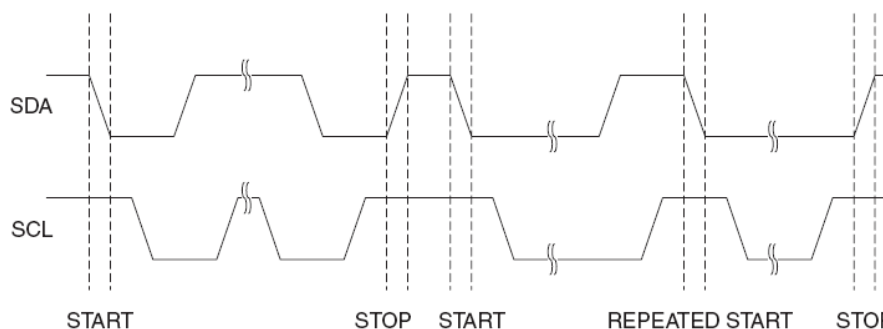


Fig. 14.3 – Condição de início, parada e re-início.

Todos os endereços transmitidos no barramento TWI são compostos por 9 bits, onde 7 são os bits referentes ao endereço, 1 bit de controle para leitura/escrita e 1 bit de confirmação (*acknowledge*). Se o bit de leitura/escrita é 1, uma leitura será realizada; caso contrário, uma escrita. Quando o escravo reconhece que está sendo endereçado, ele sinaliza colocando a linha SDA em baixo no nono ciclo do SCL (ACK). Se o endereço do escravo estiver ocupado, ou por outra razão não pode atender ao mestre, a linha SDA deve ser mantida alta no ciclo de *clock* do ACK. O mestre pode, então, transmitir uma condição de parada, ou re-início para iniciar uma nova transmissão. O pacote de endereço consistindo do endereço do escravo e o bit de leitura ou escrita é chamado SLA+R ou SLA+W, respectivamente.

O bit mais significativo MSB do endereço é transmitido primeiro. O endereço do escravo pode ser alocado pelo projetista, mas o endereço 0x00 é reservado para uma chamada geral. A Fig. 14.4 ilustra o formato do pacote de dados referente ao endereço.

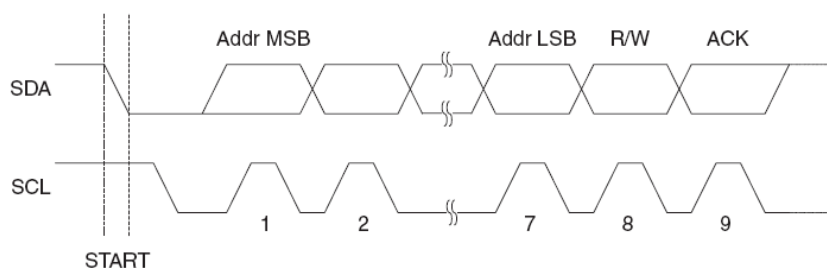


Fig. 14.4 – Formato do pacote de dados referentes ao endereço.

Quando uma chamada geral é realizada, todos os escravos devem responder colocando a linha SDA em nível baixo no ciclo de ACK. Essa chamada é realizada quando o mestre deseja transmitir a mesma mensagem aos escravos do sistema. Quando uma chamada geral é seguida por um bit de escrita, todos os escravos devem colocar a linha SDA em nível baixo no ciclo de ACK. Os pacotes seguintes de dados serão recebidos por todos os escravos que responderam. Transmitir uma chamada geral seguida de um bit de leitura não deve ser realizado, pois todos os escravos escreverão no barramento ao mesmo tempo.

Todos os pacotes transmitidos no barramento TWI possuem nove bits, um byte de dados e um bit de resposta (*acknowledge*). Durante a transmissão de dados, o mestre gera o clock e a condição de início e de parada, enquanto o escravo é responsável por responder se recebeu o dado. Um sinal ACK é realizado pelo receptor colocando a linha SDA em zero durante o nono ciclo do SCL. Se o receptor deixa a linha em nível alto, sinaliza um não-ACK, NACK. Quando o receptor receber o último byte ou por alguma razão não pode receber mais bytes, ele deve informar o transmissor enviando um NACK após o último byte. O bit mais significativo (MSB) do byte é transmitido primeiro. O formato do pacote de dados é apresentado na Fig. 14.5.

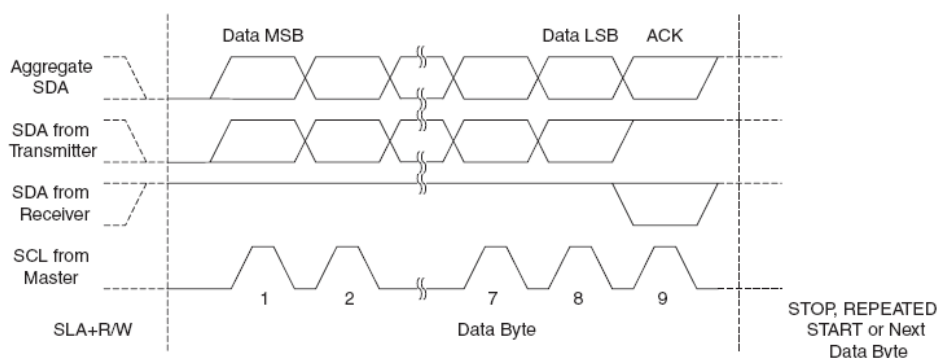


Fig. 14.5 – Pacote de dados.

Uma transmissão consiste de uma condição de início, um SLA+R/W, ou um ou mais pacotes de dados e uma condição de parada. Uma mensagem vazia, consistindo de um início seguido por uma condição de parada é ilegal. O escravo pode estender o período de tempo que SCL fica em nível baixo. Isto é útil se a velocidade do mestre é muito rápida para o escravo, ou o escravo necessita de maior tempo para o processando dos dados. O aumento do período de SCL em nível baixo não afeta o período SCL alto, o qual é determinado pelo mestre. Como consequência, o escravo pode reduzir a velocidade de transferência do TWI prolongando o ciclo ativo do SCL.

A Fig. 14.6 apresenta uma transmissão típica. Notar que vários bytes de dados podem ser transmitidos entre o SLA+R/W e a condição de parada, dependendo do protocolo implementado pelo software.

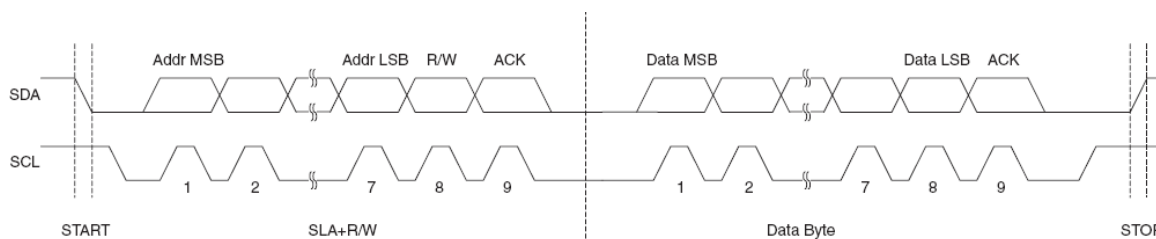


Fig. 14.6 – Transmissão de dados típica.

O barramento permite o uso de vários mestres, entretanto, este assunto não será abordado. A Fig. 14.7 exemplifica uma comunicação completa entre mestre e escravo, com endereçamento de 7 bits: o mestre transmite e o escravo recebe e vice-versa.

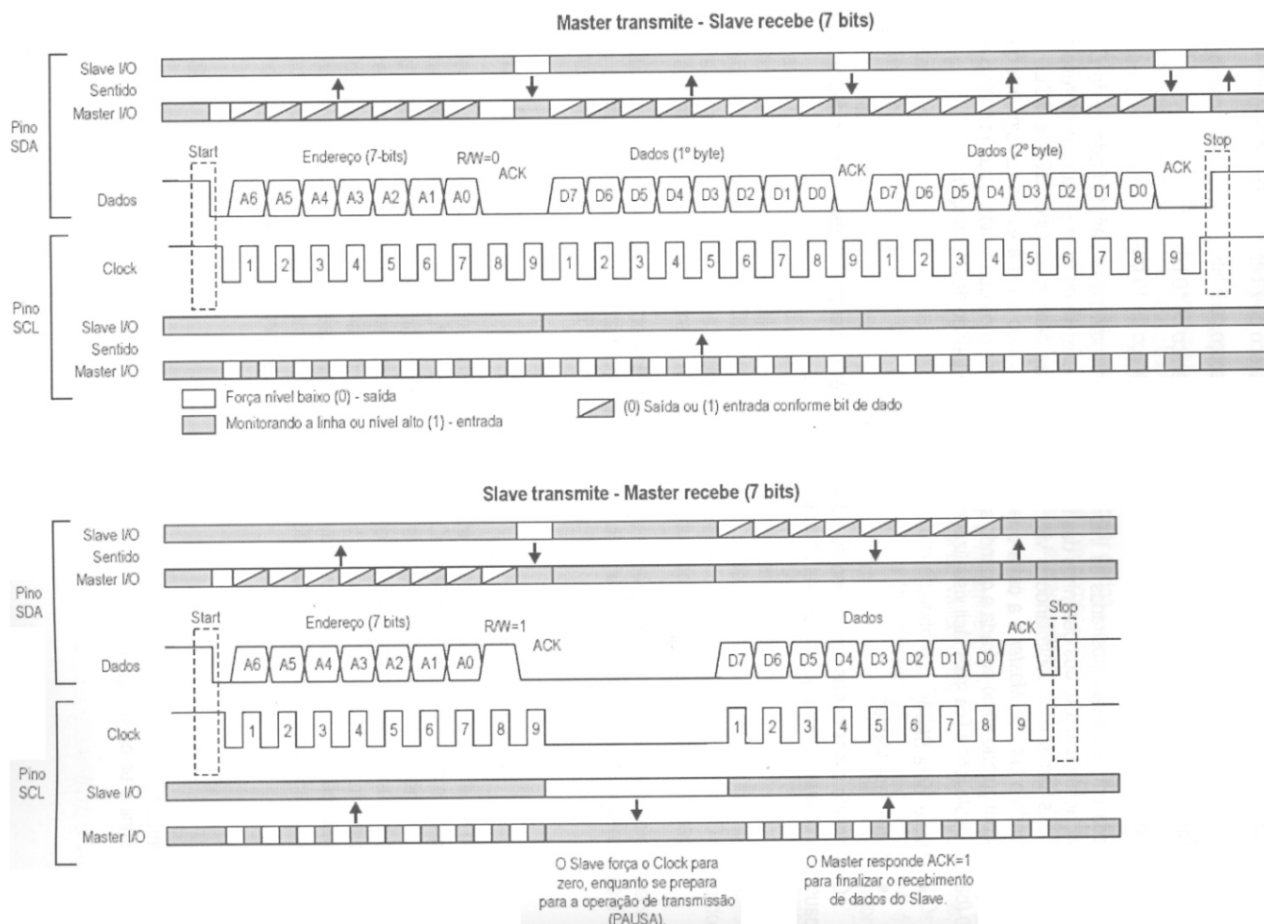


Fig. 14.7 - Exemplo da comunicação com endereçamento de 7 bits entre mestre e escravo.

## 14.1 REGISTRADORES DO TWI

O período SCL é controlado ajustando o registrador da taxa de bit (TWBR) e os bits de *prescaler* no registrador TWSR. O modo escravo não depende desse ajuste, entretanto, a frequência da CPU deve ser no mínimo 16 vezes maior que a frequência SCL. A frequência SCL é gerada de acordo com a seguinte equação:

$$SCL_{Freq} = \frac{CPU_{freq}}{16 + 2 \cdot (TWBR) \cdot 4 \cdot TWPS} \quad [Hz] \quad (14.1)$$

onde: TWBR é o valor do registrador de ajuste da taxa de bits e TWPS é o valor do *prescaler* do registrador de *status* do TWI.

### TWBR – TWI Bit Rate Register

Bit	7	6	5	4	3	2	1	0
<b>TWBR</b>	<b>TWBR7</b>	<b>TWBR6</b>	<b>TWBR5</b>	<b>TWBR4</b>	<b>TWBR3</b>	<b>TWBR2</b>	<b>TWBR1</b>	<b>TWBR0</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Este registrador seleciona o fator de divisão do gerador da taxa de bits, conforme Eq. 14.1.

## TWCR – TWI Control Register

Bit	7	6	5	4	3	2	1	0
<b>TWCR</b>	<b>TWINT</b>	<b>TWEA</b>	<b>TWSTA</b>	<b>TWSTO</b>	<b>TWWC</b>	<b>TWEN</b>	<b>–</b>	<b>TWIE</b>
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W
Initial Value	0	0	0	0	0	0	0	0

É empregado para controlar a operação do TWI: habilitar, iniciar uma condição de início, gerar o ACK de recebimento, gerar uma condição de parada. Mantém o barramento em alto enquanto o dado para o barramento é escrito no TWDR. Também indica uma colisão de dados se houver tentativa de escrita no TWDR enquanto este estiver inacessível.

### **BIT 7 – TWINT: TWI Interrupt Flag**

Este bit é setado por hardware quando o TWI concluir sua atividade corrente e espera uma resposta do software de aplicação. Se o bit I do SREG e o bit TWIE do TWCR estiverem setados, a CPU é desviada para o endereço do vetor de interrupção correspondente. Enquanto TWINT=1, o período baixo do SCL é mantido. Ele deve ser limpo por software pela escrita de 1. Limpar o bit TWINT inicia a operação do TWI, logo o acesso aos registradores TWAR, TWSR e TWDR já devem estar completos.

### **BIT 6 – TWEA: TWI Enable Acknowledge Bit**

Este bit controla a geração do pulso de ACK. Se for escrito em 1, um pulso de ACK é gerado no barramento TWI se as seguintes condições forem encontradas:

1. O endereço próprio do dispositivo como escravo tenha sido recebido.
2. Uma chamada geral tenha sido recebida, enquanto o bit TWGCE do TWAR estiver em 1.
3. Um byte de dados tenha sido recebido no modo mestre ou escravo.

Zerando o bit, o  $\mu$ controlador pode ser virtualmente desconectado temporariamente do barramento TWI. O reconhecimento de endereço pode ser habilitado escrevendo 1 no bit novamente.

### **BIT 5 – TWSTA: TWI Start Condition Bit**

Este bit setado indica que o modo de comunicação será mestre. Deve ser limpo por software após uma condição de início ser transmitida.

### **BIT 4 – TWSTO: TWI Stop Condition Bit**

Este bit em 1 gera uma condição de parada, sendo limpo automaticamente. No modo escravo pode ser utilizado para a recuperação de uma condição de erro, não gerando uma condição de parada, mas colocando os pinos do barramento em alta impedância e uma condição de não endereçamento.

### **BIT 3 – TWWC: TWI Write Collision Flag**

Este bit é setado quando se tenta escrever no registrador de dados TWDR enquanto TWINT estiver em baixo. É limpo escrevendo em TWDR quando TWINT estiver em alto.

### **BIT 2 – TWEN: TWI Enable Bit**

Este bit habilita a operação do TWI e ativa a interface TWI.

### **BIT 0 – TWIE: TWI Interrupt Enable**

Quando em 1 e o bit I do SREG estiver setado, a interrupção do TWI estará ativa pelo tempo que o bit TWINT estiver em alto.

TWCR – TWI Status Register

Bit	7	6	5	4	3	2	1	0
<b>TWSR</b>	<b>TWS7</b>	<b>TWS6</b>	<b>TWS5</b>	<b>TWS4</b>	<b>TWS3</b>	–	<b>TWPS1</b>	<b>TWPS0</b>
Read/Write	R	R	R	R	R	R	R/W	R/W
Initial Value	1	1	1	1	1	0	0	0

**BIT 7:3– TWS: TWI Status**

Estes 5 bits refletem o estado lógico do TWI. Como o registrador também contém 2 bits de *prescaler*, o software deve mascarar estes bits quando lendo os bits de *status*.

**BIT 1,0 – TWPS: TWI Prescaler Bits**

Estes bits podem ser lidos e escritos e controlam o *prescaler* da taxa de bits (Eq. 14.1), ver Tab. 14.1.

Tab. 14.1 – *Prescaler* da taxa de bits do TWI.

TWPS1	TWPS0	Prescaler Value
0	0	1
0	1	4
1	0	16
1	1	64

TWDR – TWI Data Register

Bit	7	6	5	4	3	2	1	0
<b>TWDR</b>	<b>TWD7</b>	<b>TWD6</b>	<b>TWD5</b>	<b>TWD4</b>	<b>TWD3</b>	<b>TWD2</b>	<b>TWD1</b>	<b>TWD0</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	1	1	1	1	1	1	1	1

No modo de transmissão, o TWDR contém o próximo byte a ser transmitido. No modo de recepção o TWDR contém o último byte recebido. Ele pode ser escrito se o TWI não estiver enviando ou recebendo um byte. O bit ACK é controlado automaticamente pelo TWI e a CPU não pode acessá-lo diretamente.

TWAR – TWI (Slave) Address Register

Bit	7	6	5	4	3	2	1	0
<b>TWAR</b>	<b>TWA6</b>	<b>TWA5</b>	<b>TWA4</b>	<b>TWA3</b>	<b>TWA2</b>	<b>TWA1</b>	<b>TWA0</b>	<b>TWGCE</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	1	1	1	1	1	1	1	0

Este registrador deve ser carregado com o endereço de 7 bits (7 MSB), para o qual o TWI irá responder quando programado para o modo escravo.

O bit 0 (TWGCE) é usado para habilitar o reconhecimento de chamada geral (0x00). Existe um comparador de endereço que gera um pedido de interrupção se houver igualdade no endereço do escravo ou se a chamada geral estiver habilitada.



## 14.2 USANDO O TWI

O TWI é orientado a byte e baseado em interrupções. Interrupções são feitas após os eventos do barramento, como a recepção de um byte ou a transmissão de uma condição de início. Sendo o TWI baseado em interrupções, o software fica livre para executar outras operações durante a transferência de bytes. Um exemplo simples da aplicação do TWI é apresentado na Fig. 14.8, no qual o mestre deseja transmitir um simples byte de dados para o escravo.

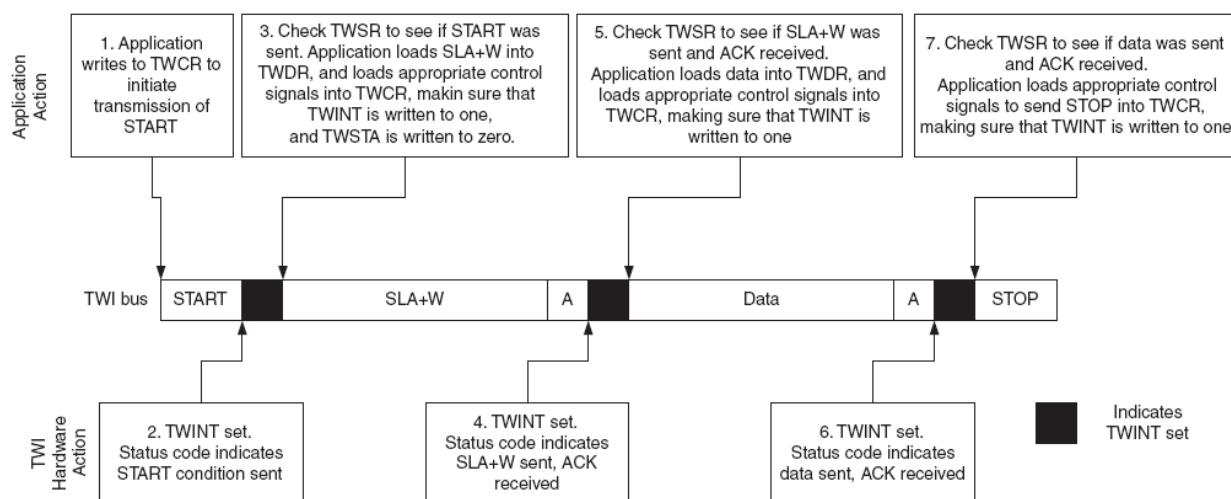


Fig. 14.8 – Transmissão típica com o TWI.

O código a seguir ilustra a programação para o exemplo acima.

```
//===== //
//          CÓDIGO EXEMPLO PARA USO DO TWI          //
//===== //
TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN); //Send START Condition
//-----//
while (!(TWCR & (1<<TWINT))); //Wait for TWINT Flag set. This indicates that
//the START condition has been transmitted
//-----//
if ((TWSR & 0xF8) != START) //Check value of TWI Status Register.
ERROR(); //Mask prescaler bits. If status different
//from START go to ERROR
//-----//
TWDR = SLA_W;
TWCR = (1<<TWINT) | (1<<TWEN); //Load SLA_W into TWDR Register.
//Clear TWINT bit in TWCR to start transmission of address
//-----//
while (!(TWCR & (1<<TWINT))); //Wait for TWINT Flag set. This indicates that the SLA+W has
//been transmitted, and ACK/NACK has been received.
//-----//
if ((TWSR & 0xF8) != MT_SLA_ACK) // Check value of TWI Status Register. Mask prescaler bits.
ERROR(); //If status different from MT_SLA_ACK go to ERROR
//-----//
TWDR = DATA;
TWCR = (1<<TWINT) | (1<<TWEN); //Load DATA into TWDR Register. Clear TWINT bit in TWCR to
//start transmission of data
//-----//
while (!(TWCR & (1<<TWINT))); //Wait for TWINT Flag set. This indicates that the DATA has
//been transmitted, and ACK/NACK has been received.
//-----//
if ((TWSR & 0xF8) != MT_DATA_ACK) //Check value of TWI Status Register. Mask prescaler bits.
ERROR(); //If status different from MT_DATA_ACK go to ERROR
//-----//
TWCR = (1<<TWINT)|(1<<TWEN)| (1<<TWSTO); //Transmit STOP condition
//===== //
```

O TWI pode operar em 4 modos principais: Mestre Transmite (MT), Mestre Recebe (MR), Escravo Transmite (ST) e Escravo Recebe (SR). Vários desses modos podem ser empregados em uma mesma aplicação. Por exemplo, o modo MT pode ser usado para escrever dados em um EEPROM I2C, e o modo MR para ler dados da mesma.

Os 4 modos serão descritos sucintamente de acordo com as tabelas e figuras abaixo, considerando as seguintes abreviações:

S: START condition

Rs: REPEATED START condition

R: Read bit (high level at SDA)

W: Write bit (low level at SDA)

A: Acknowledge bit (low level at SDA)

$\bar{A}$ : Not acknowledge bit (high level at SDA)

Data: 8-bit data byte

P: STOP condition

SLA: Slave Address

Tab. 14.2 – Códigos de *status* para o modo de transmissão mestre.

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+W	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+W or Load SLA+R	0 0	0 0	1 1	X X	SLA+W will be transmitted; ACK or NOT ACK will be received SLA+R will be transmitted; Logic will switch to Master Receiver mode
0x18	SLA+W has been transmitted; ACK has been received	Load data byte or No TWDR action or No TWDR action or No TWDR action	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	Data byte will be transmitted and ACK or NOT ACK will be received Repeated START will be transmitted STOP condition will be transmitted and TWSTO Flag will be reset STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x20	SLA+W has been transmitted; NOT ACK has been received	Load data byte or No TWDR action or No TWDR action or No TWDR action	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	Data byte will be transmitted and ACK or NOT ACK will be received Repeated START will be transmitted STOP condition will be transmitted and TWSTO Flag will be reset STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x28	Data byte has been transmitted; ACK has been received	Load data byte or No TWDR action or No TWDR action or No TWDR action	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	Data byte will be transmitted and ACK or NOT ACK will be received Repeated START will be transmitted STOP condition will be transmitted and TWSTO Flag will be reset STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x30	Data byte has been transmitted; NOT ACK has been received	Load data byte or No TWDR action or No TWDR action or No TWDR action	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	Data byte will be transmitted and ACK or NOT ACK will be received Repeated START will be transmitted STOP condition will be transmitted and TWSTO Flag will be reset STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x38	Arbitration lost in SLA+W or data bytes	No TWDR action or No TWDR action	0 1	0 0	1 1	X X	Two-wire Serial Bus will be released and not addressed Slave mode entered A START condition will be transmitted when the bus becomes free

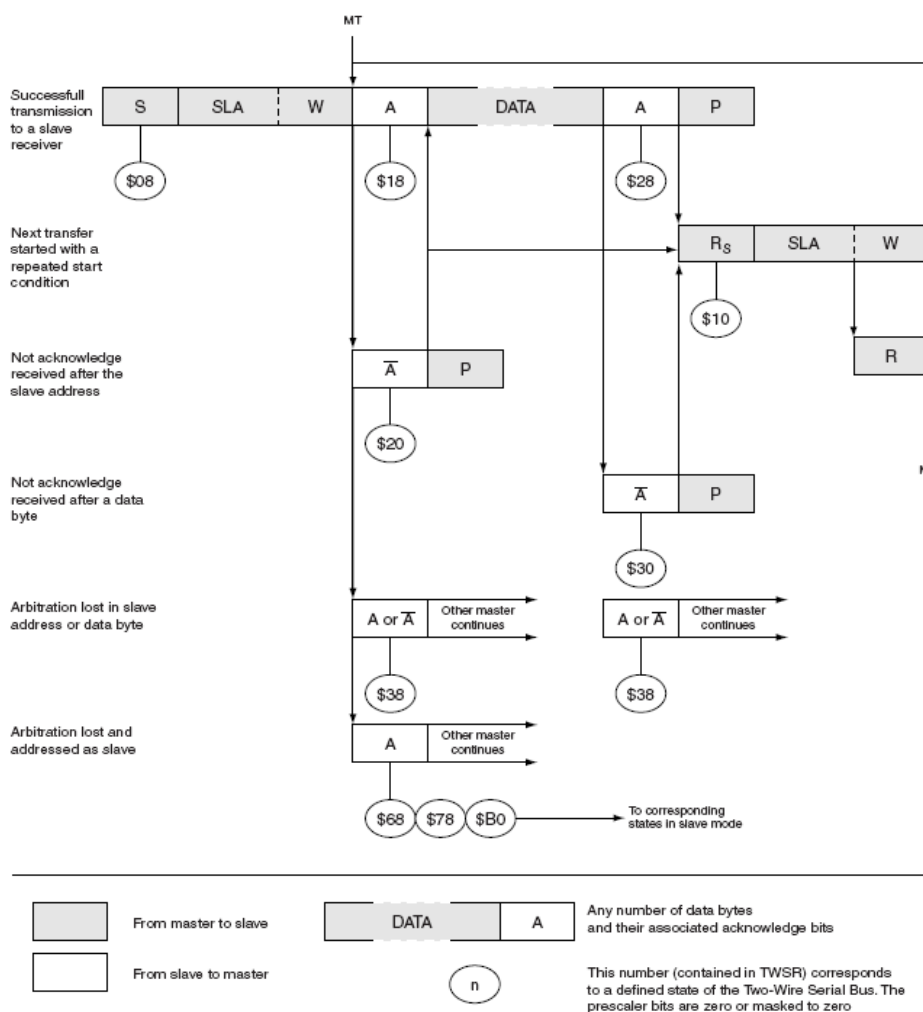


Fig. 14.9 – Formatos e estados do modo de transmissão mestre.

Tab. 14.3 – Códigos de *status* para o modo de recepção mestre.

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+R	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+R or	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received SLA+W will be transmitted Logic will switch to Master Transmitter mode
		Load SLA+W	0	0	1	X	
0x38	Arbitration lost in SLA+R or NOT ACK bit	No TWDR action or	0	0	1	X	Two-wire Serial Bus will be released and not addressed Slave mode will be entered A START condition will be transmitted when the bus becomes free
		No TWDR action	1	0	1	X	
0x40	SLA+R has been transmitted; ACK has been received	No TWDR action or	0	0	1	0	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
		No TWDR action	0	0	1	1	
0x48	SLA+R has been transmitted; NOT ACK has been received	No TWDR action or	1	0	1	X	Repeated START will be transmitted STOP condition will be transmitted and TWSTO Flag will be reset STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
		No TWDR action or	0	1	1	X	
		No TWDR action	1	1	1	X	
0x50	Data byte has been received; ACK has been returned	Read data byte or	0	0	1	0	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
		Read data byte	0	0	1	1	
0x58	Data byte has been received; NOT ACK has been returned	Read data byte or	1	0	1	X	Repeated START will be transmitted STOP condition will be transmitted and TWSTO Flag will be reset STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
		Read data byte or	0	1	1	X	
		Read data byte	1	1	1	X	

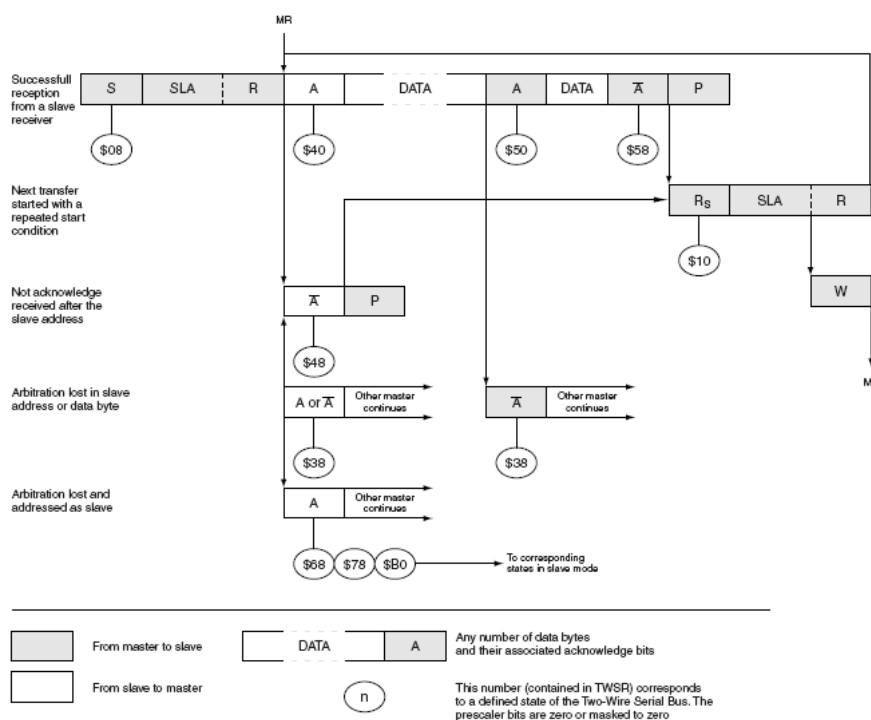


Fig. 14.10 – Formatos e estados do modo de recepção mestre.

Tab. 14.4 – Códigos de *status* para o modo de recepção escravo.

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response				Next Action Taken by TWI Hardware	
		To/from TWDR	To TWCR				
			STA	STO	TWINT		TWEA
0x60	Own SLA+W has been received; ACK has been returned	No TWDR action or No TWDR action	X X	0 0	1 1	0 1	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
0x68	Arbitration lost in SLA+R/W as Master; own SLA+W has been received; ACK has been returned	No TWDR action or No TWDR action	X X	0 0	1 1	0 1	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
0x70	General call address has been received; ACK has been returned	No TWDR action or No TWDR action	X X	0 0	1 1	0 1	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
0x78	Arbitration lost in SLA+R/W as Master; General call address has been received; ACK has been returned	No TWDR action or No TWDR action	X X	0 0	1 1	0 1	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
0x80	Previously addressed with own SLA+W; data has been received; ACK has been returned	Read data byte or Read data byte	X X	0 0	1 1	0 1	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
0x88	Previously addressed with own SLA+W; data has been received; NOT ACK has been returned	Read data byte or Read data byte or  Read data byte or  Read data byte	0 0  1  1	0 0  0  0	1 1  1  1	0 1  0  1	Switched to the not addressed Slave mode; no recognition of own SLA or GCA Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1" Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0x90	Previously addressed with general call; data has been re- ceived; ACK has been returned	Read data byte or Read data byte	X X	0 0	1 1	0 1	Data byte will be received and NOT ACK will be returned Data byte will be received and ACK will be returned
0x98	Previously addressed with general call; data has been received; NOT ACK has been returned	Read data byte or Read data byte or  Read data byte or  Read data byte	0 0  1  1	0 0  0  0	1 1  1  1	0 1  0  1	Switched to the not addressed Slave mode; no recognition of own SLA or GCA Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1" Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0xA0	A STOP condition or repeated START condition has been received while still addressed as Slave	No action	0   0   1   1	0   0   0   0	1   1   1   1	0   1   0   1	Switched to the not addressed Slave mode; no recognition of own SLA or GCA Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1" Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free

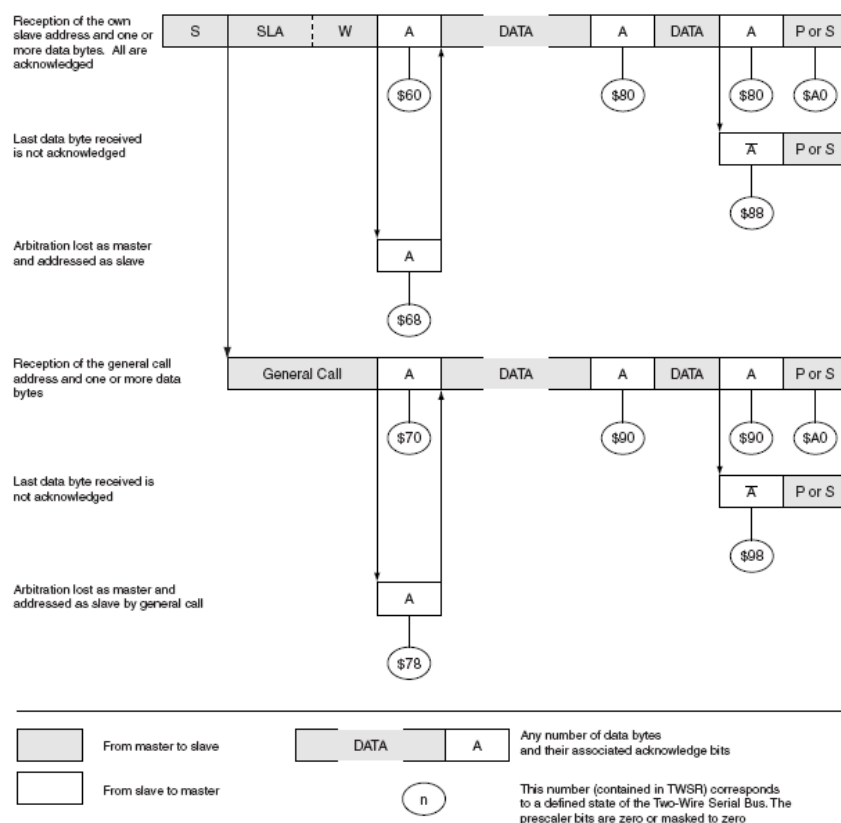


Fig. 14.11 – Formatos e estados do modo de recepção escravo.

Tab. 14.5 – Códigos de *status* para o modo de transmissão escravo.

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0xA8	Own SLA+R has been received; ACK has been returned	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received Data byte will be transmitted and ACK should be received
		Load data byte	X	0	1	1	
0xB0	Arbitration lost in SLA+R/W as Master; own SLA+R has been received; ACK has been returned	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received Data byte will be transmitted and ACK should be received
		Load data byte	X	0	1	1	
0xB8	Data byte in TWDR has been transmitted; ACK has been received	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received Data byte will be transmitted and ACK should be received
		Load data byte	X	0	1	1	
0xC0	Data byte in TWDR has been transmitted; NOT ACK has been received	No TWDR action or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1" Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
		No TWDR action or	0	0	1	1	
		No TWDR action or	1	0	1	0	
		No TWDR action	1	0	1	1	
0xC8	Last data byte in TWDR has been transmitted (TWEA = "0"); ACK has been received	No TWDR action or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1" Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
		No TWDR action or	0	0	1	1	
		No TWDR action or	1	0	1	0	
		No TWDR action	1	0	1	1	

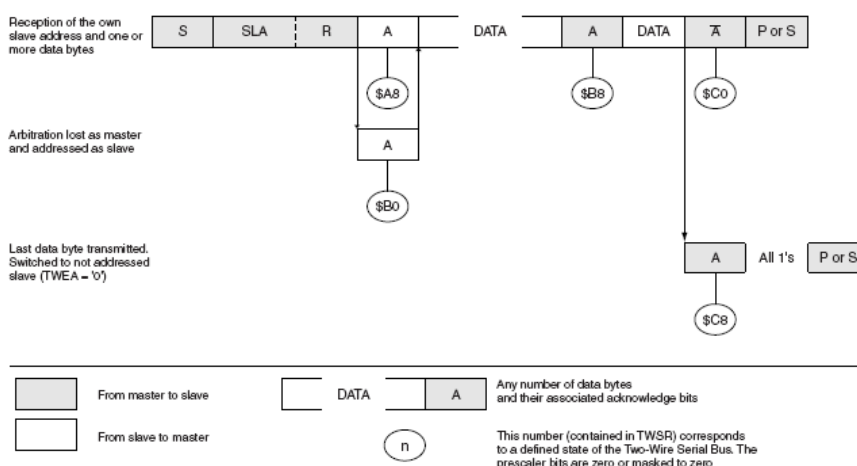


Fig. 14.12 – Formatos e estados do modo de transmissão escravo.

### 14.3 I2C VIA SOFTWARE SOMENTE

Em  $\mu$ controladores mais simples, pode não haver o módulo de hardware específico para a comunicação I2C. Este problema é solucionado com o gerenciamento total da comunicação via software. Esta ideia é válida para qualquer protocolo de comunicação, desde que o  $\mu$ controlador seja capaz de lidar com as taxas de transmissão. Entretanto, realizar todo o controle de um protocolo de comunicação exclusivamente via software pode sobrecarregar o  $\mu$ controlador e diminuir seu desempenho, bem como exigir grande esforço computacional (este é o caso da USB).

A seguir as rotinas para o I2C são apresentadas, podendo ser empregadas em qualquer  $\mu$ controlador, bastando apenas adaptar a linguagem, se for o caso.

```
//=====//
//      ROTINAS PARA COMUNICAÇÃO I2C VIA SOFTWARE      //
//=====//
//Os pinos SCL e SDA são definidos pelo usuário, neste caso, PA0=SDA e PA1=SCL //
//-----//
void seta_scl()          //seta o pino SCL
{
    set_bit(PORTA,SCL);
    _delay_us(5); //tempo p/ estabilizar dado, ajustado de acordo com a máxima freq. de trabalho
}
//-----//
void apaga_scl()         //apaga o pino SCL
{
    clr_bit(PORTA,SCL);
    _delay_us(5);
}
//-----//
void seta_sda()          //seta o pino SDA
{
    set_bit(PORTA,SDA);
    _delay_us(5);
}
//-----//
void apaga_sda()         //apaga o pino SCL
{
    clr_bit(PORTA,SDA);
    _delay_us(5);
}
//-----//
void i2c_start()         //coloca o barramento na condição de start
{
    apaga_scl();          //coloca a linha de clock em nível 0
    seta_sda();           //coloca a linha de dados em alta impedância (1)
    seta_scl();           //coloca a linha de clock em alta impedância (1)
}
```

```

    apaga_sda();          //coloca a linha de dados em nível 0
    apaga_scl();          //coloca a linha de clock em nível 0
}
//-----
void i2c_stop()          //coloca o barramento na condição de stop
{
    apaga_scl();          //coloca a linha de clock em nível 0
    apaga_sda();          //coloca a linha de dados em nível 0
    seta_scl();           //coloca a linha de clock em alta impedância (1)
    seta_sda();           //coloca a linha de dados em alta impedância (1)
}
//-----
void i2c_nack()          //coloca sinal de não reconhecimento (nack) no barramento
{
    seta_sda();           //coloca a linha de dados em alta impedância (1)
    seta_scl();           //coloca a linha de clock em alta impedância (1)
    apaga_scl();          //coloca a linha de clock em nível 0
}
//-----
unsigned char i2c_le_ack() //efetua a leitura do sinal de ack/nack
{
    unsigned char estado;

    seta_sda();           //coloca a linha de dados em alta impedância (1)
    seta_scl();           //coloca a linha de clock em alta impedância (1)

    DDRA = 0b11111110;    //pino SDA como entrada
    estado = tst_bit(PINA,SDA); //lê o bit (ack/nack)
    DDRA = 0xFF;          //PORTA como saída (pode mudar conforme projeto)

    apaga_scl();          //coloca a linha de clock em nível 0
    return estado;
}
//-----
void i2c_escreve_byte(unsigned char dado) //envia um byte pelo barramento I2C
{
    unsigned char d, conta=8;

    apaga_scl();          //coloca SCL em 0

    while(conta)          //esta rotina pode ser melhorada.
    {                     //envia primeiro o MSB
        d = dado & 0b10000000; //AND entre o dado e 0b10000000 para testar o bit mais significativo
        dado = dado<<1;        //rotaciona para a esquerda para testar o próximo bit

        if (d!=0)           //se o bit for 1 seta o dado
            set_bit(PORTA,SDA);
        else                //senão o zera
            clr_bit(PORTA,SDA);

        seta_scl();         //dá um pulso em SCL
        conta--;
        apaga_scl();
    }
    seta_sda();            //ativa SDA
}
//-----
unsigned char i2c_le_byte() //recebe um byte pelo barramento I2C
{
    unsigned char bytelido, bit_lido, conta = 8;

    bytelido = 0;
    apaga_scl();
    seta_sda();

    while (conta)
    {
        seta_scl();         //ativa SCL
        DDRA = 0b11111110;  //pino SDA como entrada

        bytelido = bytelido<<1; //rotaciona um bit a esquerda
        bit_lido = tst_bit(PINA,SDA); //lê o pino
        bytelido = bytelido | bit_lido; //OU bit a bit

        conta--;
        DDRA = 0xFF;        //PORTA como saída
        apaga_scl();        //desativa SCL
    }
    return bytelido;
}
//=====

```

## Exercício:

**14.1** – Elaborar um programa para ler e escrever em um DS1307 (*Real Time Clock*) empregando o módulo TWI do ATmega8, conforme Fig. 14.13. Configurar a saída SOUT do RTC para uma frequência de 1Hz.

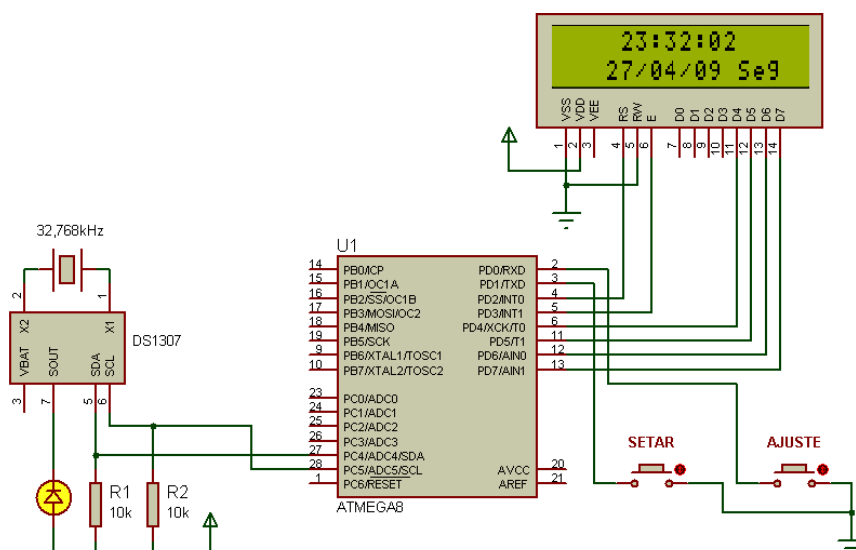


Fig. 14.13 – Empregando o barramento I2C do ATmega8 para comunicação com o DS1307.

## 15. COMUNICAÇÃO 1 FIO (VIA SOFTWARE)

A comunicação 1 fio, conhecida por 1 *wire*, foi desenvolvida pela MAXIM e permite através de uma única via conectar diferentes circuitos integrados dedicados. A comunicação é realizada em um único fio, utilizando espaços de tempos adequados para cada nível do sinal digital. Os CIs 1 *wire* podem ser empregados com o uso de apenas 3 vias (VCC, GND e sinal) ou ainda, apenas duas vias (GND e sinal) quando se emprega a alimentação parasita.

Os componentes 1 wire possuem as seguintes características:

- 1 única via para enviar e receber dados, sem a necessidade de uma via de *clock*.
- Cada CI possui um número único de identificação gravado na fábrica.
- A alimentação pode ser feita pela via de sinal (alimentação parasita).
- Suportam vários dispositivos na mesma linha, ver Fig. 15.1

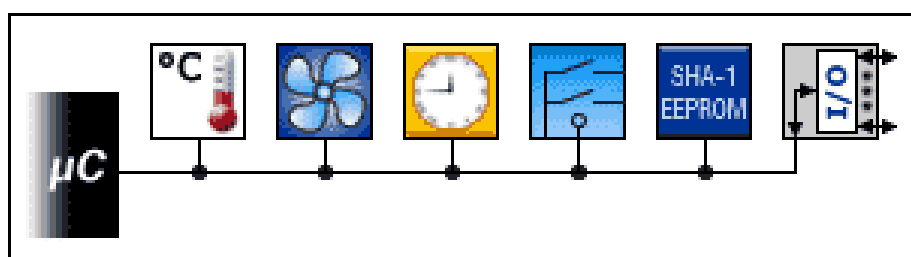


Fig. 15.1 – Conexão 1 fio para vários dispositivos.



Os dispositivos disponíveis para o trabalho 1 *wire* são: memórias, sensores de temperatura e chaves de temperatura, conversor A/D de 4 canais, relógios de tempo real, protetores de baterias, conversores I2C-1 *wire* (expansão de I/Os), e outros.

A seguir são apresentadas algumas figuras para ilustrar o funcionamento do protocolo 1 *wire* aplicado ao DS18S20 (sensor de temperatura). O detalhamento completo é encontrado no manual do fabricante. Após, tem-se o código para trabalho com a interface 1 fio para o circuito da Fig. 15.4.

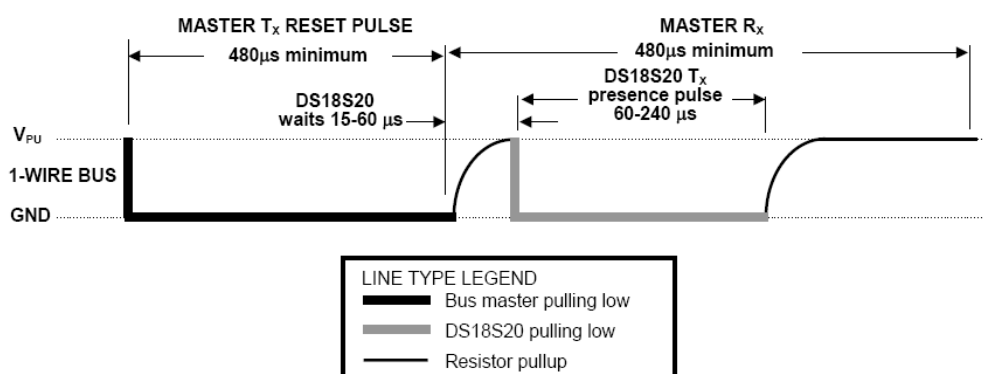


Fig. 15.2 – Diagrama temporal de inicialização.

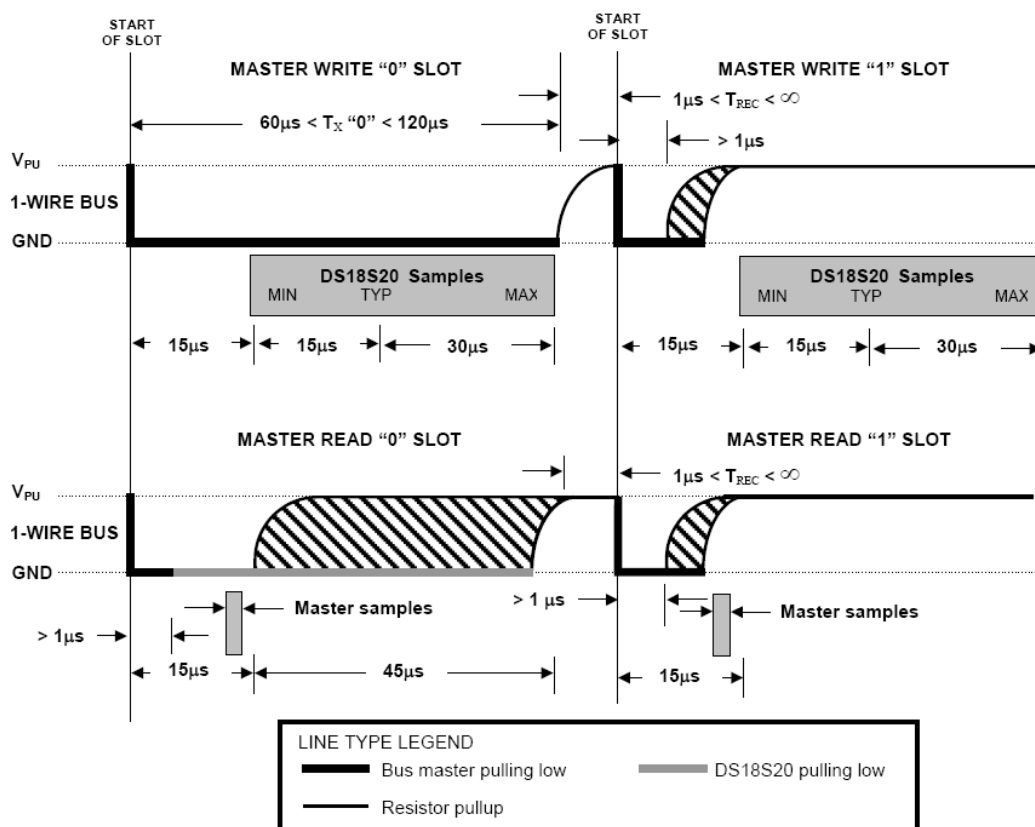


Fig. 15.3 – Diagrama temporal para leitura e escrita.

```
//=====//
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#define      set_bit(address,bit) (address|=(1<<bit))
#define      clr_bit(address,bit) (address&=~(1<<bit))
#define      tst_bit(address,bit) (address&(1<<bit))
#define      DQ PA0          //para o DS18S20 (sensor de temperatura)
//-----//
//          ROTINAS PARA COMUNICAÇÃO 1 WIRE          //
//-----//
unsigned char reset()          //reseta os dispositivos no barramento
{
    unsigned char presente;

    DDRA = 0xFF;              //DQ como saída
    clr_bit(PORTA,DQ);         //DQ em nível zero por 480us
    _delay_us(480);
    DDRA = 0b11111110; //pino SDA como entrada, o resistor de pull-up mantém DQ em nível alto
    _delay_us(60);
    presente = tst_bit(PINA,DQ); //lê DQ, 0 = dispositivo presente, 1 = nenhum dispositivo detectado
    _delay_us(240);           //o pulso de presença pode ter 240us
    return(presente);
}
//-----//
void alimenta_barramento() //força o barramento em nível alto. Utilizado com dispositivos alimentados
{
    //no modo parasita
    DDRA = 0xFF;
    set_bit(PORTA,DQ);
    _delay_ms(750);
    DDRA = 0b11111110;      //ativa pull-up
}
//-----//
unsigned char le_bit() //lê um bit do barramento 1 wire
{
    unsigned char bit_lw;
    DDRA = 0xFF;           // dá um pulso na linha, inicia quadro de leitura
    clr_bit(PORTA,DQ);      // coloca saída em zero
    DDRA = 0b11111110; //pino SDA como entrada, o resistor de pull-up mantém DQ em nível alto
    _delay_us(15);         // aguarda o dispositivo colocar o dado na saída

    bit_lw = tst_bit(PINA,DQ); //lê DQ
    return (bit_lw);         //retorna o dado
}
//-----//
void escreve_bit(unsigned char bit_lw) //escreve um bit no barramento 1 wire
{
    DDRA = 0xFF;
    clr_bit(PORTA,DQ);      //linha DQ em zero

    if (bit_lw==1) set_bit(PORTA,DQ); //coloca dado 1 na saída
    _delay_us(120);

    set_bit(PORTA,DQ);
    DDRA = 0b11111110;      //ativa pull-up
}
//-----//
unsigned char le_byte() //lê um byte do barramento 1 wire
{
    unsigned char i, dado = 0;

    for (i=0;i<8;i++)      //lê oito bits iniciando pelo bit menos significativo
    {
        if (le_bit()==1) dado|=0x01<<i;
        _delay_us(90);      //aguarda o fim do quadro de leitura do bit atual
    }
    return (dado);
}
//-----//
void escreve_byte(char dado) //escreve um byte no barramento 1 wire
{
    unsigned char i, temp;

    for (i=0; i<8; i++)    //envia o byte iniciando do bit menos significativo
    {
        temp = dado>>i;      //desloca o dado 1 bit à direita
        temp &= 0x01;        //isola o bit 0 (LSB)
        escreve_bit(temp);   //escreve o bit no barramento
    }
}
//=====//
```

## Exercício:

15.1 – Elaborar um programa para trabalhar com o sensor de temperatura DS18S20, conforme Fig. 15.4.

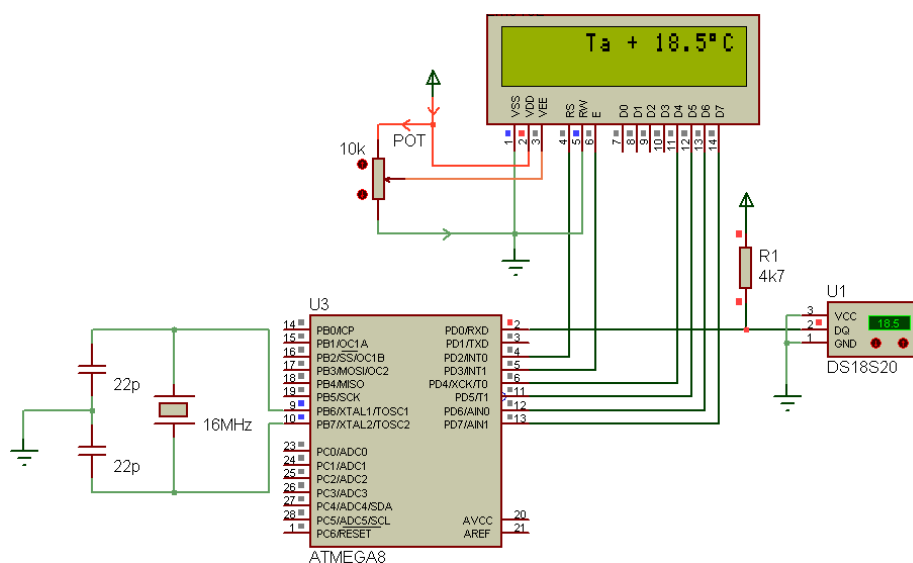


Fig. 15.4 – Circuito com o DS18S20.

## 16. COMPARADOR ANALÓGICO

O comparador analógico compara os valores de entrada no pino positivo AIN0 e no pino negativo AIN1. Quando a tensão no pino positivo é maior que no pino negativo, a saída do comparador analógico vai a 1. Esta saída pode ser ajustada para disparar um evento de captura do Temporizador/Contador1. Adicionalmente o comparador pode disparar sua própria interrupção. O diagrama em blocos do comparador é apresentado na Fig. 16.1.

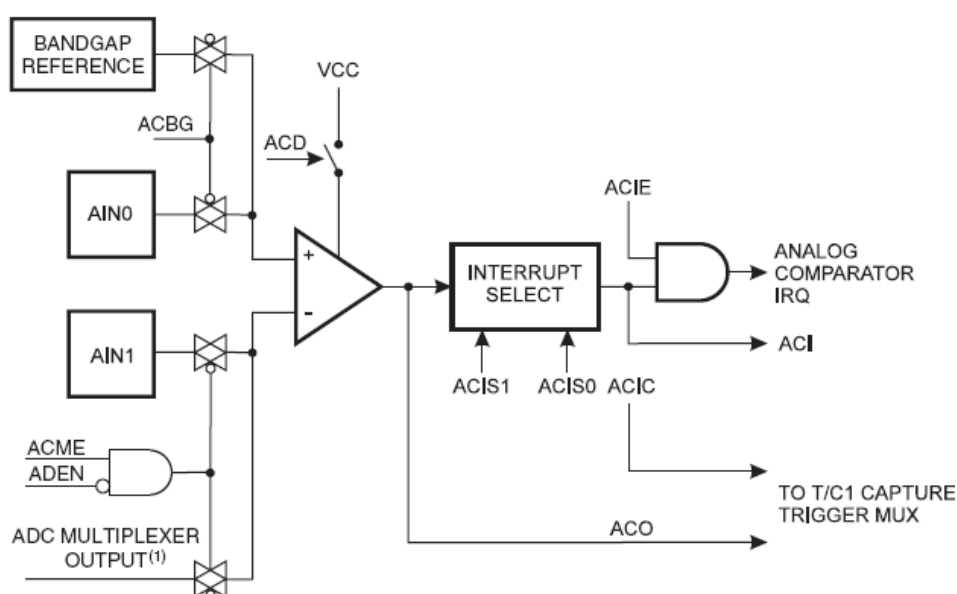


Fig. 16.1 – Diagrama do comparador analógico.

## REGISTRADORES

### SFIOR – Special Function I/O Register

Bit	7	6	5	4	3	2	1	0
<b>SFIOR</b>	–	–	–	–	ACME	PUD	PSR2	PSR10
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

#### **BIT 3– ACME: Analog Comparator Multiplexer Enable**

Quando este bit é colocado em nível lógico 1 e o conversor A/D é desligado (ADEN no ADCSRA é zero), o multiplexador do A/D seleciona a entrada negativa do comparador analógico. Quando colocado em nível lógico zero, AIN1 é aplicado na entrada negativa do comparador.

### ACSR – Analog Comparator Control and Status Register

Bit	7	6	5	4	3	2	1	0
<b>ACSR</b>	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	N/A	0	0	0	0	0

#### **BIT 7– ACD: Analog Comparator Disable**

Quando este bit for 1 a alimentação do comparador analógico é desligada. Este bit pode ser setado a qualquer momento para desligar o comparador e economizar energia. Quando ACD for alterado a interrupção do comparador deve estar desabilitada, caso contrário uma interrupção pode ocorrer.

#### **BIT 6– ACBG: Analog Comparator Bandgap Select**

Quando este bit for 1 uma referência fixa de tensão substitui a entrada positiva do comparador analógico (aprox. 1,3V). Quando ACBG=0, AIN0 é aplicado na entrada positiva do comparador.

#### **BIT 5– ACO: Analog Comparator Output**

A saída do comparador analógico é sincronizada e então diretamente conectada ao ACO. A sincronização introduz um atraso de 1 a 2 ciclos de *clock*.

#### **BIT 4– ACI: Analog Comparator Interrupt Flag**

Este bit é setado por hardware quando houver um evento de disparo do comparador.

#### **BIT 3– ACIE: Analog Comparator Interrupt Enable**

ACIE=1 habilita a interrupção do comparador em conjunto com o bit I do SREG.

#### **BIT 2– ACIC: Analog Comparator Input Capture Enable**

Este bit é setado para habilitar a função de entrada de captura do Temporizador/Contador1.

#### **BIT 1,0– ACIS1/0: Analog Comparator Interrupt Mode Select**

Estes bits definem qual evento irá disparar a interrupção do comparador analógico, ver Tab. 16.1. Quando estes bits forem alterados, a interrupção deve estar desabilitada para evitar uma interrupção indesejada.

Tab. 16.1 – Ajuste dos bits ACIS1 e ACIS0.

ACIS1	ACIS0	Interrupt Mode
0	0	Comparator Interrupt on Output Toggle
0	1	Reserved
1	0	Comparator Interrupt on Falling Output Edge
1	1	Comparator Interrupt on Rising Output Edge

É possível selecionar qualquer um dos pinos do AD – ADC7..0 para substituir a entrada negativa do comparador analógico. O multiplexador do A/D é usado para selecionar essas entradas e obviamente o A/D deve ser desligado para utilizar essa característica. Se o bit de habilitação do multiplexador do comparador analógico (ACME no SFIOR) estiver setado e o A/D desligado (ADEN no ADCSRA é zero), MUX2..0 no registrador ADMUX seleciona o pino de entrada para substituir o pino negativo do comparador. Se ACME estiver limpo ou ADEN setado, AIN1 é aplicado à entrada negativa do comparador analógico.

Tab. 16.2 – Multiplexação da entrada do comparador analógico.

ACME	ADEN	MUX2..0	Analog Comparator Negative Input
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6 only in TQFP and QFN/MLF Package.
1	0	111	ADC7 only in TQFP and QFN/MLF Package.

## 16.1 MEDINDO RESISTÊNCIA E CAPACITÂNCIA

O comparador analógico pode ser empregado para medir resistências e capacitâncias empregando um circuito RC, onde o tempo de carga e/ou descarga do capacitor será medido em relação a uma tensão de referência empregando-se o modo de captura do T/C1.

### RESISTÊNCIA

O circuito da Fig. 16.2 ilustra um circuito para medição precisa de uma resistência desconhecida  $R_X$ , com base no valor conhecido de um resistor de referência  $R_{ref}$  e um capacitor de valor exato conhecido.

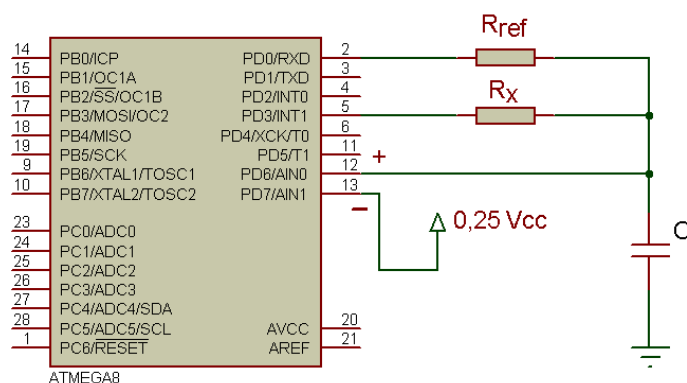


Fig. 16.2 – Circuito para medir uma resistência desconhecida  $R_X$ .

Para o funcionamento o comparador analógico deve disparar a captura do T/C1 quando a tensão na entrada positiva cair abaixo da tensão da entrada negativa, neste caso,  $0,25 V_{CC}$ .

Inicialmente o capacitor é carregado pelos dois resistores  $R_{ref}$  e  $R_X$ , os pinos PD0 e PD3 são colocados como saída em nível alto. O capacitor se carrega pela resistência equivalente do paralelo de  $R_{ref}$  e  $R_X$  ( $R_{eq}$ ). Um tempo suficiente de carga é de aproximadamente  $7 \cdot R_{eq} \cdot C$ .

Após carregado o capacitor o mesmo será descarregado por  $R_{ref}$ . Antes disso, o valor de contagem do T/C1 é armazenado. Em seguida o pino PD3 é configurado como entrada (alta impedância) e o pino PD0 é colocado em zero para descarga do capacitor. Quando a tensão do capacitor cai abaixo de  $0,25 V_{CC}$  a saída do comparador provoca a captura do T/C1. A diferença do valor dessa captura e o valor do T/C1 no início da descarga é o tempo de descarga do capacitor ( $t_{ref}$ ).

Novamente o capacitor deve ser carregado por  $R_{ref}$  e  $R_X$ , como feito no início do processo. Após isso, é a vez de descarregar o capacitor por  $R_X$  mantendo o pino do  $R_{ref}$  como entrada. O processo de contagem do tempo se faz do mesmo modo anterior e tem-se o tempo de descarga do capacitor por  $R_X$  ( $t_X$ ). A Fig. 16.3 ilustra todo o processo.

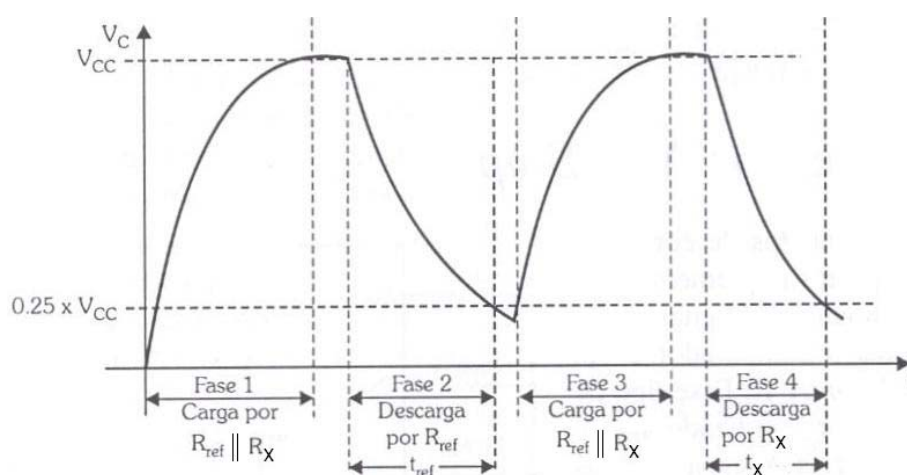


Fig. 16.3 – Carga do capacitor por  $R_{ref} \parallel R_X$  e descarga por  $R_{ref}$  e  $R_X$ .

Como a tensão de descarga de um capacitor é dada por:

$$V_C = V_{CC} \cdot e^{\left(\frac{-t}{R \cdot C}\right)} \quad , [V] \quad (16.1)$$

e a tensão final de descarga de  $0,25 V_{CC}$  é a mesma para o  $R_X$  e  $R_{ref}$ , então:

$$V_{CC} \cdot e^{\left(\frac{-t_{ref}}{R_{ref} \cdot C}\right)} = V_{CC} \cdot e^{\left(\frac{-t_X}{R_X \cdot C}\right)} \quad (16.2)$$

O que resulta:

$$R_X = R_{ref} \cdot \frac{t_X}{t_{ref}} \quad . [\Omega] \quad (16.3)$$

Para precisão no circuito de medição, os componentes utilizados devem ser de precisão, bem como a tensão de referência também deve ser provida com exatidão. A medição de resistência com o método acima pode ser empregada para a leitura do valor da resistência de sensores, por exemplo.

## CAPACITOR

Para o cálculo de uma capacitância desconhecida, o circuito da Fig. 16.2 continua sendo empregado, com mudanças para avaliar o tempo de carga do capacitor desconhecido. Agora o comparador analógico deve disparar o modo de captura do T/C1 quando o valor da entrada positiva ultrapassar em mais de 0,63 o valor da entrada negativa, a Fig. 16.4 ilustra o circuito para análise.

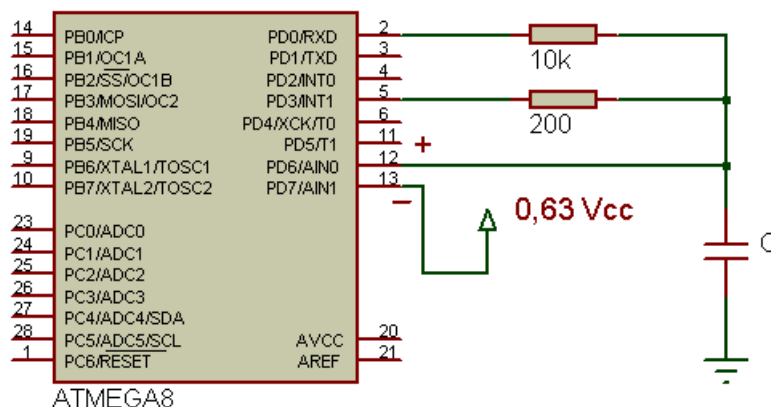


Fig. 16.4 – Medindo uma capacitância.

A tensão de carga de um capacitor é dada por:

$$V_C = V_{CC} \cdot [1 - e^{\left(\frac{-t}{R \cdot C}\right)}] \quad [V] \quad (16.4)$$

Fazendo-se  $t = R \cdot C$ , resulta:

$$V_C = V_{CC} \cdot [1 - e^{-1}] \cong 0.63 \cdot V_{CC} \quad (16.5)$$

Assim, para se obter o valor de  $C$  basta saber o tempo de carga do capacitor até disparar o comparador:

$$C = \frac{t_{carga}}{R_C} \quad [F] \quad (16.6)$$

Para funcionamento, primeiro deve-se descarregar o capacitor pelos dois resistores do circuito (pinos PD0 e PD3 como saídas em zero), aqui deve ser estimado um tempo de descarga, limitando o valor máximo de capacitância que pode ser medida. Após a descarga do capacitor, o pino PD0 deve ser colocado em nível alto e o pino PD3 como entrada, para que o capacitor se carregue somente pelo resistor de  $10\text{ k}\Omega$  ( $R_C$ ). O tempo de carga é computado entre o instante que se começa a carregar o capacitor e o instante que o comparador analógico dispara o evento de captura do T/C1.

### **Exercícios:**

- 16.1** – Elaborar um programa para executar uma determinada função quando uma tecla for pressionada. Use o comparador analógico para gerar a interrupção para o botão.
- 16.2** – Elaborar um programa para medir uma resistência desconhecida empregando o circuito da Fig. 15.2.
- 16.3** – Como os circuitos apresentados para a medição de resistência e capacitância podem ser utilizados para se medir indutância? Quais as técnicas que podem ser empregadas para aumentar a resolução desses circuitos e como melhorá-los?

## **17. CONVERSOR ANALÓGICO-DIGITAL**

O conversor A/D do ATmega8 emprega o processo de aproximações sucessivas para converter um sinal analógico em digital. Suas principais características são:

- 10 bits de resolução (1024 pontos).
- Precisão de  $\pm 2$  LSB.
- Tempo de conversão de 13-260  $\mu\text{s}$ .
- Até 15 kSPS na resolução máxima.
- 6 canais de entrada multiplexados (+2 nos encapsulamentos TQFP e QFN/MLF).
- Faixa de tensão de entrada de 0 até  $V_{CC}$ .
- Tensão de referência selecionável de 2,56 V.
- Modo de conversão simples ou contínua.
- Interrupção ao término da conversão.
- Eliminador de ruído.

O valor mínimo representado digitalmente é 0 V (GND) e o valor máximo corresponde à tensão do pino AREF menos 1 LSB; opcionalmente, a tensão do pino AVCC ou a tensão interna de referência de 2,56V. A Fig. 17.1 ilustra o diagrama em blocos do conversor A/D.



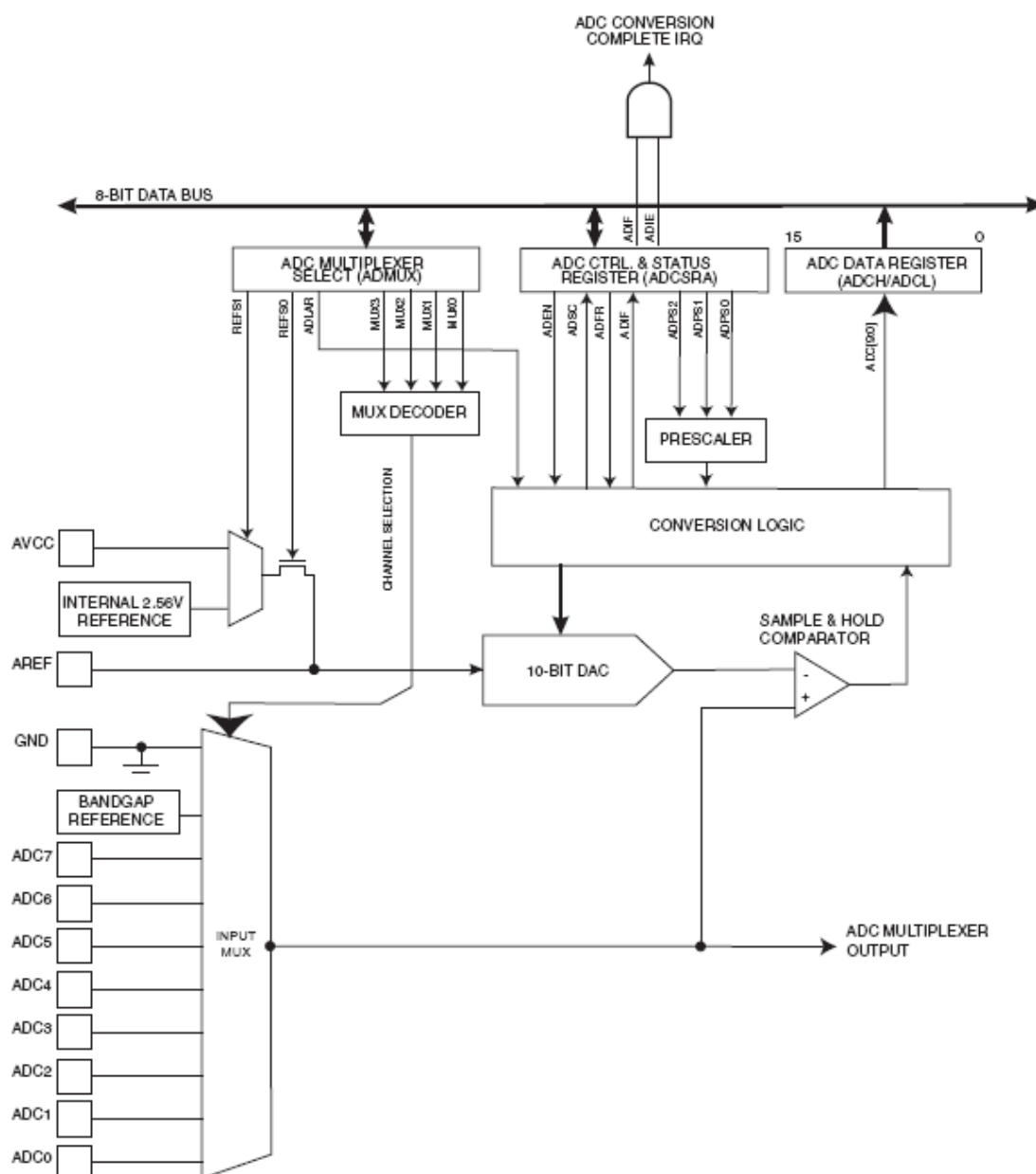


Fig. 17.1 – Diagrama do conversor A/D do ATmega8.

O A/D produz um resultado de 10 bits apresentado no registrador de resultado ADC (ADCH e ADCL). Por definição o resultado é apresentado com ajuste a direita (ADCL tem os 8 bits LSB), opcionalmente o resultado pode ter ajuste a esquerda (ADCH com os 8 bits MSB e os bits 7:6 do ADCL, com os LSB). Isto é feito ajustando o bit ADLAR no registrador ADMUX. O ajuste a esquerda é interessante quando se deseja apenas 8 bits de resolução do AD, bastando apenas, então, ler ADCH. Caso contrário, ADCL deve ser lido primeiro para garantir que o conteúdo do registrador pertence à mesma conversão. Quando a leitura do ADCL é realizada os registradores de resultado são bloqueados para atualização até que o ADCH seja lido. Neste caso o conteúdo da conversão não é salvo (perdido). A interrupção irá ocorrer do mesmo modo se os registradores estiverem bloqueados e o resultado de uma nova conversão for perdido.

Uma conversão simples é iniciada quando o bit ADSC é colocado em 1 no registrador ADCSRA. Esse bit se mantém em 1 durante a conversão e é limpo por hardware ao final dessa. Se um canal diferente é escolhido para conversão, o A/D irá terminar a conversão corrente antes de mudar de canal. O resultado para uma conversão contínua é dado por:

$$AD = \frac{V_{IN} \cdot 1024}{V_{REF}}, \quad (17.1)$$

onde  $V_{IN}$  é a tensão para conversão na entrada no pino e  $V_{REF}$  é a tensão selecionada de referência. O valor 0x000 representa o terra e o valor 0x3FF representa  $V_{REF}$  menos 1 LSB.

No modo de conversão contínua (*Free Running*), o A/D é constantemente amostrado e os registradores de dados atualizados (ADFR=1 no ADCSRA). A conversão começa escrevendo-se 1 no bit ADSC.

O circuito de aproximações sucessivas do A/D requer uma frequência de entrada entre 50 kHz e 200 kHz para obter a resolução máxima. Se uma resolução menor que 10 bits é desejada, uma frequência maior que 200 kHz pode ser empregada para se ter uma maior taxa de amostragem. Sinais de entrada com frequências maiores que a frequência de Nyquist ( $f_{AD}/2$ - metade da freq. de amostragem) gerarão erros (*aliasing*). Eles devem ser previamente filtrados por um filtro passa baixas para eliminar o conteúdo de frequência acima da capacidade do A/D, e somente depois amostrados.

O A/D contém um módulo com *prescaler*, que aceita qualquer frequência de CPU acima de 100 kHz. O *prescaler* é ajustado nos bits ADPS do ADCSRA e começa a valer quando o A/D é habilitado pelo bit ADEN. Enquanto este bit for 1 o *prescaler* estará rodando.

Quando uma conversão simples é iniciada, colocando-se o bit ADSC em 1, a conversão inicia na próxima borda de subida do *clock* do A/D. Uma conversão normal leva 13 ciclos de *clock* do A/D. A primeira conversão leva 25 ciclos para inicializar o circuito analógico. A amostragem e retenção (*sample-and-hold*) leva 1,5 ciclos de clock após início de uma conversão normal e 13,5 ciclos após o início da primeira conversão. Quando o resultado da conversão estiver completo, o resultado é escrito nos registradores de resultado e o bit ADIF é colocado em 1 e o ADSC é limpo. Os diagramas de tempo para a conversão simples são apresentados na Fig. 17.2.

Quando uma conversão contínua estiver habilitada, uma nova conversão é iniciada logo após uma ser completada, enquanto ADSC=1. O diagrama de tempo da conversão contínua é apresentado na Fig. 17.3.

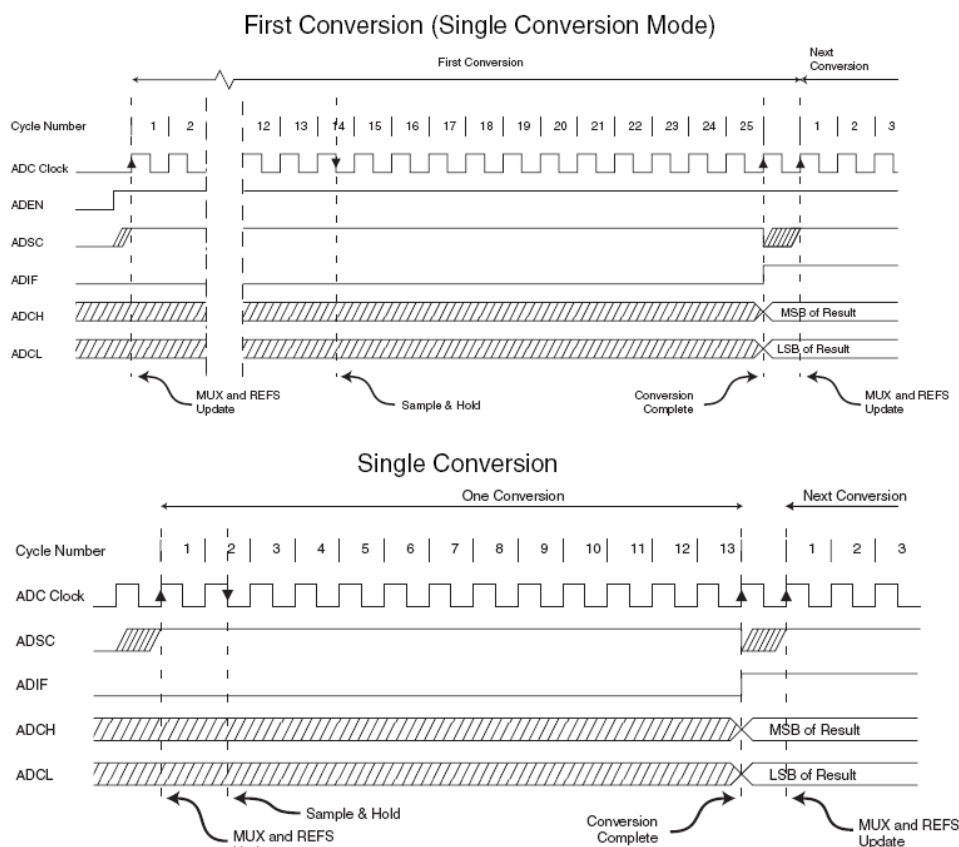


Fig. 17.2 – Diagramas de tempo para a conversão simples.

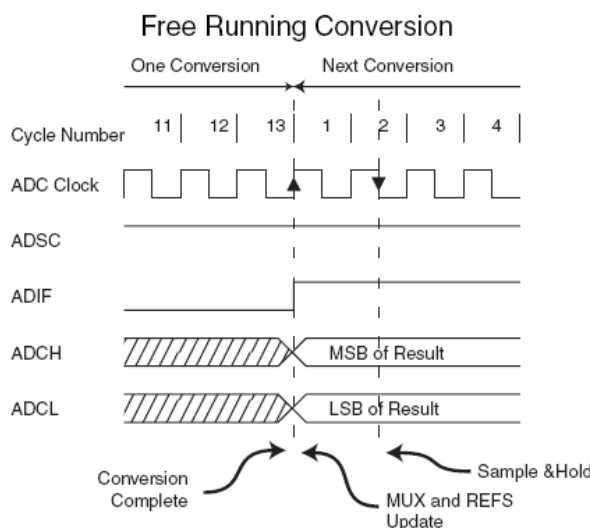


Fig. 17.3 – Diagrama de tempo para a conversão contínua.

Cuidados devem ser tomados quando se muda o canal para conversão: o registrador ADMUX responsável pela mudança pode ser atualizado seguramente quando:

1. Os bits ADFR e ADEN foram iguais a zero;
2. Durante a conversão, pelos menos um ciclo de *clock* do A/D após início da conversão;
3. Após a conversão, antes do *flag* de interrupção ser limpo.

Quando atualizado nessas condições, o ADMUX irá afetar a próxima conversão do A/D.



## REGISTRADORES

### ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
<b>ADMUX</b>	<b>REFS1</b>	<b>REFS0</b>	<b>ADLAR</b>	<b>–</b>	<b>MUX3</b>	<b>MUX2</b>	<b>MUX1</b>	<b>MUX0</b>
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

#### **BIT 7,6 – REFS: Reference Selection Bit**

Estes bits selecionam a fonte de tensão para o A/D, conforme Tab. 17.1. Se uma mudança ocorrer durante uma conversão, a mesma não terá efeito até a conversão ser completada (ADIF no ADCSRA estar setado). A referência interna não pode ser utilizada se um tensão externa estiver sendo aplicada ao pino AREF.

Tab. 17.1 – Bits para a seleção da tensão de referência do AD.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal $V_{ref}$ turned off
0	1	$AV_{CC}$ with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

#### **BIT 5 – ADLAR: ADC Left Adjust Result**

Afeta a representação do resultado da conversão dos registradores de dados do A/D. ADLAR = 1, resultado à esquerda; contrário, à direita. A alteração deste bit afeta imediatamente os registradores de dados.

#### **BIT 3:0 – MUX3:0: Analog Channel Selection Bits**

Selecionam qual entrada analógica será conectada ao A/D, ver Tab. 17.2.

Tab. 17.2 – Seleção do canal de entrada.

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000 — 1101	
1110	1.30V ( $V_{BG}$ )
1111	0V (GND)

### ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0
<b>ADCSRA</b>	<b>ADEN</b>	<b>ADSC</b>	<b>ADFR</b>	<b>ADIF</b>	<b>ADIE</b>	<b>ADPS2</b>	<b>ADPS1</b>	<b>ADPS0</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

#### **BIT 7 – ADEN: ADC Enable**

Habilita o A/D. Se ADEN=0 o A/D é desligado. Desligar o A/D durante uma conversão irá finalizá-la.

#### **BIT 6 – ADSC: ADC Start Conversion**

No modo de conversão simples, ADSC=1 irá iniciar uma conversão. No modo de conversão contínuo, ADSC=1 irá iniciar a primeira conversão. ADSC ficará em 1 durante todo o processo de conversão e será zerado automaticamente ao término dessa. Se o ADSC é escrito ao mesmo tempo em que o A/D é habilitado, a primeira conversão levará 25 ciclos de *clock* do A/D ao invés dos normais 13.

#### **BIT 5– ADFR: ADC Free Running Select**

Habilita o modo de conversão contínua. Neste modo, o A/D amostra e atualiza constantemente os registradores de dados.

#### **BIT 4– ADIF: ADC Interrupt Flag**

Este bit é setado quando uma conversão for completada e o registrador de dados atualizado.

#### **BIT 3– ADIE: ADC Interrupt Enable**

Este bit habilita a interrupção do A/D após a conversão se o bit I do SREG estiver habilitado.

#### **BIT 2:0– ADPS2:0: ADC Prescaler Select Bits**

Determinam a divisão do *clock* da CPU para o *clock* do A/D, conforme Tab. 17.3.

Tab. 17.3 – Seleção da divisão de clock para o A/D.

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

### ADCL/ADCH – Registradores de Dados do A/D

Quando a conversão estiver completa, o resultado é encontrado nestes dois registradores. Quando ADCL é lido, o registrador de dados não é atualizado até que o ADCH seja lido também. Conseqüentemente se o resultado é justificado à esquerda e não se necessitam mais que 8 bits de resolução, basta ler somente ADCH. Caso contrário, ADCL deve ser lido primeiro. A Fig. 16.5 mostra a justificação do resultado da conversão a direita e esquerda, bit ADLAR=0 e ADLAR=1, respectivamente (do registrador ADMUX).

$ADLAR = 0$	Bit	15	14	13	12	11	10	9	8
	<b>ADCH</b>	–	–	–	–	–	–	ADC9	ADC8
	<b>ADCL</b>	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
		7	6	5	4	3	2	1	0
$ADLAR = 1$	Read/Write	R	R	R	R	R	R	R	R
		R	R	R	R	R	R	R	R
	Initial Value	0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0
$ADLAR = 1$	Bit	15	14	13	12	11	10	9	8
	<b>ADCH</b>	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
	<b>ADCL</b>	ADC1	ADC0	–	–	–	–	–	–
		7	6	5	4	3	2	1	0
$ADLAR = 1$	Read/Write	R	R	R	R	R	R	R	R
		R	R	R	R	R	R	R	R
	Initial Value	0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0

Fig. 17.5 – Justificação do resultado à direita e à esquerda nos registradores de resultado do AD.

## Exercícios:

17.1 – Elaborar um programa para ler o sensor de temperatura LM35, conforme circuito da Fig 17.6.

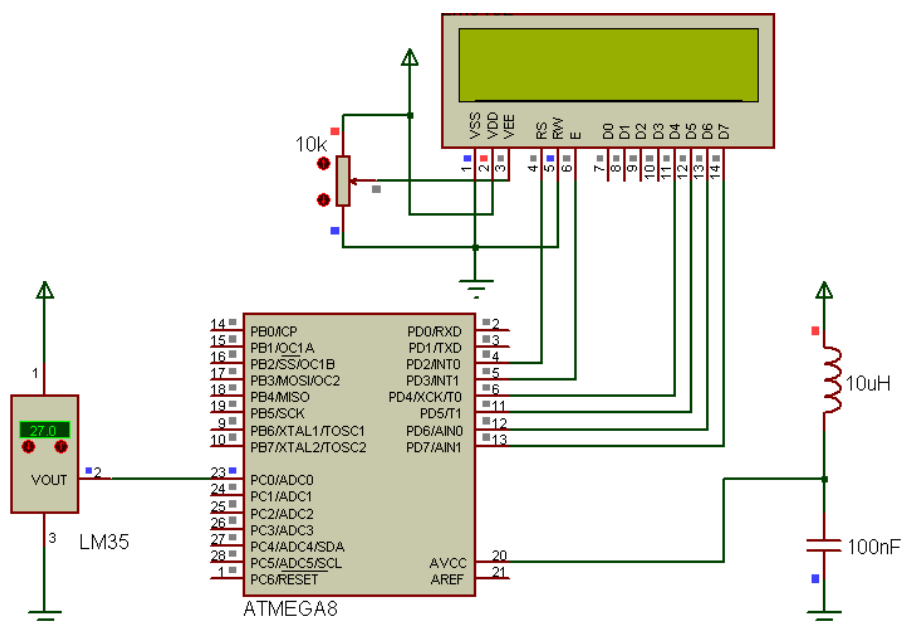


Fig. 17.6 – Empregando o A/D para ler um sensor de temperatura.

17.2 – Empregando o A/D do ATmega8, elaborar um e circuito e respectivo programa para medir com precisão uma resistência desconhecida.

## 18. GRAVANDO O ATMEGA8

Depois do programa desenvolvido e simulado, com o circuito montado, é hora de gravar o código na memória flash do  $\mu$ controlador. Nesse momento, as características que não podem ser acessadas pelo código do usuário estarão disponíveis via programa de gravação. Dentre as definições que podem ser ajustadas estão:

- Fonte de *clock* empregada (externa, interna, cristal, frequência, etc.)
- Ajuste do *Power-on-Reset*.
- Ajuste do *Brown-out*.
- Habilitação do *Watchdog Time*.
- Gravação da EEPROM com valores definidos previamente.
- Proteção do código contra cópia não autorizada.
- Emprego de *Bootloader*.

A fonte de *clock* irá definir que tipo de relógio a CPU irá empregar e qual tipo de componente eletrônico está sendo usado para este fim. Pode-se, por exemplo, definir o uso da fonte interna, sem o uso de componentes externos, bem como sua frequência.

O *Power-on Reset* é empregado para a definição do tempo que a CPU levará para se auto-inicializar, após a energização do circuito. Garante um *reset* preciso na energização; com isso, o emprego de uma rede RC para inicializar fisicamente o  $\mu$ controlador é desnecessária e o pino de reset pode ser ligado diretamente ao VCC.

O *Brown-out* inicializa a CPU caso a tensão de energização caia abaixo de um valor escolhido. Desta forma, garantindo o correto funcionamento do software em caso de instabilidade da energia de alimentação.

O *Watchdog Time* é fundamental para evitar o travamento do software em aplicações onde o travamento pode ser crítico como, por exemplo, em semáforos. O travamento do programa pode ocorrer devido a um ruído eletromagnético ou a um erro do programa. O *Watchdog Time* funciona como um temporizador independente, cujo tempo de estouro é programável. O software deve reiniciá-lo antes da ocorrência do estouro, caso contrário o  $\mu$ controlador é *resetado* e o contador do programa é desviado para o endereço do vetor de *reset*.

É possível gravar a EEPROM com valores pré-definidos, como tabelas, por exemplo. Além disso, é possível garantir que o código não seja copiado empregando os *lock* bits.

O ATmega8 tem dois bytes de fusíveis, para realizar várias das configurações acima, conforme Tab. 18.1.



Tab. 18.1 – Fusíveis de programação do ATmega8.

Fuse High Byte	Bit No.	Description	Default Value
RSTDISBL	7	Select if PC6 is I/O pin or RESET pin	1 (unprogrammed, PC6 is RESET-pin)
WDTON	6	WDT always on	1 (unprogrammed, WDT enabled by WDTCR)
SPIEN	5	Enable Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
CKOPT	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size	0 (programmed)
BOOTSZ0	1	Select Boot Size	
BOOTRST	0	Select Reset Vector	1 (unprogrammed)

Fuse Low Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown out detector trigger level	1 (unprogrammed)
BODEN	6	Brown out detector enable	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	1 (unprogrammed)
SUT0	4	Select start-up time	0 (programmed)
CKSEL3	3	Select Clock source	0 (programmed)
CKSEL2	2	Select Clock source	0 (programmed)
CKSEL1	1	Select Clock source	0 (programmed)
CKSEL0	0	Select Clock source	1 (unprogrammed)

MUITO CUIDADO deve ser tomado ao se gravar os bits RSTDISBL e SPIEN se a programação SPI pretende ser empregada. Se o pino PC6 for utilizado como porta de I/O, a gravação SPI não poderá mais ser realizada. Quando o  $\mu$ controlador estiver soldado no circuito, ele será gravado via SPI, desabilitar acidentalmente essa gravação impedirá a gravação do componente no circuito (o que é muito pior para componentes SMD).

Uma das primeiras configurações que deve ser realizada é a escolha do *clock*. Aqui outro cuidado deve ser tomado na gravação SPI. Se for escolhido o emprego de fonte externa e o circuito do  $\mu$ controlador não dispuser dessa fonte (por exemplo, cristal oscilador), o  $\mu$ controlador não poderá ser gravado novamente porque ele ficará sem a fonte de *clock*. A gravação só poderá ser realizada novamente após o suprimento dessa fonte.

O ATmega8 também possui 6 *lock* bits para proteção da memória de programa, impedindo a cópia do código gravado e a sua replicação. A Tab. 18.2 apresenta as possíveis configurações desses bits (1 significa não programado e 0 significa programado), onde SPM é *Store Program Memory*. Se a memória for protegida, uma nova escrita pode ser feita apagando-se a memória.

Tab. 18.2 – Bits para proteção da memória.

Memory Lock Bits			Protection Type
LB Mode	LB2	LB1	
1	1	1	No memory lock features enabled.
2	1	0	Further programming of the Flash and EEPROM is disabled in Parallel and Serial Programming mode. The Fuse Bits are locked in both Serial and Parallel Programming mode.
3	0	0	Further programming and verification of the Flash and EEPROM is disabled in parallel and Serial Programming mode. The Fuse Bits are locked in both Serial and Parallel Programming modes.
BLB0 Mode	BLB02	BLB01	
1	1	1	No restrictions for SPM or LPM accessing the Application section.
2	1	0	SPM is not allowed to write to the Application section.
3	0	0	SPM is not allowed to write to the Application section, and LPM executing from the Boot Loader section is not allowed to read from the Application section. If Interrupt Vectors are placed in the Boot Loader section, interrupts are disabled while executing from the Application section.
4	0	1	LPM executing from the Boot Loader section is not allowed to read from the Application section. If Interrupt Vectors are placed in the Boot Loader section, interrupts are disabled while executing from the Application section.
BLB1 Mode	BLB12	BLB11	
1	1	1	No restrictions for SPM or LPM accessing the Boot Loader section.
2	1	0	SPM is not allowed to write to the Boot Loader section.
3	0	0	SPM is not allowed to write to the Boot Loader section, and LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section.
4	0	1	LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section.

A gravação pode ser realizada com o emprego do *AVR Studio* ou outro programa que permita a gravação, como o PROGISP, por exemplo. No *AVR Studio* basta ir ao menu <Tools> , <Program AVR>, <Connect>, ver Fig. 18.1. Após isto, escolhe-se o hardware para gravação (este tem que ser comprado ou confeccionado), Fig. 18.2.

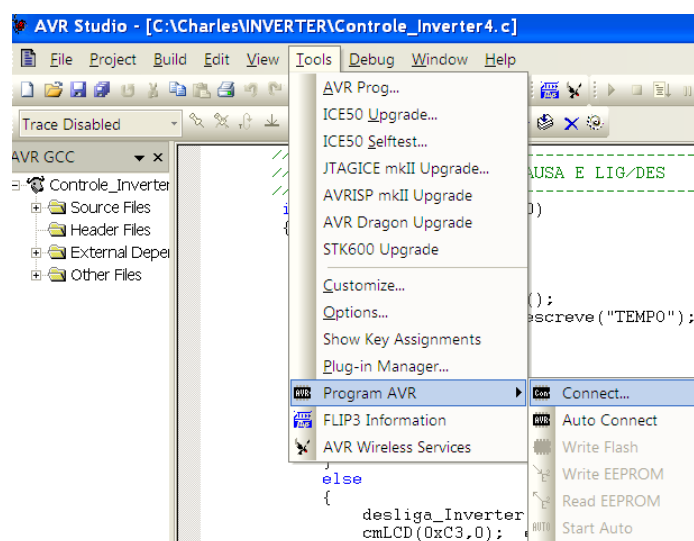


Fig. 18.1 - Conectando o circuito gravador.

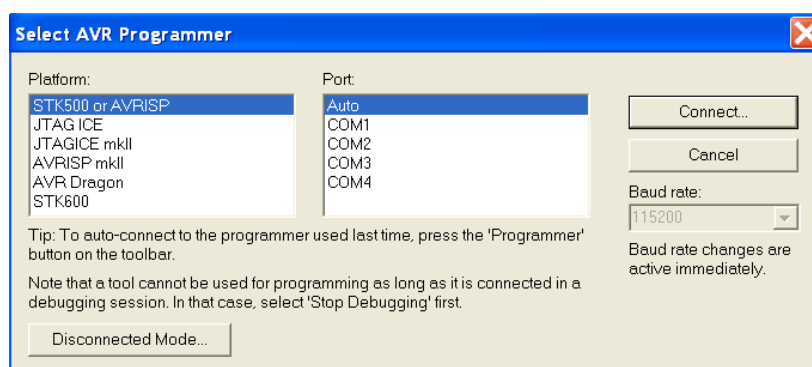


Fig. 18.2 - Escolhendo o tipo do gravador (STK500, por exemplo).

Se o gravador estiver conectado corretamente, será aberta a janela da Fig. 18.3 (ex. para o ATmega88). Na janela aberta, ajusta-se o modo de programação. Considerando o ISP, deve ser ajustada sua frequência de comunicação (menor que  $\frac{1}{4}$  da frequência de operação do  $\mu$ controlador). Se o ajuste de frequência estiver correto, a assinatura do componente poderá ser lida e o componente, gravado.

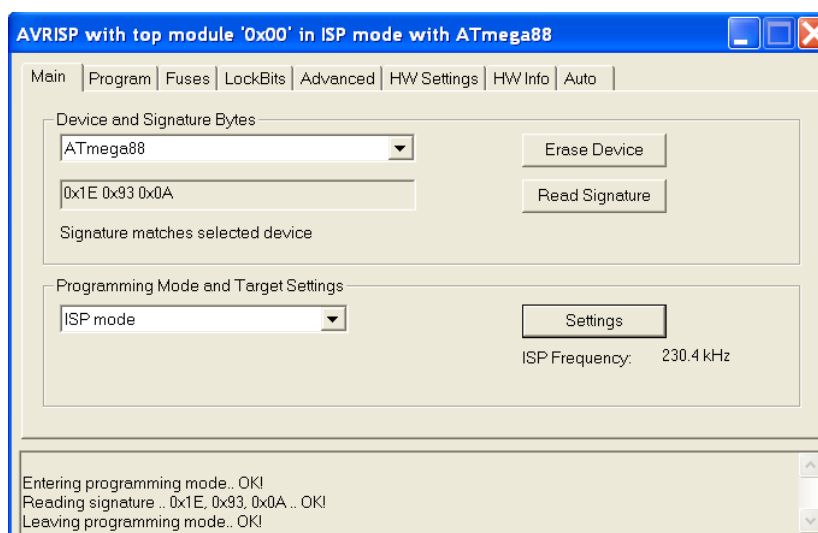


Fig. 18.3 - Janela de gravação.

Primeiro, pode-se tentar gravar o *clock* definindo-se a frequência de trabalho da CPU. A aba <Fuses> contém as especificações de trabalho, conforme mencionado anteriormente, ver Fig. 18.4.

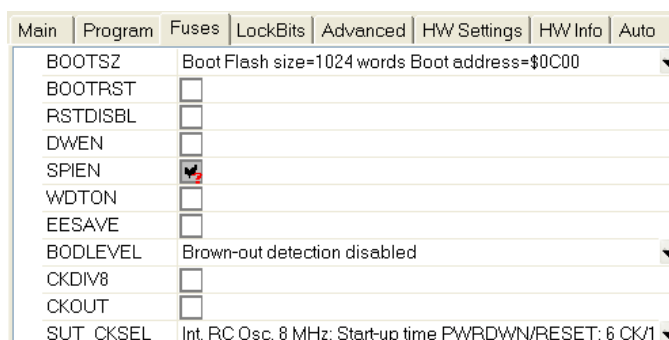


Fig. 18.4 – Ajustando os fusíveis de programação.

A segurança da memória é feita na aba <LockBits>, conforme fig. 18.5.

Main	Program	Fuses	LockBits	Advanced	HW Settings	HW Info	Auto
LB		No memory lock features enabled					▼
BLB0		No lock on SPM and LPM in Application Section					▼
BLB1		No lock on SPM and LPM in Boot Section					▼

Fig. 18.5 – Bits para proteção da memória de programa.

A gravação é feita na aba <Program>, onde se especifica o arquivo de gravação \*.hex e se apaga a memória. Caso a memória esteja bloqueada, ela deverá ser apagada antes da gravação. Ver Fig. 18.6.

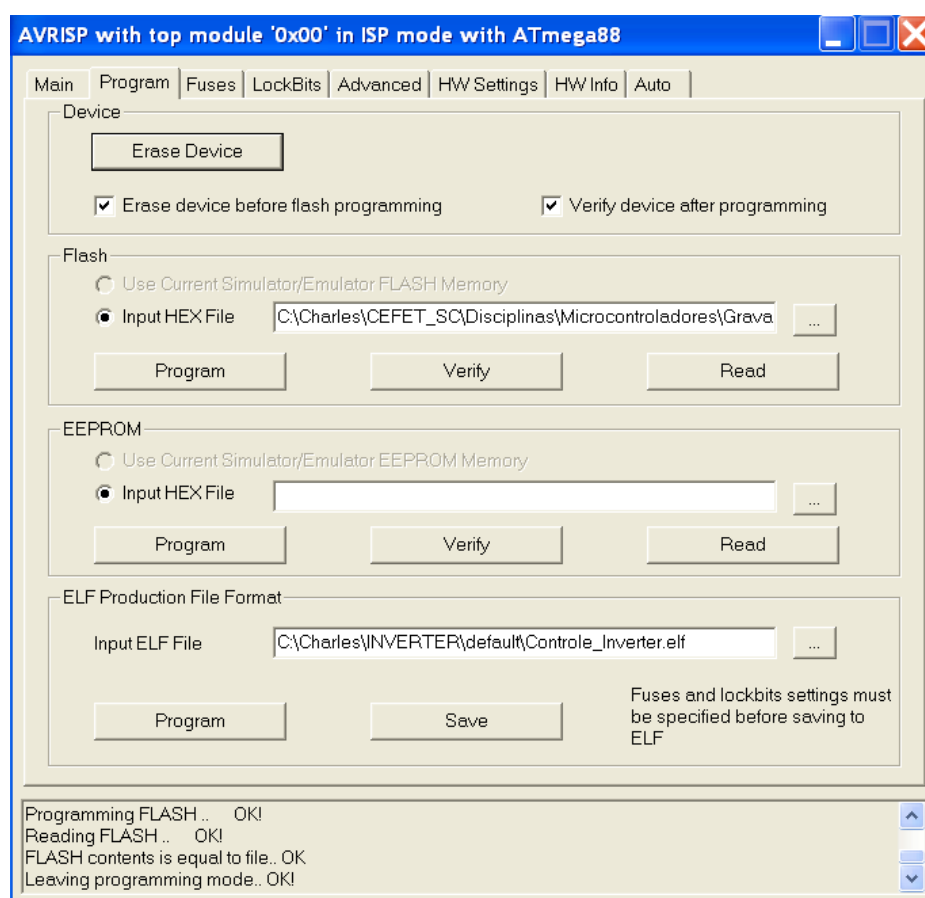


Fig. 18.6 – Janela para a gravação do  $\mu$ controlador.

A gravação ocorrerá inúmeras vezes *in-circuit* até que o hardware esteja funcionando adequadamente, conforme especificação do projeto. Após a gravação, raramente o programa gravado não necessitará alterações para satisfazer as características desejadas de trabalho. Logo, a ferramenta de gravação será muito empregada até se chegar à versão final do programa.

## 19. CONCLUSÕES

Um resumo sobre o ATmega8 foi apresentado. Os inúmeros detalhes faltantes se encontram no catálogo do fabricante, principal fonte para esta obra. Sua consulta é imprescindível.

A economia de energia no  $\mu$ controlador é fundamental quando se empregam baterias de alimentação. Para se aprofundar e entender as possibilidades que o ATmega possui, o catálogo do fabricante deve ser consultado.

Os projetos propostos apenas ilustram algumas possibilidades de emprego do ATmega. Circuitos reais precisam agregar mais componentes e detalhes que não são necessários na simulação. Circuitos complexos estão longe do que aqui foi posto.

Importante lembrar que um bom projeto implica em um código eficiente e no emprego do menor número de componentes possível, mantendo a qualidade do resultado desejado. Um bom projetista e programador só é feito com anos de experiência. Além disso, de nada adianta ter um bom programa e circuito se o projeto da placa de circuito impresso não for bem feito.

Dada a portabilidade do código C, bem como a necessidade de não reescrever códigos já desenvolvidos, é altamente recomendado que sejam criadas bibliotecas próprias de funções, como, por exemplo, para o uso do LCD. No estudo de rotinas prontas, é fundamental a completa compreensão dessas, caso contrário, a habilidade de programação não se desenvolve.

Na área de  $\mu$ controladores atualizar-se é questão de sobrevivência. Não se deve ficar preso a uma única tecnologia. O mercado muda constantemente, as aplicações sem fio são cada vez mais comuns e que a interface USB está em voga.

## 20. BIBLIOGRAFIA

**ATMEGA8(L):** Manual do fabricante (doc2486.pdf).

**ITM-12864K0YTBL:** Manual do fabricante - *Intech LCD Group*.

SCHILDT, Herbert. **C Completo e Total**. Makron Books, 3º ed., 1997.

SOUZA, David José, e Nicolas César Lavinia. **Conectando o PIC - Recursos Avançados**. Érica, 4º ed., 2003.

PEREIRA, Fábio. **Microcontroladores PIC – Programação em C**. Érica, 7º ed., 2003.

PEREIRA, Fábio. **Microcontroladores MSP430 - Teoria e Prática**. Érica, 1º ed., 2005.

PEREIRA, Fábio. **Tecnologia ARM - Microcontroladores de 32 Bits**. Érica, 1º ed., 2007.

TOCCI, Ronald J., Neal S. Widmer e Gregory L. Moss. **Sistemas Digitais – Princípios e Aplicações**. Prentice Hall do Brasil, 10º ed., 2008.

VILLAÇA, Marcos V. M. **Introdução aos Microcontroladores** – Apostila. Instituto Federal de Santa Catarina, 2º ed., Departamento de Eletrônica, 2007.

### Sítios:

[www.atmel.com](http://www.atmel.com)

*Application Notes:*

- *Software Universal Serial Bus (USB)*
- *Dallas 1-Wire® master*
- *Efficient C Coding for AVR*
- *AVR Hardware Design Considerations*
- *In-System Programming*
- *Using the TWI module as I2C master*
- *Using the TWI module as I2C slave*

[www.microchip.com](http://www.microchip.com)

*Application Note:*

- *Microchip Tips 'n Tricks*

[www.maxim-ic.com](http://www.maxim-ic.com)

[www.nxp.com](http://www.nxp.com) (philips\_i2c\_logic\_overview.pdf)

[www.techtoys.com.hk](http://www.techtoys.com.hk) (chp8\_JHD12864LCD.pdf)

## ANEXOS

### 1. BOOTLOADER

Agora que os microcontroladores disponíveis comercialmente possuem uma memória de programa razoável, surgiu um novo conceito: o boot loader.

O boot loader é um programa muito pequeno que é programado na parte baixa ou alta da memória do microcontrolador instalado no circuito em desenvolvimento. Este programa consegue comunicar com as ferramentas de desenvolvimento (utilizadas para escrever o código fonte) através de uma porta série ou um outro tipo de ligação série, como por exemplo um barramento USB, I<sup>2</sup>C, ou CAN.

O programa boot loader pode interpretar um determinado número de comandos relacionados com a leitura, programação e apagamento da memória de programa do microcontrolador associado. O princípio de funcionamento é muito simples.

Quando o programa que está sendo desenvolvido está pronto para testar, a comunicação com o boot loader é iniciada. O boot loader carrega então o código na memória de programa do microcontrolador, evitando assim a necessidade de um programador externo.

Depois de completada esta operação, o boot loader transfere o controle para o programa carregado, e o utilizador pode então executar e testar o seu programa. Um novo programa pode ser carregado com o boot loader sempre que for necessário (após o programa anterior ter sido apagado).

O ganho obtido usando este método tem um fator de dez em relação ao método tradicional de programação, o que naturalmente tem um efeito positivo em todo o processo de desenvolvimento.

Provavelmente já se percebeu que o microcontrolador tem que obedecer a um certo número de requisitos para que se possa usar um boot loader. Em particular, tem que possuir:

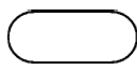
- memória de programa suficiente para armazenar o boot loader e o programa em desenvolvimento,
- ser capaz de apagar e programar internamente a memória de programa, e
- uma porta RS232 ou outro tipo de ligação série, como por exemplo USB ou CAN.

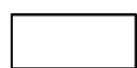
Se estes requisitos forem satisfeitos é muito fácil utilizar um boot loader. Existem muitos programas freeware disponíveis na Internet para famílias de microcontroladores que suportam carregamento através de boot loader.

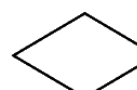
Revista Elektor, Fev/2006.

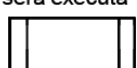
### 2. SÍMBOLOS EMPREGADOS EM FLUXOGRAMAS


Alguns símbolos utilizados em fluxogramas são apresentados a seguir.


 **TERMINAL**  
Ponto de início, término ou interrupção de um programa.


 **PROCESSAMENTO**  
Um grupo de instruções que executam uma função de processamento do programa.


 **DECISÃO**  
Indica a possibilidade de desvios para diversos pontos do programa

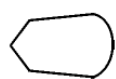
**PROCESSO PRÉ-DEFINIDO**  
Indica uma rotina que será executada fora do programa principal (sub-rotinas) 

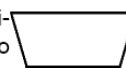
 **CONECTOR FORA DE PÁGINA**  
Uma entrada ou saída de uma página para outra

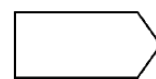
**FLUXO**  
Indica a direção do fluxo de dados ou de um processamento 

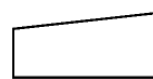
 **CONEXÃO**  
Indica a rota de prosseguimento do fluxograma

**SUB-ROTINA**  
Um grupo de operações separadas do fluxo do programa 

 **VISOR**  
Saída de informações através de vídeo ou display

**ENTRADA / SAÍDA**  
Qualquer função relacionada com dispositivo de entrada ou saída genéricos 

 **MODIFICAÇÃO DE PROGRAMA**  
Qualquer função que altera o próprio programa

**TECLADO**  
Entrada de informações através de teclado. 

Revista Saber Eletrônica, Jan/ 2001.

Mnemonics	Operands	Description	Operation	Flags	#Clocks
<b>ARITHMETIC AND LOGIC INSTRUCTIONS</b>					
ADD	Rd, Rr	Add two Registers	$Rd \leftarrow Rd + Rr$	Z, C, N, V, H	1
ADC	Rd, Rr	Add with Carry two Registers	$Rd \leftarrow Rd + Rr + C$	Z, C, N, V, H	1
ADIW	Rd, K	Add Immediate to Word	$Rdh:Rdl \leftarrow Rdh:Rdl + K$	Z, C, N, V, S	2
SUB	Rd, Rr	Subtract two Registers	$Rd \leftarrow Rd - Rr$	Z, C, N, V, H	1
SUBI	Rd, K	Subtract Constant from Register	$Rd \leftarrow Rd - K$	Z, C, N, V, H	1
SBC	Rd, Rr	Subtract with Carry two Registers	$Rd \leftarrow Rd - Rr - C$	Z, C, N, V, H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	$Rd \leftarrow Rd - K - C$	Z, C, N, V, H	1
SBIW	Rd, K	Subtract Immediate from Word	$Rdh:Rdl \leftarrow Rdh:Rdl - K$	Z, C, N, V, S	2
AND	Rd, Rr	Logical AND Registers	$Rd \leftarrow Rd \cdot Rr$	Z, N, V	1
ANDI	Rd, K	Logical AND Register and Constant	$Rd \leftarrow Rd \cdot K$	Z, N, V	1
OR	Rd, Rr	Logical OR Registers	$Rd \leftarrow Rd \vee Rr$	Z, N, V	1
ORI	Rd, K	Logical OR Register and Constant	$Rd \leftarrow Rd \vee K$	Z, N, V	1
EOR	Rd, Rr	Exclusive OR Registers	$Rd \leftarrow Rd \oplus Rr$	Z, N, V	1
COM	Rd	One's Complement	$Rd \leftarrow 0xFF - Rd$	Z, C, N, V	1
NEG	Rd	Two's Complement	$Rd \leftarrow 0x00 - Rd$	Z, C, N, V, H	1
SBR	Rd, K	Set Bit(s) in Register	$Rd \leftarrow Rd \vee K$	Z, N, V	1
CBR	Rd, K	Clear Bit(s) in Register	$Rd \leftarrow Rd \cdot (0xFF - K)$	Z, N, V	1
INC	Rd	Increment	$Rd \leftarrow Rd + 1$	Z, N, V	1
DEC	Rd	Decrement	$Rd \leftarrow Rd - 1$	Z, N, V	1
TST	Rd	Test for Zero or Minus	$Rd \leftarrow Rd \cdot Rd$	Z, N, V	1
CLR	Rd	Clear Register	$Rd \leftarrow Rd \oplus Rd$	Z, N, V	1
SER	Rd	Set Register	$Rd \leftarrow 0xFF$	None	1
MUL	Rd, Rr	Multiply Unsigned	$R1:R0 \leftarrow Rd \times Rr$	Z, C	2
MULS	Rd, Rr	Multiply Signed	$R1:R0 \leftarrow Rd \times Rr$	Z, C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	$R1:R0 \leftarrow Rd \times Rr$	Z, C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	$R1:R0 \leftarrow (Rd \times Rr) << 1$	Z, C	2
FMULS	Rd, Rr	Fractional Multiply Signed	$R1:R0 \leftarrow (Rd \times Rr) << 1$	Z, C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	$R1:R0 \leftarrow (Rd \times Rr) << 1$	Z, C	2
<b>BRANCH INSTRUCTIONS</b>					
RJMP	k	Relative Jump	$PC \leftarrow PC + k + 1$	None	2
IJMP		Indirect Jump to (Z)	$PC \leftarrow Z$	None	2
RCALL	k	Relative Subroutine Call	$PC \leftarrow PC + k + 1$	None	3
ICALL		Indirect Call to (Z)	$PC \leftarrow Z$	None	3
RET		Subroutine Return	$PC \leftarrow STACK$	None	4
RETI		Interrupt Return	$PC \leftarrow STACK$	I	4
CPSE	Rd, Rr	Compare, Skip if Equal	if $(Rd = Rr)$ $PC \leftarrow PC + 2$ or 3	None	1 / 2 / 3
CP	Rd, Rr	Compare	$Rd - Rr$	Z, N, V, C, H	1
CPC	Rd, Rr	Compare with Carry	$Rd - Rr - C$	Z, N, V, C, H	1
CPI	Rd, K	Compare Register with Immediate	$Rd - K$	Z, N, V, C, H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if $(Rr(b)=0)$ $PC \leftarrow PC + 2$ or 3	None	1 / 2 / 3
SBRs	Rr, b	Skip if Bit in Register is Set	if $(Rr(b)=1)$ $PC \leftarrow PC + 2$ or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if $(P(b)=0)$ $PC \leftarrow PC + 2$ or 3	None	1 / 2 / 3
SBSI	P, b	Skip if Bit in I/O Register is Set	if $(P(b)=1)$ $PC \leftarrow PC + 2$ or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if $(SREG(s) = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if $(SREG(s) = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BREQ	k	Branch if Equal	if $(Z = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRNE	k	Branch if Not Equal	if $(Z = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRCS	k	Branch if Carry Set	if $(C = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRCC	k	Branch if Carry Cleared	if $(C = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRSH	k	Branch if Same or Higher	if $(C = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRLO	k	Branch if Lower	if $(C = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRMI	k	Branch if Minus	if $(N = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRPL	k	Branch if Plus	if $(N = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if $(N \oplus V = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRLT	k	Branch if Less Than Zero, Signed	if $(N \oplus V = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if $(H = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if $(H = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRTS	k	Branch if T Flag Set	if $(T = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if $(T = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if $(V = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if $(V = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRIE	k	Branch if Interrupt Enabled	if $(I = 1)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if $(I = 0)$ then $PC \leftarrow PC + k + 1$	None	1 / 2
<b>DATA TRANSFER INSTRUCTIONS</b>					
MOV	Rd, Rr	Move Between Registers	$Rd \leftarrow Rr$	None	1
MOVW	Rd, Rr	Copy Register Word	$Rd+1:Rd \leftarrow Rr+1:Rr$	None	1
LDI	Rd, K	Load Immediate	$Rd \leftarrow K$	None	1
LD	Rd, X	Load Indirect	$Rd \leftarrow (X)$	None	2
LD	Rd, X+	Load Indirect and Post-Inc.	$Rd \leftarrow (X), X \leftarrow X + 1$	None	2
LD	Rd, -X	Load Indirect and Pre-Dec.	$X \leftarrow X - 1, Rd \leftarrow (X)$	None	2
LD	Rd, Y	Load Indirect	$Rd \leftarrow (Y)$	None	2
LD	Rd, Y+	Load Indirect and Post-Inc.	$Rd \leftarrow (Y), Y \leftarrow Y + 1$	None	2
LD	Rd, -Y	Load Indirect and Pre-Dec.	$Y \leftarrow Y - 1, Rd \leftarrow (Y)$	None	2
LDD	Rd, Y+q	Load Indirect with Displacement	$Rd \leftarrow (Y + q)$	None	2
LD	Rd, Z	Load Indirect	$Rd \leftarrow (Z)$	None	2
LD	Rd, Z+	Load Indirect and Post-Inc.	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	None	2
LD	Rd, -Z	Load Indirect and Pre-Dec.	$Z \leftarrow Z - 1, Rd \leftarrow (Z)$	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	$Rd \leftarrow (Z + q)$	None	2
LDS	Rd, k	Load Direct from SRAM	$Rd \leftarrow (k)$	None	2
ST	X, Rr	Store Indirect	$(X) \leftarrow Rr$	None	2
ST	X+, Rr	Store Indirect and Post-Inc.	$(X) \leftarrow Rr, X \leftarrow X + 1$	None	2
ST	-X, Rr	Store Indirect and Pre-Dec.	$X \leftarrow X - 1, (X) \leftarrow Rr$	None	2
ST	Y, Rr	Store Indirect	$(Y) \leftarrow Rr$	None	2
ST	Y+, Rr	Store Indirect and Post-Inc.	$(Y) \leftarrow Rr, Y \leftarrow Y + 1$	None	2
ST	-Y, Rr	Store Indirect and Pre-Dec.	$Y \leftarrow Y - 1, (Y) \leftarrow Rr$	None	2



Mnemonics	Operands	Description	Operation	Flags	#Clocks
<b>DATA TRANSFER INSTRUCTIONS</b>					
STD	Y+q,Rr	Store Indirect with Displacement	$(Y + q) \leftarrow Rr$	None	2
ST	Z, Rr	Store Indirect	$(Z) \leftarrow Rr$	None	2
ST	Z+, Rr	Store Indirect and Post-Inc.	$(Z) \leftarrow Rr, Z \leftarrow Z + 1$	None	2
ST	-Z, Rr	Store Indirect and Pre-Dec.	$Z \leftarrow Z - 1, (Z) \leftarrow Rr$	None	2
STD	Z+q,Rr	Store Indirect with Displacement	$(Z + q) \leftarrow Rr$	None	2
STS	k, Rr	Store Direct to SRAM	$(k) \leftarrow Rr$	None	2
LPM		Load Program Memory	$R0 \leftarrow (Z)$	None	3
LPM	Rd, Z	Load Program Memory	$Rd \leftarrow (Z)$	None	3
LPM	Rd, Z+	Load Program Memory and Post-Inc	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	None	3
SPM		Store Program Memory	$(Z) \leftarrow R1:R0$	None	-
IN	Rd, P	In Port	$Rd \leftarrow P$	None	1
OUT	P, Rr	Out Port	$P \leftarrow Rr$	None	1
PUSH	Rr	Push Register on Stack	$STACK \leftarrow Rr$	None	2
POP	Rd	Pop Register from Stack	$Rd \leftarrow STACK$	None	2
<b>BIT AND BIT-TEST INSTRUCTIONS</b>					
SBI	P,b	Set Bit in I/O Register	$I/O(P,b) \leftarrow 1$	None	2
CBI	P,b	Clear Bit in I/O Register	$I/O(P,b) \leftarrow 0$	None	2
LSL	Rd	Logical Shift Left	$Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0$	Z,C,N,V	1
LSR	Rd	Logical Shift Right	$Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0$	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	$Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	$Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	$Rd(n) \leftarrow Rd(n+1), n=0..6$	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	$Rd(3..0) \leftarrow Rd(7..4), Rd(7..4) \leftarrow Rd(3..0)$	None	1
BSET	s	Flag Set	$SREG(s) \leftarrow 1$	SREG(s)	1
BCLR	s	Flag Clear	$SREG(s) \leftarrow 0$	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	$T \leftarrow Rr(b)$	T	1
BLD	Rd, b	Bit load from T to Register	$Rd(b) \leftarrow T$	None	1
SEC		Set Carry	$C \leftarrow 1$	C	1
CLC		Clear Carry	$C \leftarrow 0$	C	1
SEN		Set Negative Flag	$N \leftarrow 1$	N	1
CLN		Clear Negative Flag	$N \leftarrow 0$	N	1
SEZ		Set Zero Flag	$Z \leftarrow 1$	Z	1
CLZ		Clear Zero Flag	$Z \leftarrow 0$	Z	1
SEI		Global Interrupt Enable	$I \leftarrow 1$	I	1
CLI		Global Interrupt Disable	$I \leftarrow 0$	I	1
SES		Set Signed Test Flag	$S \leftarrow 1$	S	1
CLS		Clear Signed Test Flag	$S \leftarrow 0$	S	1
SEV		Set Twos Complement Overflow	$V \leftarrow 1$	V	1
CLV		Clear Twos Complement Overflow	$V \leftarrow 0$	V	1
SET		Set T in SREG	$T \leftarrow 1$	T	1
CLT		Clear T in SREG	$T \leftarrow 0$	T	1
SEH		Set Half Carry Flag in SREG	$H \leftarrow 1$	H	1
CLH		Clear Half Carry Flag in SREG	$H \leftarrow 0$	H	1
<b>MCU CONTROL INSTRUCTIONS</b>					
NOP		No Operation		None	1
SLEEP		Sleep	(see specific descr. for Sleep function)	None	1
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None	1

## Status Register

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

### BIT 7 – I: Global Interrupt Enable

Este bit é a chave geral para habilitar as interrupções. Cada interrupção individual possui seus registradores de controle. O bit I é limpo quando uma interrupção ocorre (impedindo que outras ocorram simultaneamente) e volta a ser setado quando se termina o tratamento da interrupção (instrução RETI).

### BIT 6 – T: Bit Copy Storage

Serve para copiar o valor de um bit de um registrador ou escrever o valor de um bit em um registrador (instruções BLD e BST).

### BIT 5 – H: Half Carry Flag

Indica quando um meio *Carry* (em um *nibble*) ocorreu em alguma operação aritmética. Útil em aritmética BCD.

### BIT 4 – S: Sign Bit, $S = N \oplus V$

O bit S é o resultado de um ou exclusivo entre o *flag* negativo N e o *flag* de estouro do complemento de dois V.

### BIT 3 – V: Two's Complement Overflow Flag

O *flag* de estouro do complemento de dois ajuda na aritmética com complemento de dois.

### BIT 2 – N: Negative Flag

O *flag* negativo indica quando uma operação aritmética ou lógica resulta em um valor negativo.

### BIT 1 – Z: Zero Flag

O *flag* zero indica quando uma operação aritmética ou lógica resulta em zero.

### BIT 0 – C: Carry Flag

O *flag* de *Carry* indica quando houve *Carry* numa operação matemática ou lógica.

#### 4. PRODUZINDO UM CÓDIGO C EFICIENTE

##### Reduzir o tamanho do Código compilado

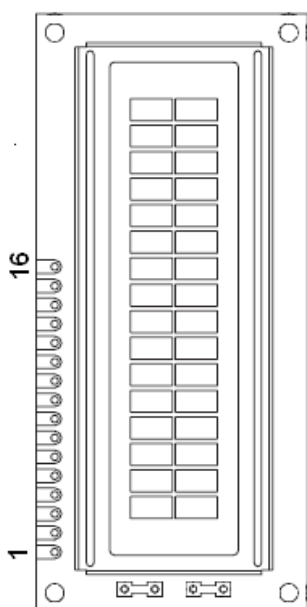
1. Compile com a máxima otimização possível.
2. Use variáveis locais sempre que possível.
3. Use o menor tipo de dado possível, *unsigned* se aplicável.
4. Se uma variável não local é referenciada somente dentro de uma função, ela deve ser declarada como *static*.
5. Junte variáveis não-locais em estruturas, se conveniente. Isto aumenta a possibilidade de endereçamento indireto sem recarga de ponteiro.
6. Para acessar memória mapeada, use ponteiros.
7. Use `for(;;){ }` para laços infinitos.
8. Use `do{ } while(expressão)` se aplicável.
9. Use laços com contadores decrescentes e pré-decrementados, se possível.
10. Acesse a memória de I/O diretamente (não use ponteiros).
11. Use *macros* ao invés de funções para tarefas que produzem menos que 2-3 linhas de código em *assembly*.
12. Evite usar funções dentro de interrupções.

##### Reduzindo o tamanho da memória RAM necessária

1. Todas as constantes e literais devem ser colocados na memória flash.
2. Evite usar variáveis globais. Empregue variáveis locais sempre que possível.

#### 5. DISPLAY DE CRISTAL LÍQUIDO 16x2 - CONTROLADOR HD44780

##### PINAGEM



16	LED-
15	LED+
14	DB7
13	DB6
12	DB5
11	DB4
10	DB3
9	DB2
8	DB1
7	DB0
6	E
5	R/W
4	RS
3	VEE
2	VCC
1	VSS

Pino	Função	Descrição
1	Alimentação	Terra ou GND
2	Alimentação	VCC ou +5V
3	V0	<b>Tensão para ajuste de contraste</b>
4	RS Seleção:	1 - Dado, 0 - Instrução
5	R/W Seleção:	1 - Leitura, 0 - Escrita
6	E Chip select	1 ou (1 → 0) - Habilita, 0 - Desabilitado
7	B0 LSB	Barramento de Dados
8	B1	
9	B2	
10	B3	
11	B4	
12	B5	
13	B6	
14	B7 MSB	
15	A (qdo existir)	Anodo p/ LED backlight
16	K (qdo existir)	Catodo p/ LED backlight

## CÓDIGOS DE INSTRUÇÕES

DESCRIÇÃO	MODO	Código h
Display	Liga (sem cursor)	0C
	Desliga	0A / 08
Limpa Display com Home cursor		01
Controle do Cursor	Liga	0E
	Desliga	0C
	Desloca para Esquerda	10
	Desloca para Direita	14
	Cursor Home	02
	Cursor Piscante	0D
	Cursor com Alternância	0F
Sentido de deslocamento do cursor ao entrar com caracter	Para a esquerda	04
	Para a direita	06
Deslocamento da mensagem ao entrar com caracter	Para a esquerda	07
	Para a direita	05
Deslocamento da mensagem sem entrada de caracter	Para a esquerda	18
	Para a direita	1C
End. da primeira posição	primeira linha	80
	segunda linha	C0

## DETALHAMENTO DAS INSTRUÇÕES

INSTRUÇÃO	R S	R/ W	B7	B6	B5	B4	B3	B2	B1	B0	DESCRIÇÃO e tempo de execução (uS)	t
Limpa Display	0	0	0	0	0	0	0	0	0	1	-Limpa todo o display e retorna o cursor para a primeira posição da primeira linha	1.6 mS
Home p/ Cursor	0	0	0	0	0	0	0	0	1	*	-Retorna o cursor para a 1. coluna da 1. Linha -Retorna a mensagem previamente deslocada a sua posição original	1.6 mS
Fixa o modo de funcionamento	0	0	0	0	0	0	0	1	X	S	-Estabelece o sentido de deslocamento do cursor (X=0 p/ esquerda, X=1 p/ direita) -Estabelece se a mensagem deve ou não ser deslocada com a entrada de um novo caracter (S=1 SIM, X=1 p/ direita) -Esta instrução tem efeito somente durante a leitura e escrita de dados.	40 uS
Controle do Display	0	0	0	0	0	0	1	D	C	B	-Liga (D=1) ou desliga display (D=0) -Liga(C=1) ou desliga cursor (C=0) -Cursor Piscante(B=1) se C=1	40 uS
Desloca cursor ou mensagem	0	0	0	0	0	1	C	R	*	*	-Desloca o cursor (C=0) ou a mensagem (C=1) para a Direita se (R=1) ou esquerda se (R=0) - Desloca sem alterar o conteúdo da DDRAM	40 uS
Fixa o modo de utilização do módulo LCD	0	0	0	0	1	Y	N	F	*	*	-Comunicação do módulo com 8 bits(Y=1) ou 4 bits(Y=0) -Número de linhas: 1 (N=0) e 2 ou mais (N=1) -Matriz do caracter: 5x7(F=0) ou 5x10(F=1) - Esta instrução deve ser ativada durante a inicialização	40 uS
Posiciona no endereço da CGRAM	0	0	0	1	Endereço da CGRAM						-Fixa o endereço na CGRAM para posteriormente enviar ou ler o dado (byte)	40 uS
Posiciona no endereço da DDRAM	0	0	1	Endereço da DDRAM							-Fixa o endereço na DDRAM para posteriormente enviar ou ler o dado (byte)	40 uS
Leitura do Flag Busy	0	1	B F	AC							-Lê o conteúdo do contador de endereços (AC) e o BF. O BF (bit 7) indica se a última operação foi concluída (BF=0 concluída) ou está em execução (BF=1).	0
Escreve dado na CGRAM / DDRAM	0	1	Dado a ser gravado no LCD								- Grava o byte presente nos pinos de dados no local apontado pelo contador de endereços (posição do cursor)	40 uS
Lê Dado na CGRAM / DDRAM	1	1	Dado lido do módulo								- Lê o byte no local apontado pelo contador de endereços (posição do cursor)	40 uS

## ENDEREÇO DOS SEGMENTOS

LCD 16x2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
linha 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
linha 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF

## CONJUNTO E CÓDIGO DOS CARACTERES

	HIGH	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
LOW		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
x0	xxxx 0000	CG RAM (1)			00P0P									—	△	△	△
x1	xxxx 0001	CG RAM (2)			!1A0a9									7	7	4	ä
x2	xxxx 0010	CG RAM (3)			"2BRbr									「	イ	ツ	×
x3	xxxx 0011	CG RAM (4)			#3C5cs									」	ウ	テ	ε
x4	xxxx 0100	CG RAM (5)			\$4DTdt									、	イ	ト	μ
x5	xxxx 0101	CG RAM (6)			%5EUeu									・	オ	ナ	1
x6	xxxx 0110	CG RAM (7)			&6FUfu									ヲ	カ	ニ	ヨ
x7	xxxx 0111	CG RAM (8)			'7GWgw									フ	キ	ヲ	ウ
x8	xxxx 1000	(1)			(8HXhx									イ	ウ	ナ	ル
x9	xxxx 1001	(2)			)9IYiy									オ	ウ	ル	ナ
xA	xxxx 1010	(3)			*:JZjz									エ	コ	ナ	ル
xB	xxxx 1011	(4)			+;KCK<									★	ウ	ロ	ナ
xC	xxxx 1100	(5)			,<L¥ll									ナ	ウ	フ	ナ
xD	xxxx 1101	(6)			-=MIm>									ユ	ズ	ナ	ル
xE	xxxx 1110	(7)			.>N^n→									ヨ	セ	ナ	ル
xF	xxxx 1111	(8)			/?O_o€									ウ	ナ	ル	ナ

## 6. ERROS DA USART NO ATMEGA8

$$R_{slow} = \frac{(D+1)S}{S-1+D \cdot S+S_F}$$

$$R_{fast} = \frac{(D+2)S}{(D+1)S+S_M}$$

- D Sum of character size and parity size (D = 5- to 10-bit)
- S Samples per bit. S = 16 for Normal Speed mode and S = 8 for Double Speed mode.
- S<sub>F</sub> First sample number used for majority voting. S<sub>F</sub> = 8 for Normal Speed and S<sub>F</sub> = 4 for Double Speed mode.
- S<sub>M</sub> Middle sample number used for majority voting. S<sub>M</sub> = 9 for Normal Speed and S<sub>M</sub> = 5 for Double Speed mode.
- R<sub>slow</sub> is the ratio of the slowest incoming data rate that can be accepted in relation to the Receiver baud rate. R<sub>fast</sub> is the ratio of the fastest incoming data rate that can be accepted in relation to the Receiver baud rate.

## ERRO MÁXIMO RECOMENDADO

U2X = 0

D# (Data+Parity Bit)	R <sub>slow</sub> (%)	R <sub>fast</sub> (%)	Max Total Error (%)	Recommended Max Receiver Error (%)
5	93,20	106,67	+6.67/-6.8	± 3.0
6	94,12	105,79	+5.79/-5.88	± 2.0
7	94,81	105,11	+5.11/-5.19	± 2.0
8	95,36	104,58	+4.58/-4.54	± 2.0
9	95,81	104,14	+4.14/-4.19	± 1.5
10	96,17	103,78	+3.78/-3.83	± 1.5

U2X = 1

D# (Data+Parity Bit)	R <sub>slow</sub> (%)	R <sub>fast</sub> (%)	Max Total Error (%)	Recommended Max Receiver Error (%)
5	94,12	105,66	+5.66/-5.88	± 2.5
6	94,92	104,92	+4.92/-5.08	± 2.0
7	95,52	104,35	+4.35/-4.48	± 1.5
8	96,00	103,90	+3.90/-4.00	± 1.5
9	96,39	103,53	+3.53/-3.61	± 1.5
10	96,70	103,23	+3.23/-3.30	± 1.0

## ERROS DEVIDO À FREQUÊNCIA DE OPERAÇÃO

Baud Rate (bps)	f <sub>osc</sub> = 1.0000 MHz				f <sub>osc</sub> = 1.8432 MHz				f <sub>osc</sub> = 2.0000 MHz			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	25	0.2%	51	0.2%	47	0.0%	95	0.0%	51	0.2%	103	0.2%
4800	12	0.2%	25	0.2%	23	0.0%	47	0.0%	25	0.2%	51	0.2%
9600	6	-7.0%	12	0.2%	11	0.0%	23	0.0%	12	0.2%	25	0.2%
14.4k	3	8.5%	8	-3.5%	7	0.0%	15	0.0%	8	-3.5%	16	2.1%
19.2k	2	8.5%	6	-7.0%	5	0.0%	11	0.0%	6	-7.0%	12	0.2%
28.8k	1	8.5%	3	8.5%	3	0.0%	7	0.0%	3	8.5%	8	-3.5%
38.4k	1	-18.6%	2	8.5%	2	0.0%	5	0.0%	2	8.5%	6	-7.0%
57.6k	0	8.5%	1	8.5%	1	0.0%	3	0.0%	1	8.5%	3	8.5%
76.8k	–	–	1	-18.6%	1	-25.0%	2	0.0%	1	-18.6%	2	8.5%
115.2k	–	–	0	8.5%	0	0.0%	1	0.0%	0	8.5%	1	8.5%
230.4k	–	–	–	–	–	–	0	0.0%	–	–	–	–
250k	–	–	–	–	–	–	–	–	–	–	0	0.0%
Max <sup>(1)</sup>	62.5 kbps		125 kbps		115.2 kbps		230.4 kbps		125 kbps		250 kbps	

1. UBRR = 0, Error = 0.0%

Baud Rate (bps)	f <sub>osc</sub> = 3.6864 MHz				f <sub>osc</sub> = 4.0000 MHz				f <sub>osc</sub> = 7.3728 MHz			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	95	0.0%	191	0.0%	103	0.2%	207	0.2%	191	0.0%	383	0.0%
4800	47	0.0%	95	0.0%	51	0.2%	103	0.2%	95	0.0%	191	0.0%
9600	23	0.0%	47	0.0%	25	0.2%	51	0.2%	47	0.0%	95	0.0%
14.4k	15	0.0%	31	0.0%	16	2.1%	34	-0.8%	31	0.0%	63	0.0%
19.2k	11	0.0%	23	0.0%	12	0.2%	25	0.2%	23	0.0%	47	0.0%
28.8k	7	0.0%	15	0.0%	8	-3.5%	16	2.1%	15	0.0%	31	0.0%
38.4k	5	0.0%	11	0.0%	6	-7.0%	12	0.2%	11	0.0%	23	0.0%
57.6k	3	0.0%	7	0.0%	3	8.5%	8	-3.5%	7	0.0%	15	0.0%
76.8k	2	0.0%	5	0.0%	2	8.5%	6	-7.0%	5	0.0%	11	0.0%
115.2k	1	0.0%	3	0.0%	1	8.5%	3	8.5%	3	0.0%	7	0.0%
230.4k	0	0.0%	1	0.0%	0	8.5%	1	8.5%	1	0.0%	3	0.0%
250k	0	-7.8%	1	-7.8%	0	0.0%	1	0.0%	1	-7.8%	3	-7.8%
0.5M	–	–	0	-7.8%	–	–	0	0.0%	0	-7.8%	1	-7.8%
1M	–	–	–	–	–	–	–	–	–	–	0	-7.8%
Max <sup>(1)</sup>	230.4 kbps		460.8 kbps		250 kbps		0.5 Mbps		460.8 kbps		921.6 kbps	

1. UBRR = 0, Error = 0.0%

Baud Rate (bps)	f <sub>osc</sub> = 8.0000 MHz				f <sub>osc</sub> = 11.0592 MHz				f <sub>osc</sub> = 14.7456 MHz			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	207	0.2%	416	-0.1%	287	0.0%	575	0.0%	383	0.0%	767	0.0%
4800	103	0.2%	207	0.2%	143	0.0%	287	0.0%	191	0.0%	383	0.0%
9600	51	0.2%	103	0.2%	71	0.0%	143	0.0%	95	0.0%	191	0.0%
14.4k	34	-0.8%	68	0.6%	47	0.0%	95	0.0%	63	0.0%	127	0.0%
19.2k	25	0.2%	51	0.2%	35	0.0%	71	0.0%	47	0.0%	95	0.0%
28.8k	16	2.1%	34	-0.8%	23	0.0%	47	0.0%	31	0.0%	63	0.0%
38.4k	12	0.2%	25	0.2%	17	0.0%	35	0.0%	23	0.0%	47	0.0%
57.6k	8	-3.5%	16	2.1%	11	0.0%	23	0.0%	15	0.0%	31	0.0%
76.8k	6	-7.0%	12	0.2%	8	0.0%	17	0.0%	11	0.0%	23	0.0%
115.2k	3	8.5%	8	-3.5%	5	0.0%	11	0.0%	7	0.0%	15	0.0%
230.4k	1	8.5%	3	8.5%	2	0.0%	5	0.0%	3	0.0%	7	0.0%
250k	1	0.0%	3	0.0%	2	-7.8%	5	-7.8%	3	-7.8%	6	5.3%
0.5M	0	0.0%	1	0.0%	—	—	2	-7.8%	1	-7.8%	3	-7.8%
1M	—	—	0	0.0%	—	—	—	—	0	-7.8%	1	-7.8%
Max <sup>(1)</sup>	0.5 Mbps		1 Mbps		691.2 kbps		1.3824 Mbps		921.6 kbps		1.8432 Mbps	

1. UBRR = 0, Error = 0.0%

Baud Rate (bps)	f <sub>osc</sub> = 16.0000 MHz				f <sub>osc</sub> = 18.4320 MHz				f <sub>osc</sub> = 20.0000 MHz			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%	479	0.0%	959	0.0%	520	0.0%	1041	0.0%
4800	207	0.2%	416	-0.1%	239	0.0%	479	0.0%	259	0.2%	520	0.0%
9600	103	0.2%	207	0.2%	119	0.0%	239	0.0%	129	0.2%	259	0.2%
14.4k	68	0.6%	138	-0.1%	79	0.0%	159	0.0%	86	-0.2%	173	-0.2%
19.2k	51	0.2%	103	0.2%	59	0.0%	119	0.0%	64	0.2%	129	0.2%
28.8k	34	-0.8%	68	0.6%	39	0.0%	79	0.0%	42	0.9%	86	-0.2%
38.4k	25	0.2%	51	0.2%	29	0.0%	59	0.0%	32	-1.4%	64	0.2%
57.6k	16	2.1%	34	-0.8%	19	0.0%	39	0.0%	21	-1.4%	42	0.9%
76.8k	12	0.2%	25	0.2%	14	0.0%	29	0.0%	15	1.7%	32	-1.4%
115.2k	8	-3.5%	16	2.1%	9	0.0%	19	0.0%	10	-1.4%	21	-1.4%
230.4k	3	8.5%	8	-3.5%	4	0.0%	9	0.0%	4	8.5%	10	-1.4%
250k	3	0.0%	7	0.0%	4	-7.8%	8	2.4%	4	0.0%	9	0.0%
0.5M	1	0.0%	3	0.0%	—	—	4	-7.8%	—	—	4	0.0%
1M	0	0.0%	1	0.0%	—	—	—	—	—	—	—	—
Max <sup>(1)</sup>	1 Mbps		2 Mbps		1.152 Mbps		2.304 Mbps		1.25 Mbps		2.5 Mbps	

1. UBRR = 0, Error = 0.0%

## 7. TABELAS DE CONVERSÃO – CHAR – DEC – HEX - BIN

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(nul)	0	0x00	(sp)	32	0x20	@	64	0x40	`	96	0x60
(soh)	1	0x01	!	33	0x21	A	65	0x41	a	97	0x61
(stx)	2	0x02	"	34	0x22	B	66	0x42	b	98	0x62
(etx)	3	0x03	#	35	0x23	C	67	0x43	c	99	0x63
(eot)	4	0x04	\$	36	0x24	D	68	0x44	d	100	0x64
(enq)	5	0x05	%	37	0x25	E	69	0x45	e	101	0x65
(ack)	6	0x06	&	38	0x26	F	70	0x46	f	102	0x66
(bel)	7	0x07	'	39	0x27	G	71	0x47	g	103	0x67
(bs)	8	0x08	(	40	0x28	H	72	0x48	h	104	0x68
(ht)	9	0x09	)	41	0x29	I	73	0x49	i	105	0x69
(nl)	10	0x0a	*	42	0x2a	J	74	0x4a	j	106	0x6a
(vt)	11	0x0b	+	43	0x2b	K	75	0x4b	k	107	0x6b
(np)	12	0x0c	,	44	0x2c	L	76	0x4c	l	108	0x6c
(cr)	13	0x0d	-	45	0x2d	M	77	0x4d	m	109	0x6d
(so)	14	0x0e	.	46	0x2e	N	78	0x4e	n	110	0x6e
(si)	15	0x0f	/	47	0x2f	O	79	0x4f	o	111	0x6f
(dle)	16	0x10	0	48	0x30	P	80	0x50	p	112	0x70
(dc1)	17	0x11	1	49	0x31	Q	81	0x51	q	113	0x71
(dc2)	18	0x12	2	50	0x32	R	82	0x52	r	114	0x72
(dc3)	19	0x13	3	51	0x33	S	83	0x53	s	115	0x73
(dc4)	20	0x14	4	52	0x34	T	84	0x54	t	116	0x74
(nak)	21	0x15	5	53	0x35	U	85	0x55	u	117	0x75
(syn)	22	0x16	6	54	0x36	V	86	0x56	v	118	0x76
(etb)	23	0x17	7	55	0x37	W	87	0x57	w	119	0x77
(can)	24	0x18	8	56	0x38	X	88	0x58	x	120	0x78
(em)	25	0x19	9	57	0x39	Y	89	0x59	y	121	0x79
(sub)	26	0x1a	:	58	0x3a	Z	90	0x5a	z	122	0x7a
(esc)	27	0x1b	;	59	0x3b	[	91	0x5b	{	123	0x7b
(fs)	28	0x1c	<	60	0x3c	\	92	0x5c		124	0x7c
(rs)	29	0x1d	=	61	0x3d	]	93	0x5d	}	125	0x7d
(rs)	30	0x1e	>	62	0x3e	^	94	0x5e	~	126	0x7e
(us)	31	0x1f	?	63	0x3f	_	95	0x5f	(del)	127	0x7f

Dec	Hex	Bin	Dec	Hex	Bin	Dec	Hex	Bin	Dec	Hex	Bin
0	0	00000000	64	40	01000000	128	80	10000000	192	c0	11000000
1	1	00000001	65	41	01000001	129	81	10000001	193	c1	11000001
2	2	00000010	66	42	01000010	130	82	10000010	194	c2	11000010
3	3	00000011	67	43	01000011	131	83	10000011	195	c3	11000011
4	4	00000100	68	44	01000100	132	84	10000100	196	c4	11000100
5	5	00000101	69	45	01000101	133	85	10000101	197	c5	11000101
6	6	00000110	70	46	01000110	134	86	10000110	198	c6	11000110
7	7	00000111	71	47	01000111	135	87	10000111	199	c7	11000111
8	8	00001000	72	48	01001000	136	88	10001000	200	c8	11001000
9	9	00001001	73	49	01001001	137	89	10001001	201	c9	11001001
10	a	00001010	74	4a	01001010	138	8a	10001010	202	ca	11001010
11	b	00001011	75	4b	01001011	139	8b	10001011	203	cb	11001011
12	c	00001100	76	4c	01001100	140	8c	10001100	204	cc	11001100
13	d	00001101	77	4d	01001101	141	8d	10001101	205	cd	11001101
14	e	00001110	78	4e	01001110	142	8e	10001110	206	ce	11001110
15	f	00001111	79	4f	01001111	143	8f	10001111	207	cf	11001111
16	10	00010000	80	50	01010000	144	90	10010000	208	d0	11010000
17	11	00010001	81	51	01010001	145	91	10010001	209	d1	11010001
18	12	00010010	82	52	01010010	146	92	10010010	210	d2	11010010
19	13	00010011	83	53	01010011	147	93	10010011	211	d3	11010011
20	14	00010100	84	54	01010100	148	94	10010100	212	d4	11010100
21	15	00010101	85	55	01010101	149	95	10010101	213	d5	11010101
22	16	00010110	86	56	01010110	150	96	10010110	214	d6	11010110
23	17	00010111	87	57	01010111	151	97	10010111	215	d7	11010111
24	18	00011000	88	58	01011000	152	98	10011000	216	d8	11011000
25	19	00011001	89	59	01011001	153	99	10011001	217	d9	11011001
26	1a	00011010	90	5a	01011010	154	9a	10011010	218	da	11011010
27	1b	00011011	91	5b	01011011	155	9b	10011011	219	db	11011011
28	1c	00011100	92	5c	01011100	156	9c	10011100	220	dc	11011100
29	1d	00011101	93	5d	01011101	157	9d	10011101	221	dd	11011101
30	1e	00011110	94	5e	01011110	158	9e	10011110	222	de	11011110
31	1f	00011111	95	5f	01011111	159	9f	10011111	223	df	11011111
32	20	00100000	96	60	01100000	160	a0	10100000	224	e0	11100000
33	21	00100001	97	61	01100001	161	a1	10100001	225	e1	11100001
34	22	00100010	98	62	01100010	162	a2	10100010	226	e2	11100010
35	23	00100011	99	63	01100011	163	a3	10100011	227	e3	11100011
36	24	00100100	100	64	01100100	164	a4	10100100	228	e4	11100100
37	25	00100101	101	65	01100101	165	a5	10100101	229	e5	11100101
38	26	00100110	102	66	01100110	166	a6	10100110	230	e6	11100110
39	27	00100111	103	67	01100111	167	a7	10100111	231	e7	11100111
40	28	00101000	104	68	01101000	168	a8	10101000	232	e8	11101000
41	29	00101001	105	69	01101001	169	a9	10101001	233	e9	11101001
42	2a	00101010	106	6a	01101010	170	aa	10101010	234	ea	11101010
43	2b	00101011	107	6b	01101011	171	ab	10101011	235	eb	11101011
44	2c	00101100	108	6c	01101100	172	ac	10101100	236	ec	11101100
45	2d	00101101	109	6d	01101101	173	ad	10101101	237	ed	11101101
46	2e	00101110	110	6e	01101110	174	ae	10101110	238	ee	11101110
47	2f	00101111	111	6f	01101111	175	af	10101111	239	ef	11101111
48	30	00110000	112	70	01110000	176	b0	10110000	240	f0	11110000
49	31	00110001	113	71	01110001	177	b1	10110001	241	f1	11110001
50	32	00110010	114	72	01110010	178	b2	10110010	242	f2	11110010
51	33	00110011	115	73	01110011	179	b3	10110011	243	f3	11110011
52	34	00110100	116	74	01110100	180	b4	10110100	244	f4	11110100
53	35	00110101	117	75	01110101	181	b5	10110101	245	f5	11110101
54	36	00110110	118	76	01110110	182	b6	10110110	246	f6	11110110
55	37	00110111	119	77	01110111	183	b7	10110111	247	f7	11110111
56	38	00111000	120	78	01111000	184	b8	10111000	248	f8	11111000
57	39	00111001	121	79	01111001	185	b9	10111001	249	f9	11111001
58	3a	00111010	122	7a	01111010	186	ba	10111010	250	fa	11111010
59	3b	00111011	123	7b	01111011	187	bb	10111011	251	fb	11111011
60	3c	00111100	124	7c	01111100	188	bc	10111100	252	fc	11111100
61	3d	00111101	125	7d	01111101	189	bd	10111101	253	fd	11111101
62	3e	00111110	126	7e	01111110	190	be	10111110	254	fe	11111110
63	3f	00111111	127	7f	01111111	191	bf	10111111	255	ff	11111111