Micro Tutorial AVR

Por Nuno João a.k.a. Njay

http://troniquices.wordpress.com

http://embeddeddreams.com/

Versão 0.2

Conteúdo

Data	Autor	Comentário
15/09/2008	Njay	Adicionada a Parte I
05/07/2009	Njay	Adicionada a Parte II

Parte I

Este tutorial foi inicialmente escrito no fórum electronica-pt.com, por isso às vezes parece ter alguma linguagem "estranha" ou descontextualizada. Depois de alguns problemas com o serviço de hospedagem de imagens resolvi passá-lo para este formato com distribuição sob licença Creative Commons (Atribuição - Uso Não-Comercial - Partilha nos termos da mesma Licença 3.0 Portugal - http://creativecommons.org/licenses/by-nc-sa/2.5/pt/).

Delicia-te ;) (espero!).

Motivação

Bom, já vi que o pessoal por aqui, e pelo que percebi mesmo em Portugal, é mais adepto do PIC. Eu como gosto muito do AVR e o considero uma tecnologia bastante superior ao PIC, resolvi então iniciar este tutorial para ajudar a dar os primeiros passos a quem quiser experimentar um AVR. Por falta de tempo vou ter que ir escrevendo aos pedaços, mas espero que ainda assim o tutorial fique completo rapidamente. Também posso vir a re-escrever o texto; se virem isto mudar completamente não se assustem, estou a usar o próprio fórum como folha de rascunho. Mas comecemos.

Introdução

"AVR" é o nome de uma família de microcontroladores de 8 bits comercializada pela ATMEL. A arquitectura do AVR foi desenvolvida por 2 estudantes de doutoramento noruegueses em 1992 e depois proposta à ATMEL para comercialização. Para quem souber inglês, podem ver uma pequeno vídeo sobre os AVR aqui: http://www.avrtv.com/2007/09/09/avrtv-special-005/ .
O AVR consiste, tal como um PIC e outros microcontroladores, num processador (o "core"), memórias voláteis e não-voláteis e periféricos. Ao contrário do PIC, o core do AVR foi muito bem pensado e implementado desde o inicio, e o core que é usado nos chips desenhados hoje é o mesmo que saiu no 1º AVR há mais de 10 anos (o PIC teve "dores de crescimento" e o tamanho das instruções aumentou algumas vezes ao longo do tempo de forma a suportar mais funcionalidade).

Assim de uma forma rápida podemos resumir a arquitectura do AVR nos seguintes pontos:

- Consiste num core de processamento, memória de programa (não volátil, FLASH), memória volátil (RAM estática, SRAM), memória de dados persistentes (não volátil, EEPROM) e bits fuse/lock (permitem configurar alguns parâmetros especiais do AVR).
- Arquitectura de memória Harvard (memória de programa e memória de dados separadas)
- A memória volátil (SRAM) é contínua
- A maior parte das instruções têm 16 bits de tamanho, e é este o tamanho de cada palavra na memória de programa (FLASH).
- Execução de 1 instrução por ciclo de relógio para a maior parte das instruções.
- Existem 32 registos de 8 bits disponíveis e há poucas limitações ao que se pode fazer com cada um.
- Os registos do processador e os de configuração dos periféricos estão mapeados (são acessíveis) na SRAM.
- Existe um vector de interrupção diferente por cada fonte de interrupção.
- Existem instruções com modos de endereçamento complexo, como base + deslocamento seguido de auto-incremento/decremento do endereço.
- O conjunto de instruções foi pensado para melhorar a conversão de código C em assembly.

Para facilitar a vida a todos, a mim e a vocês, vou centrar o tutorial apenas num modelo específico de AVR, o ATtiny26, e em programação C, já que o assembly é muito *hardcore* para uma introdução. Mais tarde não será difícil extender o descrito neste tutorial para mais 1 ou 2 modelos de AVR sem grandes alterações. Este ATtiny26 apesar de ser pequeno não é dos modelos mais básicos, e na minha opinião consegue um excelente equilíbrio preço/funcionalidade; tem ADC, contadores/timers, hardware que facilita a construção de interfaces SPI/I^2C/UART, velocidade de relógio até 16MHz (um PIC para ter a mesma velocidade precisava de ter um clock de 64MHz), entre outras coisas.

Ferramentas de desenvolvimento

Vamos precisar das seguintes ferramentas, todas software de utilização livre:

- AVRStudio 4.13 (73MB) - Ambiente de desenvolvimento gratuito da ATMEL para toda a linha AVR. Consiste num editor, assembler, programador e simulador. Vamos usá-lo como simulador.
- WinAVR (23MB) - Ambiente de desenvolvimento OpenSource para AVRs. Consiste num editor, compilador C/C++, linker, debugger, simulador e programador. Vamos usar apenas as aplicações para compilação de C e o programador.

Se a tua ligação à Internet for lenta, podes deixar o download do AVRStudio para mais tarde, eu digo-te quando na altura (na verdade ele não é essencial para desenvolver programas para AVR, mas ajuda **imeeeeenso**).

Após a instalação do WinAVR, vamos ter que adicionar uma descrição do nosso hardware programador ao ficheiro de configuração do *avrdude*, que é a aplicação do WinAVR que trata da programação dos AVRs. Portanto abram o ficheiro <directoria-de-instalação>\WinAVR\bin\avrdude.conf e procurem pelo comentário de texto

```
# Parallel port programmers.
```

Este comentário marca o inicio da zona de configuração de programadores de porta paralela, e logo aí vamos inserir a descrição do (hardware) programador que vamos usar e que está descrito na próxima secção:

```
# AVR Tutorial programmer at www.electronicapt.com
programmer
  id    = "avrpt";
  desc  = "AVR Tutorial at ElectronicaPt.com";
  type  = par;
  reset = 6;
  sck   = 8;
  mosi  = 7;
  miso  = 10;
;
```

O *avrdude* é um programa muito flexível e não se limita a aceitar apenas um ou alguns tipos de programadores; ele lê do ficheiro avrdude.conf a configuração de programadores e usa-a para utilizar o programador de hardware que estejamos a usar. Assim podemos utilizar muitos tipos diferentes de programadores sem ter que alterar o código do *avrdude*; inteligente, não ;)? E ele faz o mesmo para os AVRs. Cada AVR tem memórias de tamanhos diferentes e toda essa informação está no ficheiro de configuração. Mas não é preciso mexer em mais nada :).

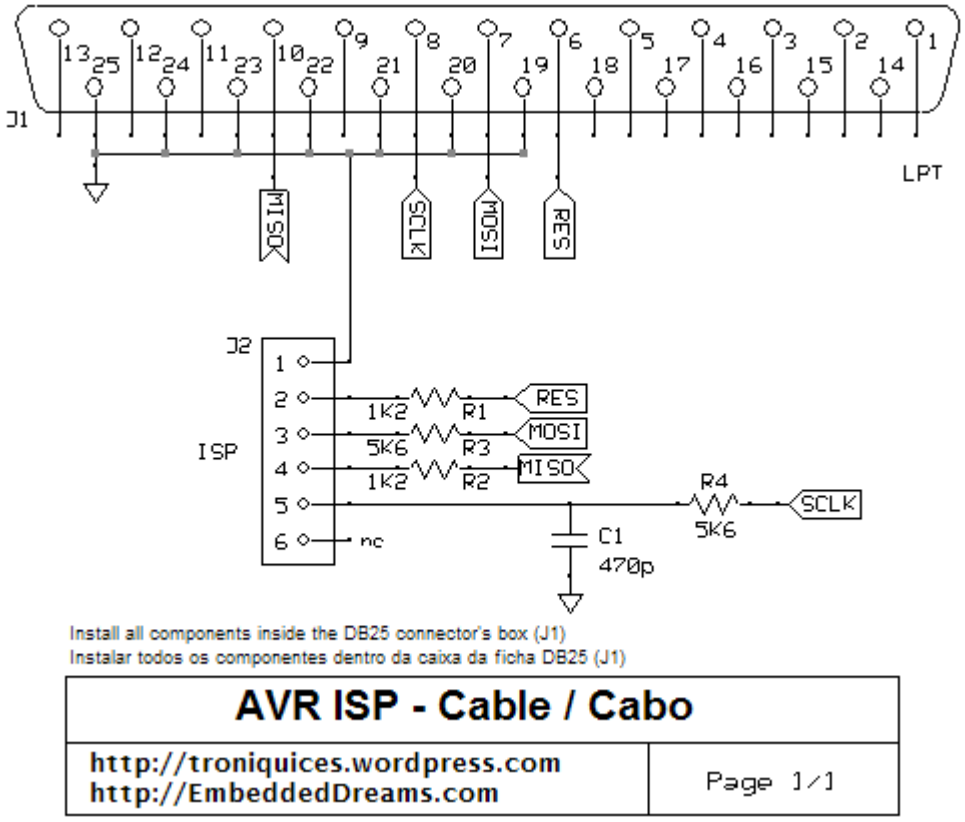
O programador de AVRs que vamos usar é de ligar à porta paralela do PC, o que quer dizer que o teu PC terá que ter uma LPT. Existem uns conversores de USB para LPT, mas normalmente não funcionam, tem que ser uma porta LPT "pura". Também terás que ter direitos de administração no teu PC para instalar um driver de porta paralela.

Agora vamos tratar do hardware.

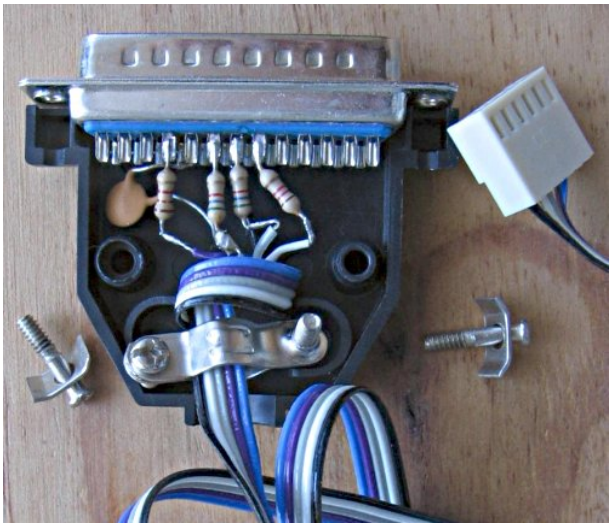
Hardware

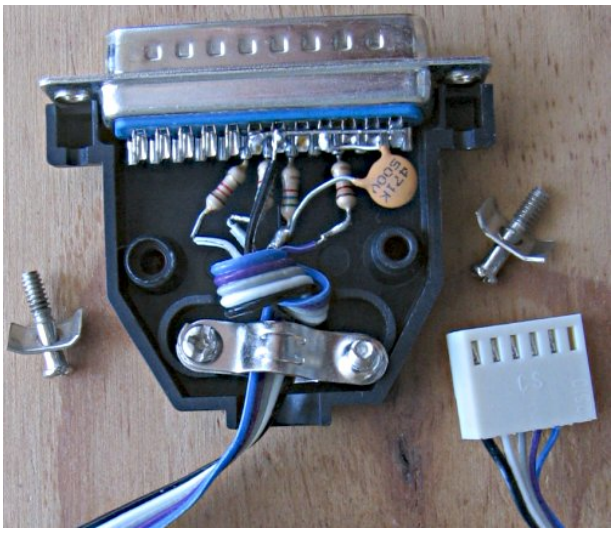
Programador

Precisamos de ter um circuito programador para gravar os nossos programas no AVR. O AVR tem 2 modos de programação: o paralelo/alta voltagem e o série/baixa voltagem. Estes modos permitem ler e escrever nas suas memórias não voláteis e nos bits fuse/lock. O circuito abaixo é um programador de modo série, que vamos utilizar.



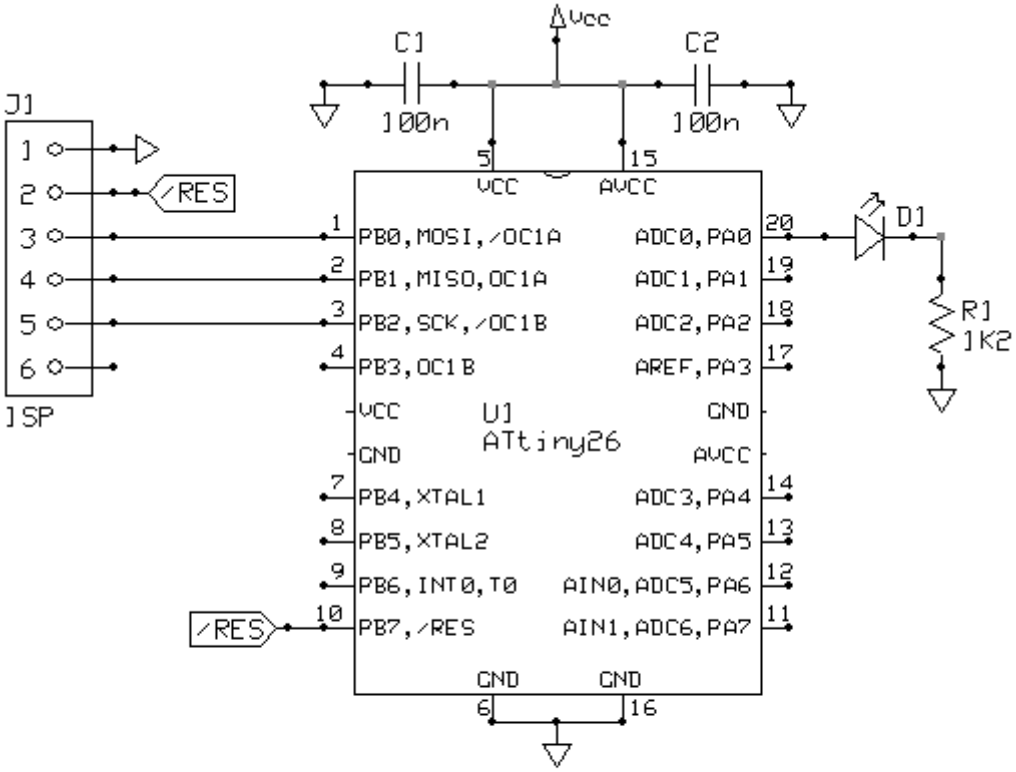
Este circuito tem que ser montado dentro da caixa da ficha que liga à porta paralela do PC, e é essencial que fique ali mesmo à saída da porta paralela (mais perto do que dentro da ficha que lá liga, impossível):





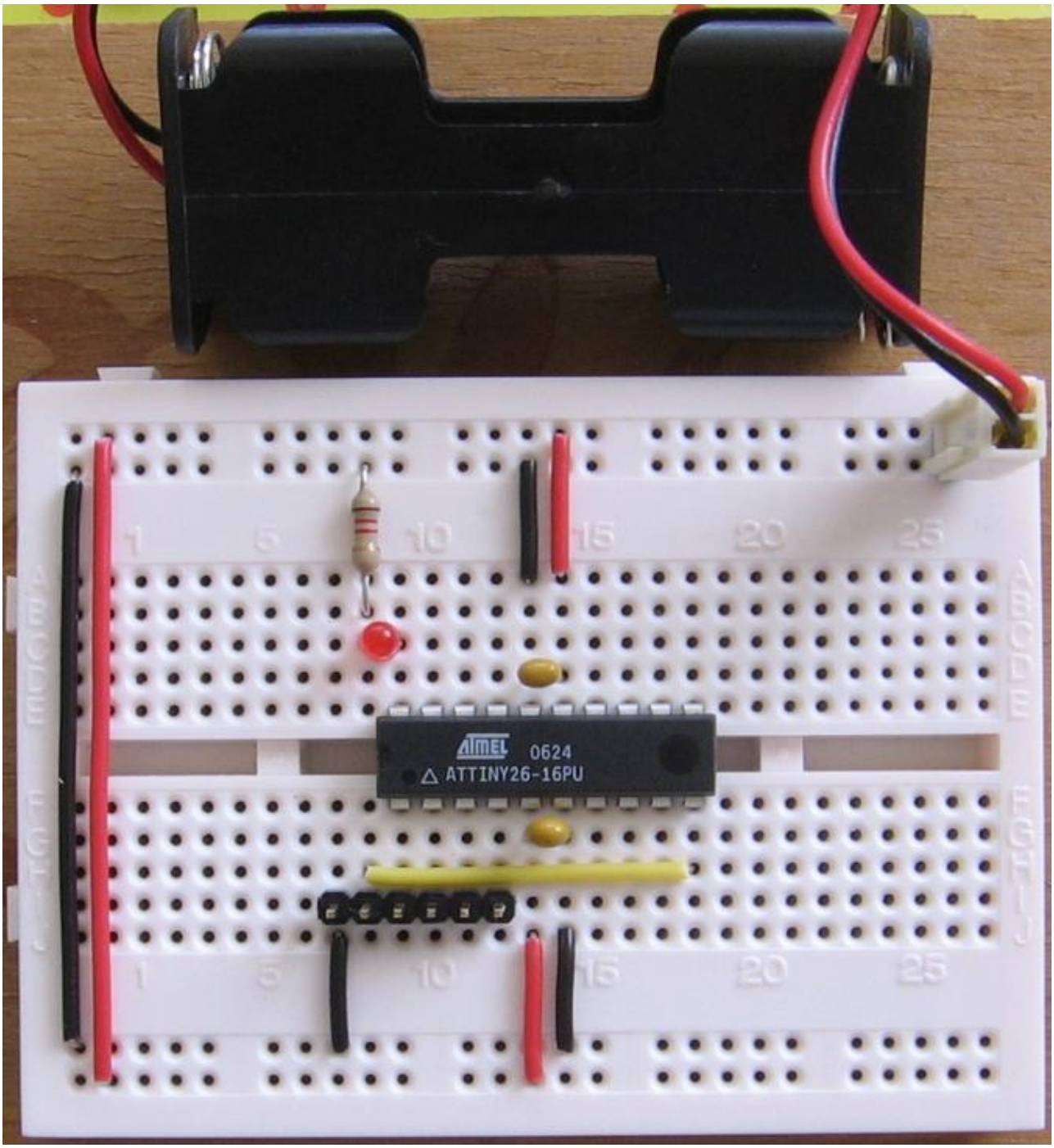
Circuito de Teste

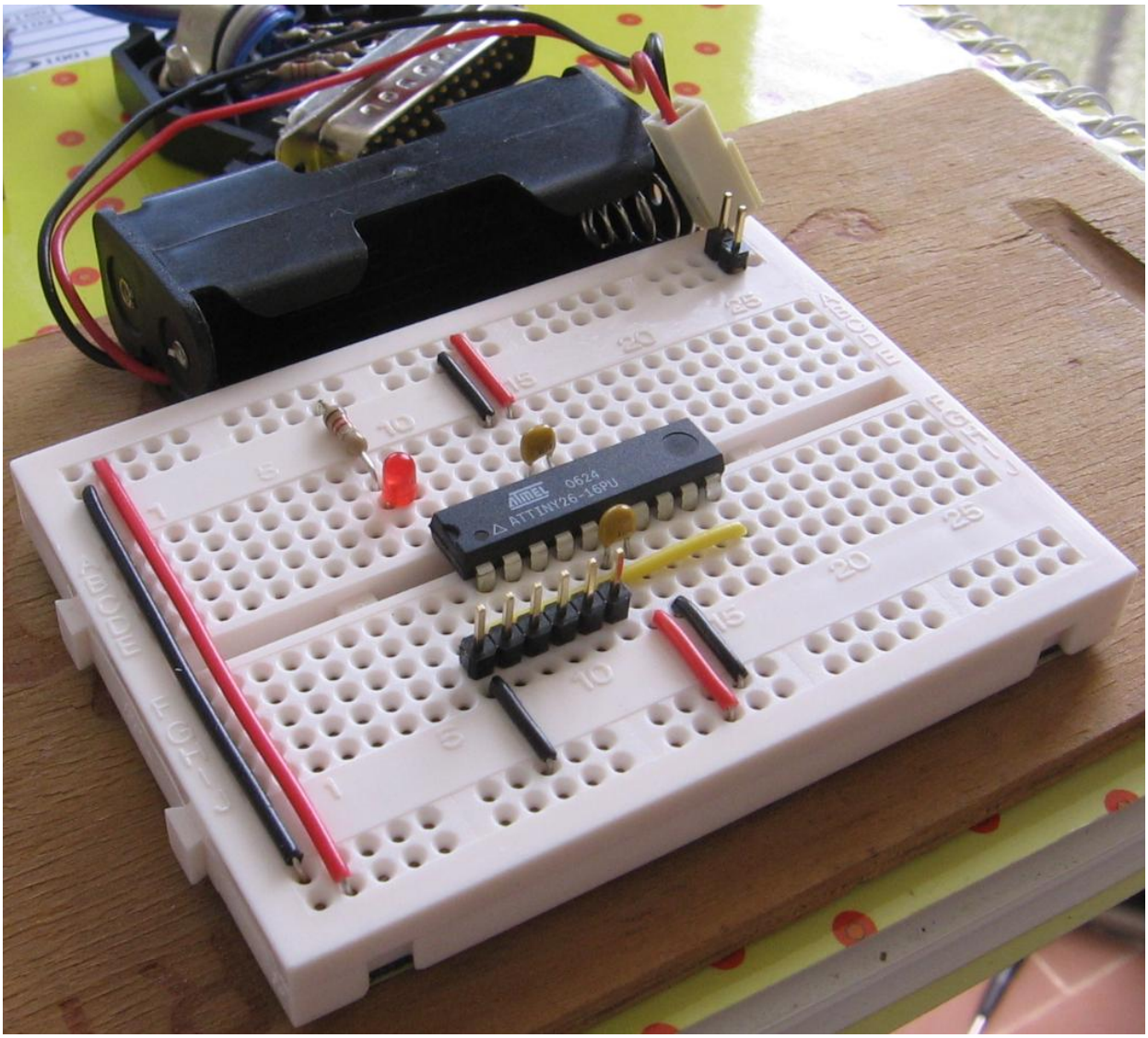
Vamos montar numa matriz de contactos o circuito abaixo. Este pode ser alimentado por qualquer tensão entre 4.5V e 5.5V, tipicamente 5V. No entanto a maior parte destes AVR funciona com tensões a partir de 3V, pelo que, para a nossa pequena introdução e com o AVR a baixa velocidade de relógio (1MHz), pode ser usado um par de pilhas AA ou AAA de 1.5V ou uma pilha de lítio tipo "botão" de 3V. Isto facilita para quem não tem uma fonte de 5V. Se o circuito não funcionar (se por exemplo apresentar erros de verificação da programação), então deve-se usar uma tensão maior, por exemplo 3 pilhas AA ou AAA em série.



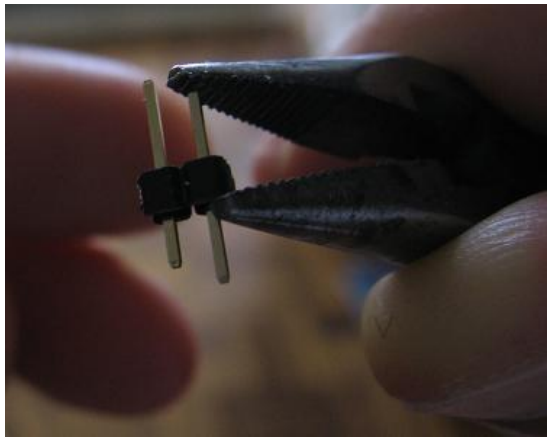
ATtiny26 - development nano-board / nano-placa de desenvolvimento	
http://troniquices.wordpress.com http://EmbeddedDreams.com	

Nesta placa temos apenas 1 LED como dispositivo externo sobre o qual o AVR pode actuar, mas depois deste tutorial vocês irão certamente ser criativos e ligar muitas outras coisas :).
Vou usar um suporte de 2 pilhas AA de 1.5V para alimentar o circuito:





Se já tentaste enfiar uma ponte de terminais (as peças dos pinos dourados) numa matriz de contactos, reparaste que eles não prendem lá, porque são demasiado curtos. Para resolver isso pega num alicate e empurra devagar os pinos 1 ou 2mm, usando a base de plástico como apoio para uma das pontas do alicate, assim:



Agora já se conseguem prender as pontes à matriz.

Eu usei pontes de terminais, mas também poderia ter usado fichas macho do tipo das brancas do programador e do suporte de pilhas que podes ver nas fotos. Esses até são um pouco melhores, porque são polarizados, isto é, só permitem que ligues as fichas numa das 2 posições possíveis, o que é importante (especialmente a fichas das pilhas!). Essas fichas são baratas e encontram-se com facilidade cá em Portugal.

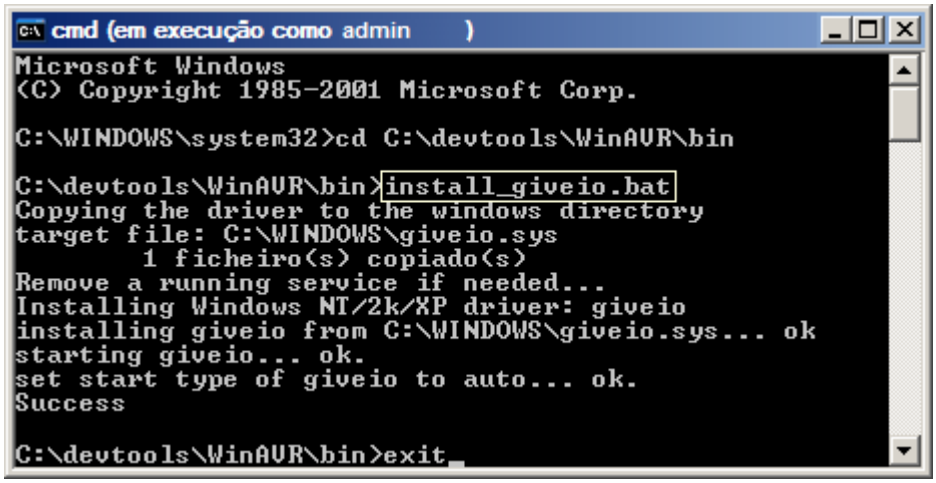
Teste ao hardware

Bom, antes de começarmos a fazer programas para o AVR, temos que verificar se o hardware (programador e circuito de teste) estão a funcionar. E vamos fazê-lo verificando se o software programador, o *avrdude*, é capaz de falar com o nosso AVR.

Para começar tens que instalar um driver no teu PC que irá permitir ao *avrdude* ter acesso à porta paralela. Para isso vai ao menu *Iniciar* do Windows e escolhe *Executar*; aí dá o comando

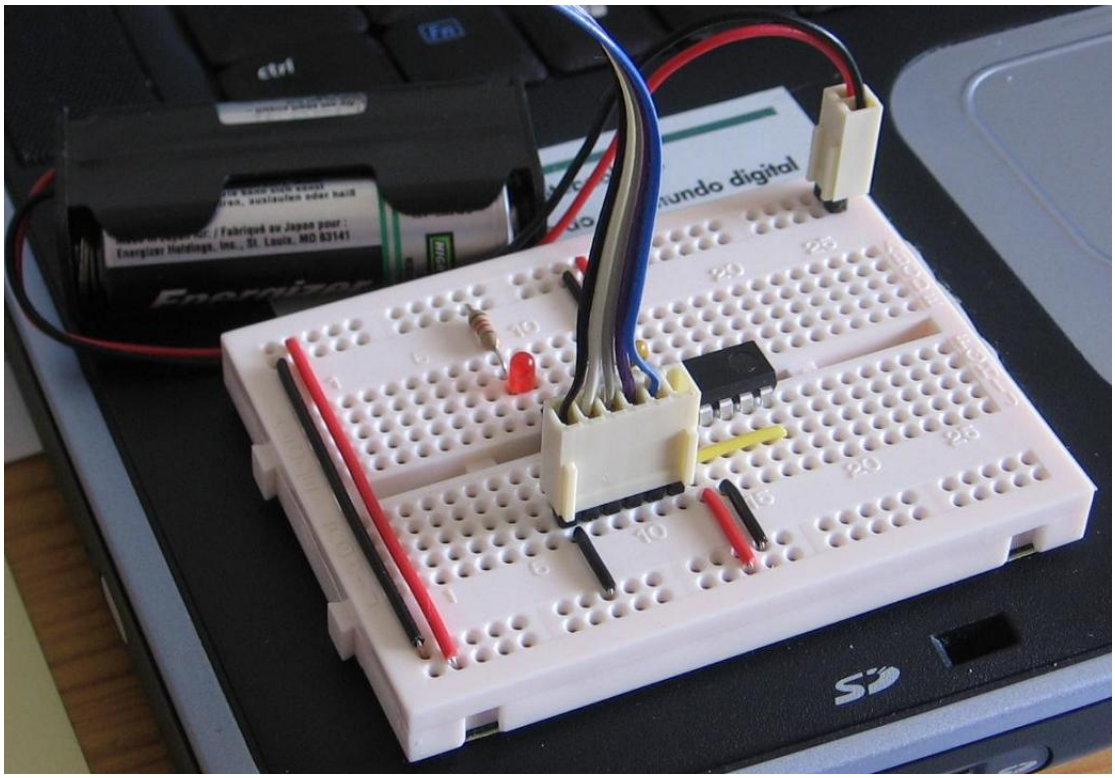
```
runas /user:administrador cmd
```

administrador é o nome de um utilizador no teu PC que tenha direitos de administração, e este comando abre uma *Linha de Comando* onde os comandos que deres se executarão como se fosses esse utilizador. Se o utilizador que usas habitualmente já tem direitos de administração, então basta-te abrir uma Linha de Comando directamente.
Na Linha de Comando então aberta, vai para a directoria ...\WinAvr\bin e corre o ficheiro install_giveio.bat para instalar o driver, como no exemplo:



Verifica se a instalação correu bem, deves ver as mesmas mensagens que na imagem acima. Em princípio só deves precisar de fazer isto uma vez, pois o driver fica instalado como serviço do Windows, de arranque automático. Já podes fechar a Linha de Comando e abrir uma nova, desta vez normal (Iniciar -> Executar -> escrever "cmd" e ENTER). Cria uma directoria de trabalho para este tutorial, por exemplo `c:\tutorial` e muda-te (`cd ...`) para lá.

Liga o programador à porta paralela e ao circuito de teste.



Tem atenção à posição da ficha branca. Depois liga as pilhas ao circuito de teste e vamos verificar se o *avrdude* consegue "falar" com o nosso AVR, dando mais um comando:

```
C:\tutorial> avrdude -c avrpt -p t26 -i 50
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.01s
avrdude: Device signature = 0x1e9109
avrdude: safemode: Fuses OK
avrdude done.  Thank you.
C:\tutorial>
```

O texto a laranja (ele não aparece laranja, fui eu que o pintei!) é uma boa indicação de que a conversa foi bem sucedida, uma vez que o *avrdude* conseguiu ler a assinatura do AVR, que é um código que identifica o modelo de AVR. Se houver mensagens de erro vais perceber... e aí tens que verificar se o cabo está bem montado, bem ligado, o circuito bem montado, se ligaste as pilhas, etc. Enquanto não vires a assinatura do AVR neste pequeno teste, não vais conseguir programá-lo.

Com o hardware a funcionar, vamos então começar a olhar para o software.

Software

Se tiveste a coragem de chegar a este ponto, estás (se não estás, devias estar :)) em pulgas para programar o teu 1º AVR :). Vamos primeiro ver como é que se compila e grava um programa no AVR, e só depois vamos à explicação dos detalhes.
Vamos fazer o programa que é o sonho de qualquer iniciante ;), pôr um LED a piscar. Copia o seguinte para um ficheiro e grava-o com o nome `pisca_led.c`.

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

int main (void)
{
    DDRA = 0x01;    // Configurar pino PA0 como output

    do {  // inicio de ciclo
        PORTA = 0x01;       // Acender o LED
        _delay_ms(250);     // Esperar 250ms...
        PORTA = 0x00;       // Apagar o LED
        _delay_ms(250);     // Esperar 250ms...
    }
    while (1);      // Saltar para o inicio do ciclo
}
```

E para compilar este programa vamos dar os comandos:

```
C:\tutorial> avr-gcc -mmcu=attiny26 -DF_CPU=1000000UL -g -O1 -o pisca_led.elf pisca_led.c
C:\tutorial> avr-objcopy -j .text -j .data -O binary pisca_led.elf pisca_led.bin
C:\tutorial>
```

Finalmente, para programar o AVR damos agora o último comando e observamos, com espanto :), o AVR ser programado:

```
C:\tutorial> avrdude -c avrpt -p t26 -i 50 -U flash:w:pisca_led.bin:r

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## |  100% 0.01s

avrdude: Device signature = 0x1e9109
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "pisca_led.bin"
avrdude: writing flash (108 bytes):

Writing | ################################################## | 100% 0.38s

avrdude: 108 bytes of flash written
avrdude: verifying flash memory against pisca_led.bin:
avrdude: load data flash data from input file pisca_led.bin:
avrdude: input file pisca_led.bin contains 108 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.35s

avrdude: verifying ...
avrdude: 108 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done.  Thank you.

C:\tutorial>
```

Assim que termina a execução deste comando, e se for bem sucedido tal como no exemplo acima, terás diante dos teus olhos um LED a piscar :). Parabéns :)!!!

Parte II

Neste novo micro-artigo, a juntar ao micro-tutorial, vou falar-vos de GPIOs - General Purpose Input/Output (entrada/saída de uso geral). Quando estiver completo vou juntar ao 1º post, mas assim podem ir já comentando e eu posso ir melhorando enquanto não termino.

Vou tentar explicar-vos de uma forma genérica o que é e como se usa um GPIO, pois um GPIO é um **conceito**, e que se encontra em todos os microcontroladores e SoC que vos poderão passar pelas mãos. Quem entender o conceito, consegue pegar num novo tipo de microcontrolador e rapidamente percebe como é que se usam os seus pinos para as funções mais básicas. No final do artigo vou então explicar como se fazem estas configurações num AVR.

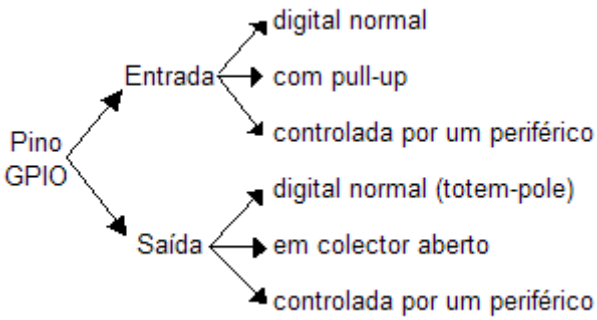
Neste artigo traduzo os termos para Português mas vou usar os termos em inglês, pois é o que vocês vão ver normalmente. Basta pensar que as datasheets estão sempre em inglês.

Então comecemos...

GPIO - General Purpose Input/Output

O conceito de GPIO surge como uma forma de se tornar um chip mais flexível, e deve ter surgido com os chips programáveis. Este conceito consiste em podermos configurar um pino de um chip para poder ter uma de entre várias funções, como por exemplo uma entrada ou uma saída. Isto tem vantagens óbvias na flexibilidade de um chip, pois o fabricante dá-vos um chip com N pinos em que vocês escolhem a função de cada um conforme as necessidades da vossa aplicação. Se a função de cada pino fosse sempre fixa, os chips seriam muito menos úteis, e provavelmente teríamos chips maiores, com muito mais pinos, numa tentativa de colmatar essa limitação, e não poderíamos alterar a função do pino durante o funcionamento do chip.

A função de base que podemos escolher para um GPIO é se o respectivo o pino é uma entrada ou saída, mas não é a única. Existem outras funções que podem ser escolhidas, embora nem todos os chips suportem todas. As funções possíveis mais comuns são:



Normalmente dizemos apenas "GPIO" quando nos estamos a referir a um "pino GPIO" e eu assim farei daqui para a frente. Vamos ver com mais detalhe cada função, a que às vezes também chamamos "tipo de pino".

Entrada digital normal

Esta é talvez a configuração mais simples que podemos ter. O pino funciona como uma entrada digital, ou seja, só podemos ler (em software) um de 2 valores: 0 ou 1. Na prática os valores 0 e 1 representam uma certa tensão que é aplicada ao pino, resultando numa leitura de 0 ou 1 por parte do software. As tensões mais comuns são 0V para representar um 0 e 5V para representar um 1, mas podem ser outras como por exemplo 3.3V ou 1.8V para representar um 1, dependendo da tensão de alimentação do chip (refiro apenas "chip" porque não são apenas os microcontroladores que têm GPIOs; por exemplo as FPGA, outro tipo de chip programável, também têm).

Portanto, num sistema que funcione com uma tensão de alimentação de 5V, se aplicarmos 5V a um pino configurado como "entrada digital normal", o software irá ler um valor 1 desse pino. Se aplicarmos 0V, o software irá ler um 0. A leitura do "estado do pino" é habitualmente efectuada lendo-se um registo do chip. Falaremos mais sobre isto no final.

Configurar um GPIO como entrada digital normal também serve como forma de **desligar** o pino do circuito. Neste caso não estamos interessados em ler valores. Ao configurá-lo como entrada, ele não afecta electricamente (de um ponto de vista digital) o circuito exterior ao chip e portanto é como se tivéssemos cortado o pino do chip. Diz-se que o pino está em "alta impedancia" ("high-Z" em inglês, pois o "Z" é muito usado para designar "impedancia"), "no ar", ou simplesmente "desligado do circuito".

Normalmente dizemos apenas que um pino está "configurado como entrada" ou ***como input***.

Entrada com pull-up ("puxa para cima")

Então se tivermos um pino configurado como input mas não lhe aplicarmos nenhuma tensão, que valor lemos no software?... A resposta é: **não podemos prever**. Tomem bem atenção a isto, vou repetir: **não podemos prever**. Quando temos uma entrada que está no ar, não podemos prever que valor vamos ler; o valor pode estar estável em 0 ou 1 ou pode estar sempre a variar, ou mudar de vez em quando consoante é dia ou noite ou Marte está alinhado com Júpiter ou o vizinho deitar-se 2 minutos mais cedo ou mais tarde. Ele pode até mudar só de lhe tocarem com o dedo.

Em algumas situações queremos ter **sempre** um valor estável na entrada. Um caso tipico é um interruptor (que pode ser um botão). Conectamos o interruptor entre a massa (o negativo da tensão de alimentação, "0V") e o pino. Aí, quando ligamos o interruptor (posição "ON"), o pino fica ligado aos 0V e portanto o chip lê um 0. Mas, e quando o interruptor está desligado? Aí o pino está no ar pois o interruptor desligado é um circuito aberto, e já sabemos que ler um input que está no ar dá-nos um valor aleatório e portanto nunca vamos saber se o interruptor está mesmo ON ou OFF. É aqui que o pull-up entra; ao configurarmos o pino com pull-up, o chip liga internamente uma resistência entre o pino e a tensão de alimentação, e portanto, quando não há nada electricamente ligado ao pino, o pino "vê" um 1. No caso do interruptor, quando este está OFF, o pull-up coloca um 1 estável à entrada do pino e fica resolvido o problema. Quando o interruptor está ON, o próprio interruptor força um 0 no pino, ligando-o à massa.

Então mas... se o pull-up puxa o pino "para cima" e o interruptor (quando está ON) puxa para baixo, não há aqui uma espécie de conflito? Não há, por uma simples razão: o valor da **resistência de pull-up** é alto (tipicamente mais de 100 KOhms) e portanto tem "pouca força". Como o interruptor liga o pino directamente à massa, é esta que "ganha". Diz-se até que o pull-up é um "pull-up fraco", ou "weak pull-up" em inglês.

Esta expressão pull-up ("puxa para cima") vem de estarmos a ligar à tensão de alimentação **positiva**, que é mais "alta" do que a massa, os 0V. Para este termo contribui ainda o facto de geralmente se desenhar a linha de alimentação positiva no topo dos esquemas, e a massa em baixo. Também podemos falar em **pull-down** ("puxa para baixo") quando nos referimos a ligar à massa. Podemos criar pull-downs ligando resistências à massa, mas tipicamente os chips não suportam este tipo de pull, por razões que fogem ao âmbito deste artigo que se quer simples.

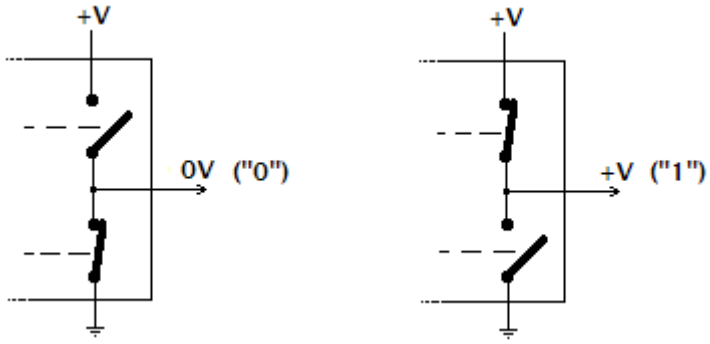
Entrada controlada por um periférico

Neste caso deixamos de ter controlo sobre o GPIO, passando esse controle para um periférico interno do chip. Por exemplo a linha Rx (recepção) de uma porta série (UART). Aqui o pino funciona como uma entrada mas quem a controla é o periférico.

Saída digital normal

Na lógica CMOS, com a qual trabalhamos mais hoje em dia, as saídas digitais são baseadas numa topologia designada "totem-pole". Este tipo de saída é constituído por 2 interruptores electrónicos (transístores) um em cima do outro, e controlados de forma a que:

1) quando queremos ter um "zero" à saída, liga-se o transístor de baixo, que liga o pino à massa (0V); o transístor de cima mantém-se desligado
2) quando queremos ter um "um" à saída, liga-se o transístor de cima, que liga o pino à tensão se alimentação (+V); o transístor de baixo mantém-se desligado



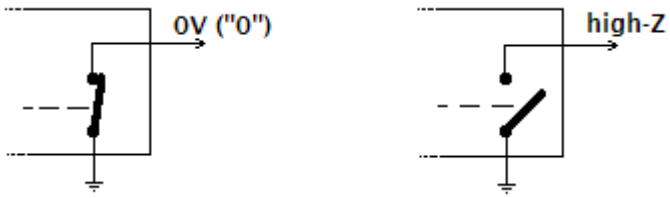
Na imagem acima podemos ver uma saída totem-pole num dos seus 2 estados mais habituais: quando tem um 0 e quando tem um 1.

Por aqui podemos ver por exemplo porque é que não se devem ligar 2 (ou mais) saídas umas às outras. Se uma delas estiver com um "1" e a outra com um "0", estamos a criar um curto-circuito na alimentação, ligando +V à massa. Numa das saídas está ligado o interruptor de cima e na outra está ligado o de baixo. Mesmo que isto só aconteça durante um período de tempo muito pequeno (milisegundos, microsegundos ou menos), vão passar correntes elevadas, fora das especificações dos chips, e se acontecer regularmente, começa um processo de degradação que leva à falha do chip em segundos, horas, semanas, meses ou anos.

Uma saída totem-pole tem ainda um 3º estado: "no ar". É outra forma de desligar um pino, mas que é usada quando o pino é sempre uma saída (não configurável). No caso de um GPIO, este pode ser configurado como entrada ficando assim desligado do circuito exterior, como vimos atrás.

*Saída em colector aberto (*open colector*)*

Este tipo de saída surgiu para resolver o problema de não se poder ter 2 ou mais saídas ligadas. Por esta altura vocês podem estar a pensar "mas porque raio é que haveríamos de querer 2 saídas ligadas uma à outra?!", e a resposta é simples: pensem por exemplo no I^2C, em que temos linhas de dados bidireccionais. No I^2C há vários chips ligados a um mesmo par de linhas e todos eles têm a possibilidade de transmitir dados nessas linhas. Daí que, de alguma forma, há várias saídas ligadas entre si.



A saída em colector aberto consiste num simples interruptor electrónico (transístor) capaz de ligar o pino à massa. Quando o interruptor está ligado a saída é 0, e quando está desligado a saída é... não sabemos. O pino fica "no ar" e portanto qualquer outro dispositivo exterior ao chip que esteja ligado ao pino pode lá colocar a tensão que entender. Num bus I^2C o que se passa é que existe uma resistência externa que mantém as linhas com tensão positiva quando nenhum dispositivo está a transmitir; ou seja, temos um "pull-up fraco". A partir daí, qualquer um dos dispositivos pode forçar uma das linhas I^2C a ter 0V, se activar o interruptor electrónico na sua saída em colector aberto.

Se 2 ou mais dispositivos activarem os interruptores das suas saídas não há nenhum conflito eléctrico, pois estão todos a ligar a mesma linha aos 0V. Problema resolvido!

Fico-me por aqui quanto às saídas Open Colector, pois os micro-controladores de 8 bits como o AVR não permitem configurar pinos neste modo. De qualquer forma ficam algumas luzes que espero vos sejam úteis para entender alguns circuitos que vos passem pelas mãos.

Saída controlada por um periférico

À semelhança do que se passa no caso da entrada controlado por um periférico, neste caso deixamos de ter controlo sobre o GPIO, passando esse controle para um periférico interno do chip. Por exemplo a linha Tx (transmissão) de uma porta série (UART). Aqui o pino funciona como uma saída mas quem a controla é o periférico (no caso da UART, uma saída normal).

GPIOs na Arquitectura AVR

Nos AVR os pinos estão agrupados em portos com no máximo 8 pinos cada. Os portos têm a designação de letras, A, B, C, etc, e cada AVR tem um conjunto de portos. Cada pino de um porto pode ser configurado num de 3 modos:

1) Entrada normal
2) Entrada com pull-up

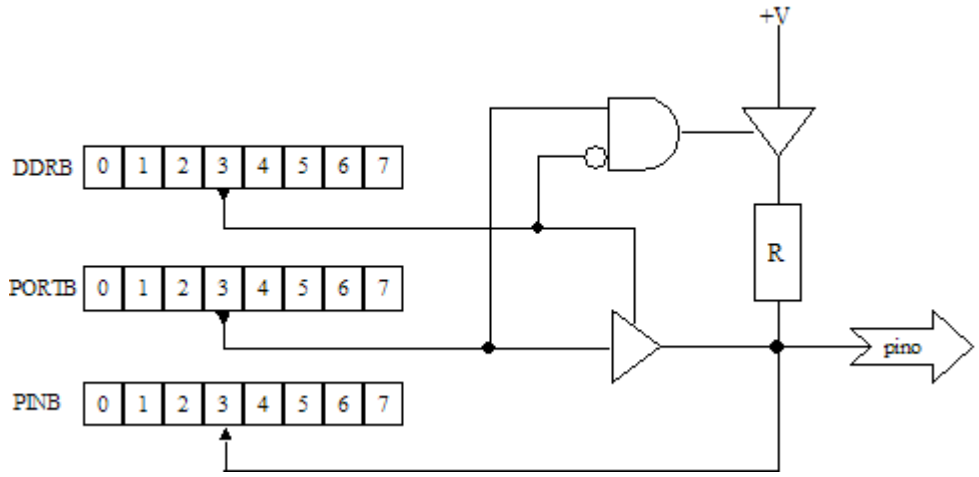
3) Saída normal
4) Entrada ou saída com controlada por um periférico

Um pino entra no 4º modo quando o respectivo periférico é activado, pelo que não vamos debruçar-nos aqui sobre isso.

A cada porto está associado um conjunto de 3 registos que são usados para configurar, ler e definir o estado de cada pino do porto individualmente. Cada bit de cada registo está associado ao respectivo pino do chip.

1) PINx - Lê o estado actual do pino
2) DDRx - Data Direction Register (registo de direcção dos dados)
3) PORTx - Define o estado da saída do porto

O "x" depende da letra do porto, de modo a que temos por exemplo o registo DDRA para o porto A. Segue-se um diagrama simplificado da lógica associada a cada pino de um porto do AVR, neste caso exemplificado para o pino 3 do porto B (designado B3):



Todos os pinos do chip têm uma lógica similar a este diagrama.

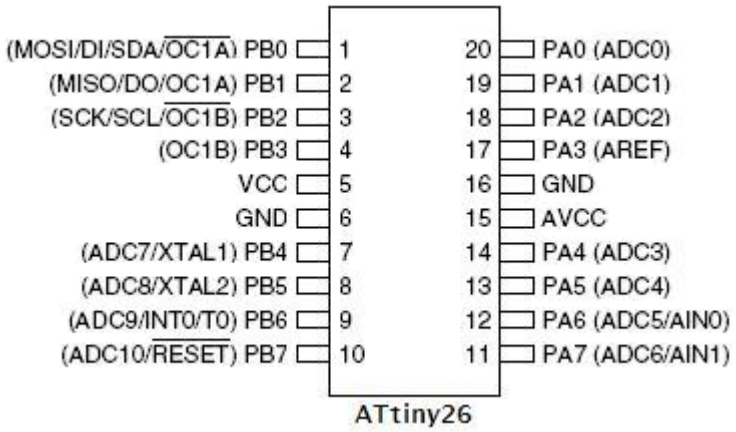
O registo PINx apresenta sempre o valor lógico ("0" ou "1") que estiver presente num pino **independentemente** da configuração. É como se o registo estivesse ali a medir a tensão directamente no pino e a reportar o seu valor lógico ao software.

O registo DDRx define, para cada pino do porto, se é uma entrada ou uma saída. Após o reset do chip, todos os pinos estão configurados como entradas, e portanto é como se todo o chip estivesse desligado do exterior, tem todos os pinos no ar. Para configurar um pino como saída temos que colocar a 1 o respectivo bit no registo DDRx.

Se um pino estiver configurado como uma saída (se o respectivo bit no registo DDRx for 1), podemos então definir o estado da saída com o respectivo bit no registo PORTx.

O PORTx controla ainda o pull-up interno quando um pino está configurado como entrada. Se o respectivo bit no PORTx estiver a 1, então o pull-up está activado.

Cada AVR tem um certo número de portos. Cada porto pode não ter pinos físicos do chip associados a todos os seus bits. As datasheets da ATMEL (fabricante dos AVR) apresentam logo na 2ª página o **pinout** (a atribuição de funcionalidade aos pinos do chip). Vamos pegar na datasheet por exemplo no ATtiny26, o AVR que usei na 1ª parte do tutorial; o pinout é este:



Isto diz-nos que este modelo de AVR tem 2 portos, A e B, em que todos os bits de ambos os portos estão associados a pinos do chip. Logo, para configuração dos pinos, este AVR tem os registos DDRA, PORTA, PINA, DDRB, PORTB, e PINB. Os nomes entre parêntesis são os nomes associados aos periféricos do AVR quando estes estão ligados; vamos esquecê-los neste artigo.

Configuração dos Portos em Linguagem C com WinAVR

É muito fácil aceder aos registos de configuração dos GPIOs (e não só) com este compilador: eles têm exactamente os nomes dos registos e usam-se como variáveis. Assim, existe por exemplo a variável DDRA e podemos escrever instruções como:

```
DDRA = 0xff;   // configurar todos os GPIOs do porto A como saídas
```

Se quisermos configurar apenas alguns GPIOs, temos disponível a macro _BV(index) que cria uma máscara de bits para um determinado bit do registo. Esta macro retorna um número de 8 bits em que apenas o bit de indice index é 1. Exemplos: _BV(0) é 1, _BV(7) é 128 (0x80), _BV(2) é 4.

Agora seguem-se alguns exemplos de configuração. Para configurar apenas o GPIO PA3 como saída e todos os restantes como entradas,

```
DDRA = _BV(PA3);   // configurar apenas o GPIO PA3 (pino 17) como saída
```

e para configurar vários GPIOs como saídas, basta efectuar um OU bit-a-bit

```
DDRA = _BV(PA3) | _BV(PA6);   // configurar os GPIOs PA3 e PA6 (pinos 17 e 12) como saídas
```

Depois bastaria colocar no registo PORTB, no respectivo bit (3), o valor que queremos que "apareça" no pino.

Para ligar o pull-up de um GPIO basta garantir que o respectivo bit está a zero no registo DDRx e depois colocar a 1 o bit no registo PORTx. Configurar o GPIO PB1 como entrada com pull-up seria assim:

```
DDRB &= ~_BV(PB1);   // configurar PB0 como entrada
PORTB |= _BV(PB1);   // ligar o pull-up
```

Aqui introduzi mais alguma sintaxe de C. Recomendo a leitura de outras referências acerca de C e operações lógicas de bits, mas assim por alto e para quem não conhece, deixo umas dicas superficiais.

`DDRB &= ~_BV(PB1)` é um "E" (AND) do registo DDRB com o valor _BV(PB1) logicamente negado (os bits invertidos); isto tem o efeito de colocar a zero apenas o bit 1 de DDRB.

`PORTB |= _BV(PB1)` é um "OU" (OR) do registo PORTB com o valor _BV(PB1); isto tem o efeito de colocar a 1 apenas o bit 1 de DDRB.

Portanto é muito fácil configurar os GPIOs em C. Quem é capaz de escrever o código C para configurar os GPIOs deste diagrama?