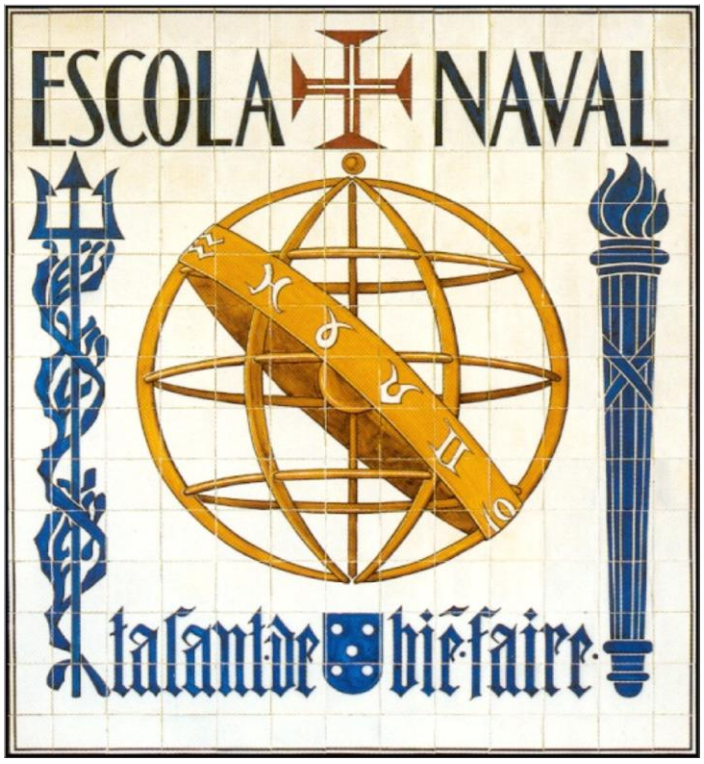
ESCOLA NAVAL

DEPARTAMENTO DE ENGENHEIROS NAVAIS

RAMO DE ARMAS E ELECTRÓNICA



Arduino

Introdução e Recursos Avançados

Nuno Pessanha Santos

ASPOF EN-AEL

nuno.pessanha.santos@marinha.pt

2009

"Não se pode ensinar tudo a alguém, pode-se apenas ajudá-lo a encontrar por si mesmo."

Galilleu Galilei

Índice

1. Introdução

Este tutorial tem como principal objectivo ser uma referência para a utilização da placa de desenvolvimento *Arduino*, não só para quem se está a iniciar no assunto, mas, também, para quem procura aprofundar mais o seu conhecimento sobre o tema.

O *Arduino* é uma ferramenta de desenvolvimento "*open source*", tendo surgido de um projecto académico - quem sabe um projecto seu não poderá ter o mesmo sucesso?... Como ferramenta é usualmente associado à filosofia de "Physical Computing", ou seja, ao conceito que engloba a criação de sistemas físicos através do uso de *Software* e *Hardware* capazes de responder a *inputs* vindos do mundo real.

Podemos designar o *Arduino* simplesmente como uma peça de *Hardware* ou um *Software* de desenvolvimento, mas é muito mais que isso. Devido ao sucesso que tem vindo a alcançar ao longo do tempo, existe uma enorme comunidade de utilizadores/seguidores em todo o Mundo, podendo afirmar-se que o *Arduino* representa também uma enorme comunidade. As razões para tal sucesso baseiam-se no seu baixo custo - dadas as suas funcionalidades -, a simplicidade na utilização e a possibilidade de utilização em vários sistemas operativo, como o Windows, Macintosh OS e Linux - capacidade essa denominada por "Cross-platform".

O estudo do *Arduino* abre-nos portas à compreensão de uma importante ferramenta de desenvolvimento através de uma aprendizagem simples mas dedicada, onde podemos fazer desde robots a domótica entre muitas outras aplicações, bastando simplesmente ter imaginação.

Devido à sua enorme utilização, como foi referido anteriormente, torna-se um assunto quase que obrigatório a abordar, sendo o seu conhecimento uma mais-valia para todos os interessados pela electrónica (e não só).

Assim, espera-se que este tutorial seja uma pequena ajuda para a aquisição de conhecimento sobre a temática em estudo, que deverá pautar pela dedicação.

2. Hardware vs Software

Antes de começar a trabalhar propriamente com esta plataforma de desenvolvimento, torna-se necessário começar por perceber o seu funcionamento, para tal descrever-se-á neste capítulo o funcionamento do *Arduino,* em termos de Hardware e Software.

Espera-se assim de certa forma "iluminar o caminho" neste estudo, fazendo com que toda a percepção do assunto a tratar se torne mais simples, ou seja, serão aqui apresentadas as bases para a sua compreensão.

2.1. Hardware

De seguida, far-se-á uma exposição do tema incidindo na placa *Arduino Duemilinove* (disponível na Escola Naval -EN), em tudo semelhante em termos de utilização à placa de desenvolvimento *Arduino Mega*. Contudo verificam-se pequenas diferenças como a diferença na sua capacidade de armazenamento (memória disponível), o número de pinos analógicos, o número de pinos digitais e as dimensões.

O *Arduino Duemilinove* (Fig. 1) apresenta-se com o microcontrolador ATMega168 ou ATMega328, enquanto o Arduino Mega (Fig.2) apresenta-se com um microcontrolador ATMega1280.

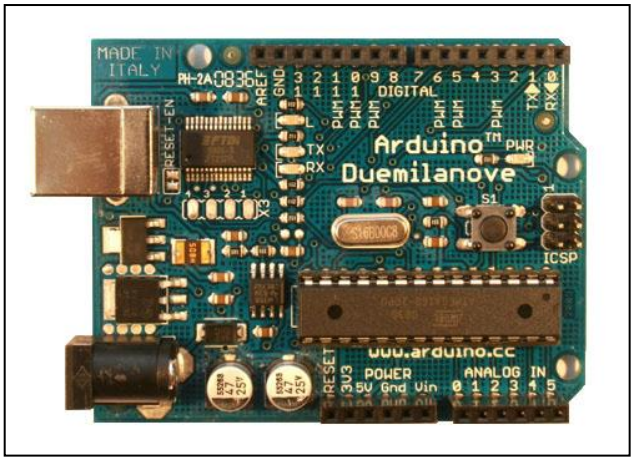


Fig. 1 – *Arduino Duemilinove*
Fonte: Site oficial *Arduino* (www.arduino.cc)

Fig. 2 – *Arduino Mega*
Fonte: Site oficial *Arduino* (www.arduino.cc)

Vão ser abordados neste subcapítulo dois temas de grande importância relativamente ao *Arduino Duemilinove*, são eles: "Tipos e quantidades de memória disponíveis" e o "*pinout* disponível". Mais informações relativas ao microcontrolador em estudo, ou qualquer outro, devem ser consultados os respectivos *datasheet*.

Mas antes de abordar os temas referidos anteriormente torna-se necessário fazer a distinção entre microprocessador e microcontrolador, que por vezes é alvo de confusão.

2.1.1. Microprocessador

Um microprocessador, basicamente, é constituído por um circuito integrado com a capacidade de executar determinadas instruções, sendo a sua velocidade de processamento determinada por um circuito que produz um determinado *Clock* (kHz, MHz ou GHz).

O seu poder de processamento é afectado por características como Bits, quantidade de núcleos, arquitectura apresentada, tipo de instruções, entre outras. Como factor de grande importância tem-se ainda a existência de memória externa, onde estão armazenados os programas que serão executados.

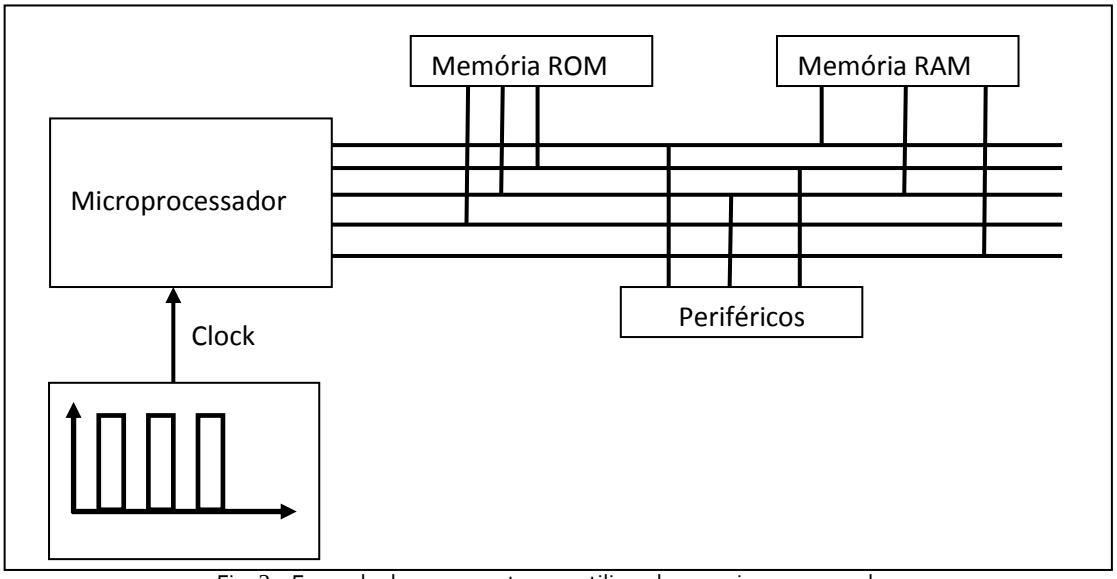


Fig. 3 – Exemplo de uma montagem utilizando um microprocessador

2.1.2. Microcontrolador

Um microcontrolador, ao contrário de um microprocessador, é desenhado e construído de forma a integrar diversos componentes num único circuito integrado, evitando, assim, a necessidade de adicionar componentes externos ao microcontrolador, que permitiriam as suas funcionalidades.

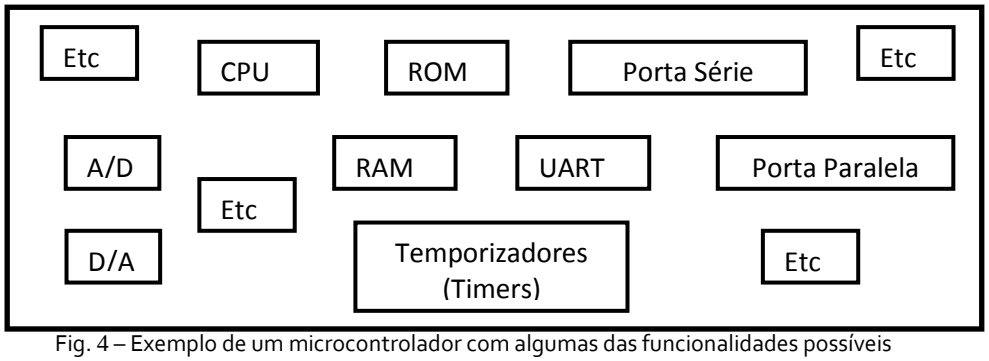


Fig. 4 – Exemplo de um microcontrolador com algumas das funcionalidades possíveis

Pela análise da figura acima, pode-se ter uma noção de alguns exemplos de componentes que se encontram disponíveis, conseguindo reunir uma grande quantidade de recursos num único circuito integrado.

Na figura seguinte, é apresentado um diagrama de blocos de um microcontrolador ATMega168 – em tudo idêntico a um ATMega328 – em que é possível identificar todos os seus constituintes.

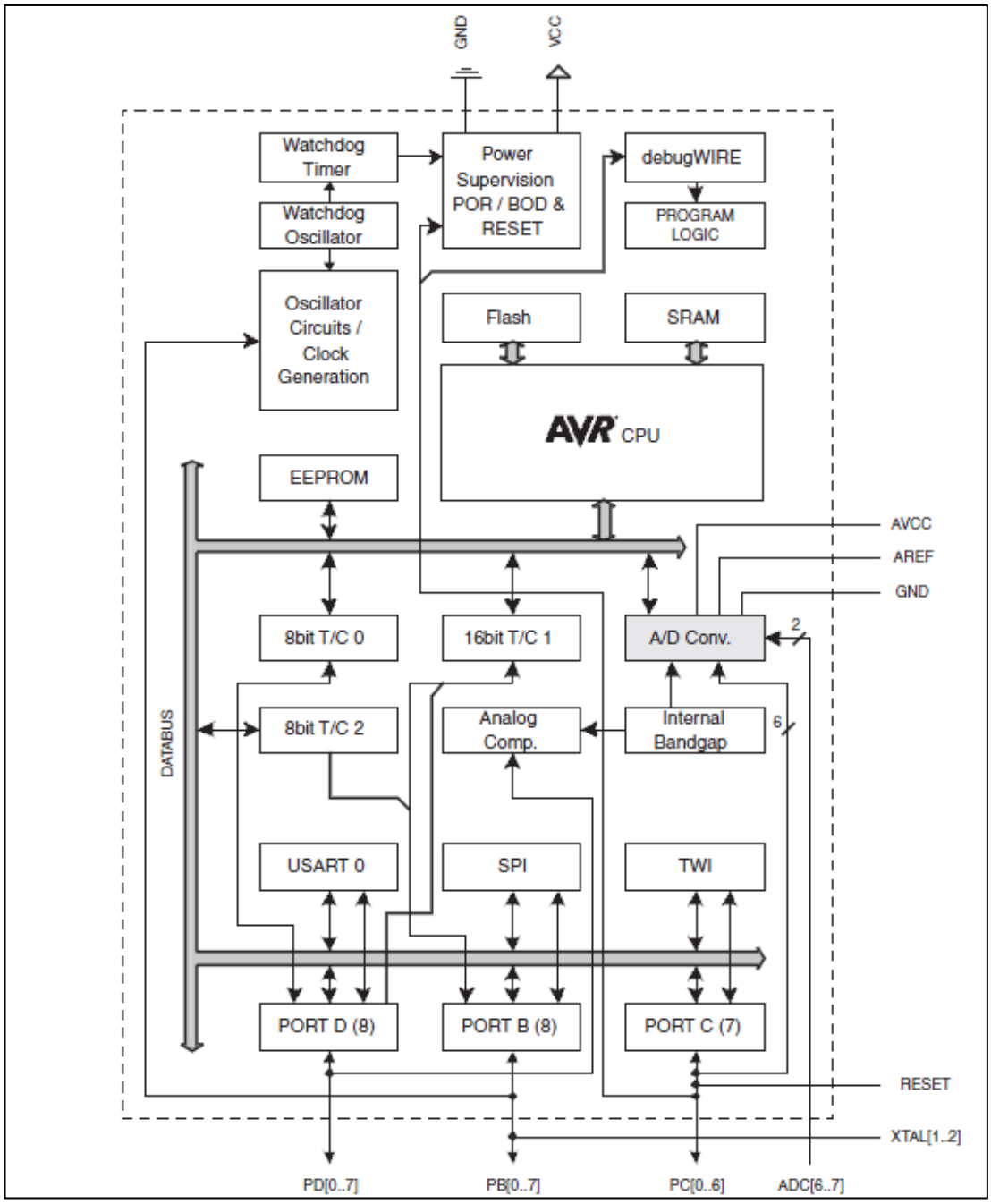


Fig. 5 – Diagrama de blocos de um microcontrolador ATMega168
Fonte: Datasheet do microcontrolador ATMega168

As diferenças entre microprocessador e microcontrolador, são fundamentais para a correcta compreensão de todos os subcapítulos que se seguem.

2.1.3. Tipos e Quantidades de Memória Disponíveis

Uma das principais diferenças em termos de capacidade nos diferentes modelos disponíveis reside na memória do microcontrolador utilizado. Vamos começar por analisar essa diferença, já que representa um factor crucial no seu desempenho.

As quantidades de memória disponíveis nos diversos modelos de microcontrolador utilizados pelo *Arduino* são os seguintes:

	ATMega168	ATMega328	ATMega1280
Flash	16 KBytes (2 KBytes Bootloader)	32 KBytes (2 KBytes Bootloader)	128 KBytes (4 KBytes Bootloader)
SRAM	1024 Bytes	2048 Bytes	8192 Bytes
EEPROM	512 Bytes	1024 Bytes	4096 Bytes

Tabela 1 – Quantidade de memória disponível em cada modelo de microcontrolador
Fonte: Retirado do datasheet dos respectivos microcontroladores

Como podemos constatar pela análise da tabela 1, o ATMega1280 leva clara vantagem em termos de capacidade. Uma das memórias mais "importantes", além da memória *Flash* que permite o armazenamento do *bootloader* e do programa – *sketch* - a ser executado, é a memória SRAM (Static Random Access Memory), que se comporta de forma semelhante à RAM nos nossos computadores. É na SRAM que o programa, ao ser executado, cria e modifica todo o tipo de variáveis necessárias à sua execução. Este tipo de memória apenas mantém os dados enquanto se encontra alimentada, o mesmo não acontece com a memória EEPROM (Electrically-Erasable Programmable Read-Only Memory) e memória Flash.

A memória Flash é nos dias de hoje comummente utilizada em cartões de memória, *pendrives*, memória de armazenamento em câmaras de vídeo, telemóveis, PDA (Personal Digital Assistant), entre muitas outras aplicações. Podemos resumir o tipo de memória, em função da sua utilização, da seguinte forma:

Flash	É aqui que o sketch e o bootloader são armazenados
SRAM	Onde são criadas e modificadas as variáveis ao longo da execução do *sketch*
EEPROM	Pode ser usada para guardar informação (não volátil)

Tabela 2 – Utilização dada pelo *Arduino* (utilizador) aos diversos tipos de memória
Fonte: Site oficial *Arduino* (www.arduino.cc)

Com vista a ser poupada memória SRAM, necessária para a correcta execução do programa, por vezes são guardadas constantes (p.ex. o número π) na memória Flash e EEPROM. Para tal, recorrem-se a bibliotecas próprias que possibilitam essa leitura e escrita, para a EEPROM pode-se recorrer à biblioteca "<EEPROM.h>" e para a memória Flash pode-se recorrer è biblioteca "<pgmspace.h>" ou "<Flash.h>.

Neste tutorial apenas vão ser abordadas a biblioteca "<EEPROM.h>" e "<Flash.h>", mas, devido ao carácter "*open source*", existem muitas outras bibliotecas disponíveis para as mesmas finalidades, devendo o leitor procurar trabalhar com a que se sente mais à vontade e obtém os melhores resultados. A abordagem a estas duas bibliotecas encontra-se no capítulo 4 – "**Recursos Avançados**".

2.1.4. *Pinout* Disponível

Torna-se importante uma descrição das possibilidades em termos de *pinout* do *Arduino*, sendo uma representação esquemática possível para o *Arduino Duemilinove* a seguinte:

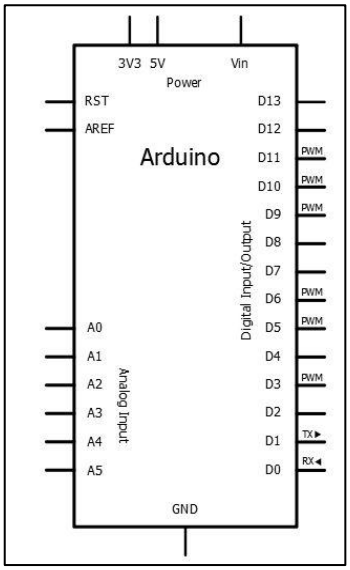


Fig. 6 – Representação esquemática do *Arduino Duemilinove*
Fonte: Obtido a partir Software open source *Fritzing*
(Software de download gratuito a partir do site - www.fritzing.org)

Pela análise da figura acima podemos retirar as seguintes conclusões, em termos de *pinout* disponível:

Pinos de I/O digitais	14 (6 com *Pulse Width Modulation* (PWM))
Pinos analógicos	6
Pinos de Ground (GND)	3
Pinos de 5 V	1
Pinos de 3.3 V	1
Pinos de Analog Reference (AREF)	1
Pinos de reset (RESET)	1

Tabela 2 – Pinout disponível (Arduino *Duemilinove)*

É ainda importante referir que a corrente máxima por cada pino analógico e digital é de 40 mA, à excepção da saída que providencia 3.3 V (visível na figura 1 e figura 5), que permite correntes máximas de 50 mA.

De acordo com Sousa and Lavinia, 2006, a capacidade de utilizar Pulse Width Modulation (PWM), é muito importante, pois permite obter uma tensão analógica a partir de um sinal digital, ou seja, de um sinal que apenas pode assumir o estado lógico 0 (0V) ou 1 (5 V). O conceito de PWM é utilizado para referir sinal que possua uma frequência constante e um "*duty cicle*" variável.

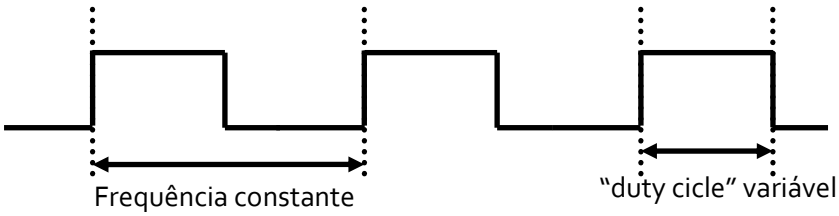


Fig. 7 – Exemplo de sinal PWM
Fonte: Adaptado de Sousa and Lavinia 2006, pág. 155

A teoria por detrás deste "fenómeno" pode ser facilmente descrita. Analisemos a seguinte equação:

$$V_{dc} = \frac{1}{T}\int_{0}^{T} V(t)dt$$

Nota: Na equação anterior, "T" representa o período e "V(t)" representa a tensão em função do tempo.

Aplicando o conceito de PWM à equação descrita anteriormente, obtemos o seguinte:

$$V(t) = \begin{cases} V_{pulso} \Rightarrow 0 \leq t \leq t_p \\ 0 \Rightarrow t_p < t \leq T \end{cases}$$

Nota: Na equação anterior "t_p" representa a duração do impulso, e "V_{pulso}" representa a tensão do impulso do sinal PWM.

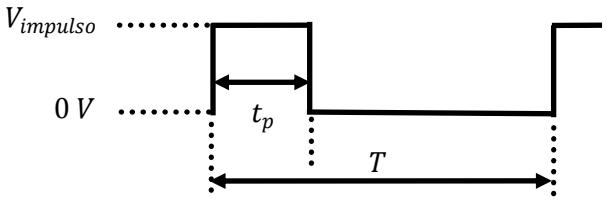


Fig. 8 – Representação de um impulso PWM
Fonte: Adaptado de Sousa and Lavinia 2006, pág. 156

Aplicando os conceitos descritos anteriormente:

$$V_{dc} = \left(\int_{0}^{t_p} V_{impulso}\ dt + \int_{t_p}^{T} 0\ dt \right)$$

Ficando pois:

$$V_{dc} = \frac{t_p}{T} * V_{impulso}$$

Podemos concluir, pela análise da equação anterior, que a tensão média ("V_{dc}") é directamente proporcional ao "*duty cicle*" do sinal PWM. Esta característica permite-nos fazer variar a tensão, neste caso específico, variar a tensão entre 0 e 5 V (*Arduino*).

Após uma breve descrição sobre o modo de funcionamento de um sinal PWM, vamos agora aprofundar um pouco mais a conversão A/D. Este factor é de grande interesse e a sua compreensão é fundamental para perceber os valores obtidos nos pinos analógicos. O microcontrolador utilizado possui um conversor analógico digital de 10 bits, fazendo as contas:

$$2^{10} = 1024$$

Como a tensão máxima de referência, por definição, se encontra nos 5V, correspondendo ao valor 1023, obtemos a seguinte resolução:

$$5 \div 1024 \cong 0{,}00488\ \text{V} \cong 5\ \text{mV}$$

O que significa que só se conseguirá "detectar" variações superiores a 5 mV, ou seja, o valor lido pelo *Arduino* só se altera a cada 5 mV de variação do sinal analógico de entrada.

Em caso de aplicações que possuam sensores analógicos, por vezes 5 mV não é uma resolução aceitável - existindo uma grande perda de resolução. Uma possível solução, sem recorrer a electrónica externa, é apresentada de seguida.

Para a resolução da questão, existe um pino de entrada denominado ***AREF***, que significa "***Analog Reference***". Este pino permite mudar a referência analógica do standard 5V para o valor de entrada. Ficando todas as entradas analógicas com a referência introduzida.

Simplificando, se for introduzido no pino ***AREF*** a tensão de 2V obtém-se a seguinte resolução:

$$2 \div 1024 \cong 1.953 \text{ mV} \cong 2 \text{ mV}$$

É importante ter em conta que todos os pinos analógicos ficam com esta referência, sendo necessária também a sua declaração por Software pela forma "***analogReference(tipo)"***. A referência analógica pode, assim, ser de três tipos:

DEFAULT	Mantém a referência por definição (5V)
INTERNAL	Altera a referência para 1.1 V (ATMega168)
EXTERNAL	A voltagem introduzida no pino **AREF** é tomada como referencia

Tabela 3 – Modos de configuração da referência analógica
Fonte: Site oficial Arduino (www.arduino.cc)

Basicamente, faz-se a configuração do conteúdo do registo **ADMUX** (página 205 e 206 do respectivo *datasheet*). É importante, por vezes, fazer uma interligação entre a função em C, uma linguagem de alto nível (com um nível de abstracção elevado, mais perto da "linguagem humana"), e o assembly, uma linguagem de baixo nível. É então muito importante tentar perceber o que acontece na configuração do microcontrolador e não apenas ver os resultados, embora sejam estes que nos interessam, não será isso que se pretende na compreensão do funcionamento do microcontrolador e, posteriormente, do *Arduino*.

É igualmente de referir que após configurar o *Arduino* para o uso do pino **AREF**, ele deixa de ter disponíveis os pinos de 3.3V e 5V, sendo estes desligados é necessário recorrer a alimentação externa, caso se queiram utilizar essas tensões de alimentação.

O *Arduino* possui capacidade de operar alimentado, quer pela porta USB ou por uma entrada **Pwr** (do tipo "Power Jack"), sendo recomendada a sua alimentação (**Pwr**) entre os 7 e os 12V, possibilita uma operação do tipo "**Standalone"**.

O pino $\mathbf{V_{in}}$ não foi referido juntamente com os outros tipos de alimentação, pois possui um comportamento duplo. Ou seja, pode servir para alimentar o *Arduino* na gama de valores especificada ou, caso se alimente o *Arduino* recorrendo a uma das alimentações especificadas anteriormente, pode-se ter disponível um pino com a tensão utilizada na entrada (p.ex. se for alimentado o *Arduino* recorrendo a uma alimentação externa de 9 V, no pino $\mathbf{V_{in}}$ estarão acessíveis os 9 V da alimentação).

Alguns pormenores de configuração de cariz mais avançado poderão ser encontrados no capítulo 4 – "**Recursos avançados".**

2.2. Software

2.2.1. Instalação e Utilização

Neste capítulo, vai ser efectuada uma breve descrição de como instalar e utilizar o *Software* de desenvolvimento *Arduino*.

O *Software* de desenvolvimento *Arduino* é bastante fácil e intuitivo de utilizar, não havendo qualquer nível de dificuldade. Foram estruturados passos de forma a simplificar a sua utilização e instalação.

1- O 1º passo consiste em efectuar o *download* do respectivo *software* de desenvolvimento, através do site oficial *Arduino* (www.arduino.cc).

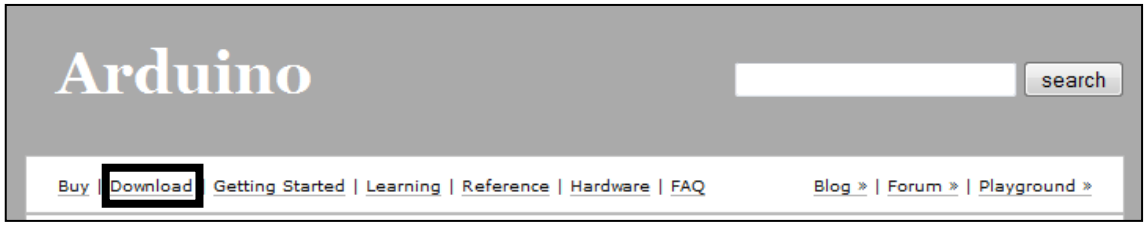


Fig. 9 – Imagem da parte superior do site oficial do *Arduino*
Fonte: Site oficial *Arduino* (www.arduino.cc)

A última versão disponível aparecerá na parte superior da página, como mostra a figura abaixo, sendo só necessário escolher a versão apropriada para o sistema operativo que nos encontramos a trabalhar. Actualmente, a versão mais actual é a 0016, mas quando estiver a ler este tutorial muito provavelmente já deve haver uma versão mais avançada.

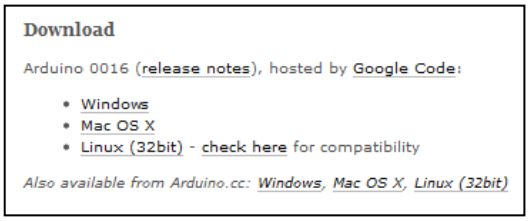


Fig. 10 – Imagem da parte superior do site oficial do *Arduino*.
Fonte: Site oficial *Arduino* (www.arduino.cc)

2- O 2º passo consiste em descompactar o ficheiro ".ZIP" (versão Windows w Mac OS) ou ".tgz" (versão Linux) para uma pasta à sua escolha. Escolha uma pasta de destino final, pois o programa não necessita de instalação. Utilizando o sistema operativo Windows, o conteúdo da pasta deverá ser o seguinte:

drivers	21-06-2009 21:18	Pasta de Ficheiros	
examples	02-07-2009 14:47	Pasta de Ficheiros	
hardware	02-07-2009 16:33	Pasta de Ficheiros	
java	02-07-2009 16:35	Pasta de Ficheiros	
lib	02-07-2009 16:35	Pasta de Ficheiros	
reference	02-07-2009 16:35	Pasta de Ficheiros	
sketchbook	21-06-2009 21:22	Pasta de Ficheiros	
arduino	30-05-2009 12:15	Aplicação	50 KB
cygiconv-2.dll	30-05-2009 12:15	Extensão da aplica...	947 KB
cygwin1.dll	30-05-2009 12:15	Extensão da aplica...	1.829 KB
ICE_JNIRegistry.dll	30-05-2009 12:15	Extensão da aplica...	64 KB
jikes	30-05-2009 12:15	Aplicação	2.568 KB
libusb0.dll	30-05-2009 12:15	Extensão da aplica...	43 KB
readme	30-05-2009 12:14	Documento de tex...	16 KB
run	30-05-2009 12:15	Ficheiro batch do ...	1 KB
rxtxSerial.dll	30-05-2009 12:15	Extensão da aplica...	76 KB

Fig. 11 – Conteúdo da pasta que contém o Software de desenvolvimento Arduino

3- O 3º passo consiste em ligar a placa de desenvolvimento ao computador e instalar os *drivers* **FTDI,** para permitir uma conversão de USB para série. A representação do *pinout* de um FTDI é a seguinte:

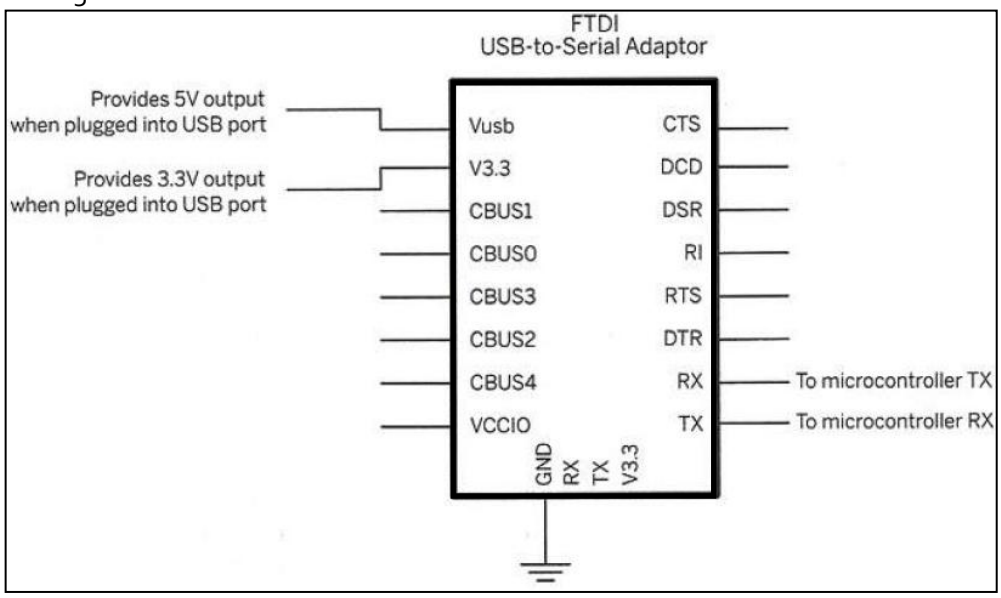


Fig. 12 – Esquema de um FTDI
Fonte: Retirado de Igoe 2007 pág. 55

Os *drivers* encontram-se disponíveis na pasta do *Software Arduino* - que descompactou no passo 2 - ou se preferir pode sempre retirar os *drivers* mais actualizados do site oficial **FTDI** - http://www.ftdichip.com/.

4- O 4º passo consiste em configurar a porta série a ser utilizada e qual o tipo de modelo *Arduino*, que nos encontramos a utilizar. Para tal, necessitamos de abrir o *Software* de desenvolvimento e escolher na barra de separadores a opção "Tools".

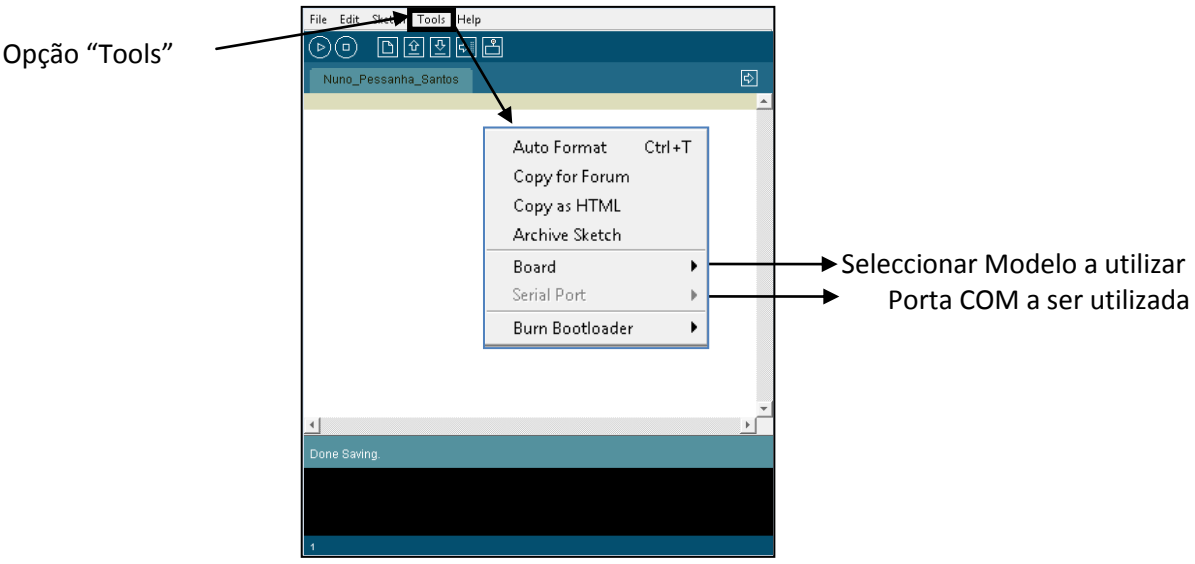


Fig. 13 – *Software* de desenvolvimento Arduino

5- O 5º e último passo para a utilização do *Software* consiste em elaborar o seu *Sketch*, compilar e, caso não tenha erros, fazer o *uploading* para a placa de desenvolvimento *Arduino*.

Fig.14 – Botão para compilar o *Sketch* elaborado

Fig.15 – Botão para efectuar *uploading*

2.2.2. Ciclo de Desenvolvimento

Um resumo do referido anteriormente pode ser encontrado na figura seguinte, que demonstra os passos necessários para elaborar uma aplicação (após ter feito a instalação do Software) de uma forma esquemática e a qual se pode designar por Ciclo de Desenvolvimento.

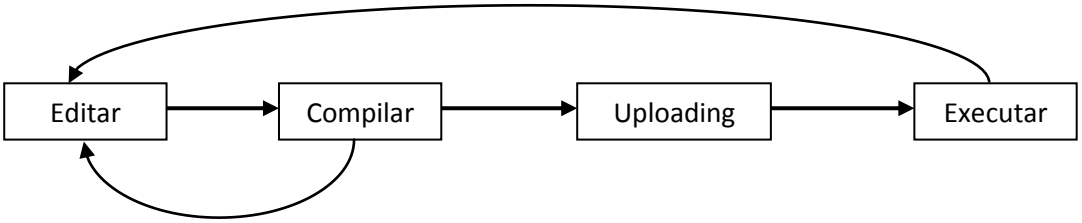


Fig.16 – Esquema da fase de desenvolvimento de uma aplicação
(Ciclo de desenvolvimento)

A análise da figura 16 permite fazer um resumo possível de todas as fases necessárias até à execução do programa criado, sendo muito importante a sua compreensão e apreensão. A figura 17 visa dar outra perspectiva ao exposto na figura 16, tentando dar a compreender uma vez mais qual o ciclo de desenvolvimento da plataforma de desenvolvimento *Arduino*.

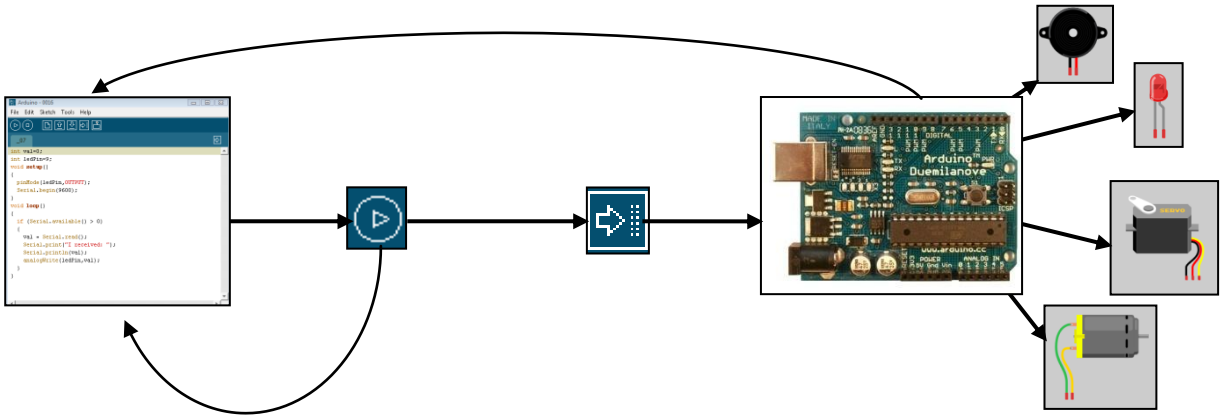


Fig.17 – Esquema da fase de desenvolvimento de uma aplicação
(Outra maneira possível de representar)

2.2.3. Interacção do *Arduino* com outro *Software*

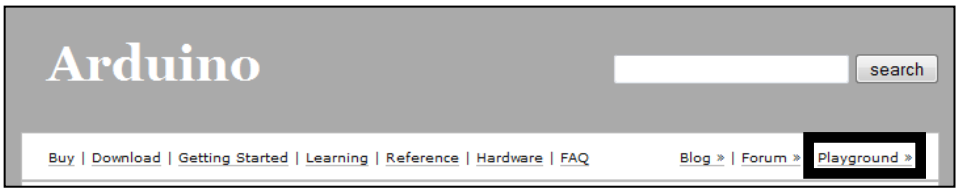


Fig. 18 – Imagem da parte superior do site oficial do *Arduino*
Fonte: Site oficial *Arduino* (www.arduino.cc)

Caso se pretenda interagir com outro tipo de *Software* com a plataforma de desenvolvimento *Arduino*, pode-se recorrer ao separador "Playground" (Fig. 18). Podendo encontrar inúmeras referências de interacção do *Arduino* com outro *Software* de desenvolvimento (p.ex. Processing, Mathematica, MatLab, entre outros).

3. Índice de Instruções

Neste capítulo, vão ser referidas as mais importantes instruções base do ambiente de desenvolvimento *Arduino,* sem recorrer a bibliotecas externas, sendo estas enumeradas por tipo de aplicação. Antes de abordar algumas das instruções possíveis numa linguagem de alto nível, podemos fazer uma breve abordagem para analisar qual a classificação a atribuir ao microcontrolador em estudo.

De acordo com Arroz, Monteiro et al. 2007, pág. 732, podemos classificar os processadores tendo em conta o seu conjunto de instruções em duas categorias:

- **CISC** (Complex Instruction Set Computers)
- **RISC** (Reduced Instruction Set Computers)

Baseando-se a arquitectura **CISC** num conjunto de instruções com modos de endereçamento bastante complexos, ao utilizar este tipo de arquitectura é permitido elaborar programas bastante compactos e codificáveis ao recorrer a um uso reduzido de instruções. Alguns dos inconvenientes deste tipo de arquitectura são baseados no elevado tempo de ciclos de relógio, necessários para executar cada uma das suas instruções. Outro inconveniente é a existência de variados modos de endereçamento.

Por outro lado os processadores baseados em arquitecturas do tipo **RISC**, apresentam um conjunto de instruções bastante reduzido, possibilitando assim obter uma enorme simplicidade e um tempo de execução menor por instrução que a arquitectura **CISC**. Conseguindo, assim, com um conjunto de instruções básicas com tempo de execução menores, obter, no final, velocidades de processamento mais elevadas que a arquitectura **CISC**.

Na sua versão mais simples, baseando a análise em Arroz, Monteiro et al. 2007 pág. 733, pode afirmar-se que os processadores **RISC** possuem os seguintes tipos de instruções:

- Instruções lógicas e aritméticas sobre registos;
- Instruções de transferência de dados entre memória e registos;
- Instruções de controlo.

No *datasheet* dos microcontroladores utilizados nos modelos do *Arduino*, pode-se perceber que estamos perante microcontroladores baseados numa arquitectura "*advanced* **RISC**". Segundo Tecnology 1992 pág. 14, em que é efectuado uma revisão completa à arquitectura "advanced RISC", é referido que esta arquitectura foi desenvolvida para ser uma especificação "standard" para uma família de microprocessadores baseada na família **MIPS**.

MIPS (Microprocessor Without Interlocked Pipeline Stages) é uma arquitectura de microcontroladores **RISC**, desenvolvida pela "MIPS Technology". **MIPS** também podem ser as iniciais de "**M**illios of **I**nstructions **P**er **S**econd", ou seja, "**M**ilhares de **I**nstruções **P**or **S**egundo.

3.1. Funções Base

3.1.1 void setup()

Descrição: Esta função apenas é executada uma vez e é normalmente utilizada para executar a inicialização de variáveis, a inicialização da utilização bibliotecas (não confundir com declaração de bibliotecas), a definição dos pinos (como *input* ou *output*), o início do uso de comunicação série, entre outros. Esta função apenas volta a ser executada novamente ao ser efectuado o *reset* ou quando se desligar e volta a ligar a placa de desenvolvimento *Arduino*.

Exemplo:

```
(1) int botao=3; // Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3".
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(botao,INPUT); // Permite definir o pino 3 como "INPUT"
(5) }
(6) void loop(){
(7) (.....)
(8) }
```

3.1.2 void loop()

Descrição: Esta função faz um "**loop**" sucessivo (como o próprio nome indica), ou seja, todos os comandos existentes no interior desta função são sucessivamente repetidos, o que pode permitir a leitura sucessiva de portas, a leitura sucessiva de parâmetros provenientes de sensores externos e actuar de acordo com as condições estabelecidas, entre muitas outras aplicações, bastando apenas ter criatividade.

Exemplo:

```
(1) int botao=3,val; // Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val"
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(botao,INPUT); // Permite definir o pino 3 como "INPUT"
(5) pinMode(13,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(6) }
(7) void loop(){
(8) val=analogRead(botao); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(9) if(val>=500){ // Ciclo if que testa a condição – "val>=500" (maior ou igual a 500)
```

(10) digitalWrite(13,HIGH); *// Se a condição for verificada, é atribuído ao pino digital 13 a condição "HIGH" (5 V)*
(11) }
(12) }

Nota: As funções atrás descritas têm de ser necessariamente do tipo "**void**", ou seja, não podem retornar qualquer valor depois da sua execução.

3.1.3. Resumindo

As funções "void setup()"e "void loop()" são de carácter obrigatório, ou seja, mesmo que não necessária a sua utilização deverão constar no código utilizado. E apenas serão chamadas funções externas que constem na função "void loop()".
Um resumo dos subcapítulos anteriores encontra-se expresso no exemplo seguinte:

Exemplo:

Declaração de variáveis globais;

```
void setup(){
Instrução 1;
Instrução 2;
(....)
}
```

Conjunto de instruções apenas executado uma vez, na inicialização do programa a executar

```
void loop(){
Instrução 6;
Instrução 9;
Função1();
(....)
}
```

Conjunto de instruções que é executado em "loop"

Nos próximos capítulos e subcapítulos, vai ser explorado o tipo de variáveis possíveis de declarar e algumas das instruções julgadas mais importantes. Essas instruções estarão divididas dependendo da sua aplicação, facilitando assim a sua organização.

3.2. Ciclos

3.2.1 Ciclo If…….else

Descrição: É utilizado para descrever uma condição, ou seja, quando uma variável for: igual, maior, menor ou diferente (de acordo com a declaração efectuada) que um determinado valor é executada uma determinada condição. Podem-se usar vários **ciclos if….else** encadeados, de forma a verificar diversas condições.

Sintaxe:

```
if(condição){
Instrução 1;
Instrução 2;
(.....)
}
else{
Instrução 3;
Instrução 4;
(.....)
}
```

A utilização da instrução **else** no ciclo **if** é dispensável, podendo apenas tomar a forma:

```
if(condição){
Instrução 1;
Instrução 2;
(.....)
}
```

A condição acima referida pode ser descrita das seguintes formas:

X == Y	X igual a Y
X != Y	X diferente de Y (não igual)
X > Y	X maior que Y
X >= Y	X maior ou igual a Y
X < Y	X menor que Y
X <= Y	X menor ou igual a Y

Tabela 4 – Condições possíveis

Exemplo:

```
(1) int botao=3,val; // Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val"
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(botao,INPUT); // Permite definir o pino 3 como "INPUT"
```

```
(5) pinMode(13,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(6) }
(7) void loop(){
(8) val=analogRead(botao); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(9) if(val>=500) // Ciclo if que testa a condição – "val>=500" (maior ou igual a 500)
(10) digitalWrite(13,HIGH); // Se a condição for verificada, é atribuído ao pino digital 13 a condição "HIGH" (5 V)
(12) else
(13) digitalWrite(13,LOW); // Se a condição não for verificada, é atribuído ao pino digital 13 a condição "LOW" (0 V)
(14) }
```

3.2.2 Ciclo for

Descrição: Esta instrução é utilizada quando se quer efectuar uma instrução ou conjunto de instruções um determinado número de vezes. Um exemplo possível de aplicação é o preenchimento de um vector (*array*), com dados provenientes de entradas analógicas.

Sintaxe:

```
for(inicialização; condição; incremento a efectuar){
Instrução 1;
Instrução 2;
(....)
}
```

A inicialização é apenas efectuada uma vez, no início do ciclo. Cada vez que o ciclo é efectuado a condição é testada. Caso a condição se verifique é efectuado o incremento, caso a condição não se verifique o ciclo termina a sua execução. A condição, referida anteriormente, é declarada à semelhança da condição utilizada no ciclo **if**. O exemplo abaixo mostra um exemplo de implementação do ciclo **for**, face às "regras" expostas acima.

Exemplo:

```
(1) int entrada_analogica=3,val,f[10],i; // Declaração de uma variável do tipo integer, com o nome "entrada_analogica" inicializada com o valor "3", de variáveis do mesmo tipo com o nome "val" e "i" não inicializada, e de um vector "f[]" com 11 posições de memória do mesmo tipo (integer).
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(entrada_analogica,INPUT); // Permite definir o pino 3 como "INPUT"
(5) }
(6) void loop(){
(7) for(i=0;i<=10;i++){ // Ciclo for que é percorrido 11 vezes, "i<=10" com i a começar no valor "0"
(8) val=analogRead(entrada_analogica); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
```

(9) f[i]=val; *//Atribui o valor da variável "val" ao vector "f[i]", ou seja, permite preencher o vector com 11 valores do pino analógico 3*
(10) }
(11) }

3.2.3 Ciclo switch / case

Descrição: É normalmente utilizada para declarar uma lista de casos possíveis, para uma determinada variável, sendo verificado cada caso e é apenas executado quando a variável respeitar a condição declarada.

Sintaxe:

switch(variável){
case 1:
Instrução a executar quando variável for 1 (variável == 1)
break;
case 2:
Instrução a executar quando variável for 2 (variável == 2)
break;
(.....)
default:
Conjunto de instruções a executar se nenhuma das condições for verificada. A utilização desta condição é opcional.
break;
}

É necessário utilizar a instrução "*break*" caso se pretenda sair do ciclo após a condição ser verificada. Caso contrário, vai continuar a ser executado o ciclo a partir da condição verificada.

Exemplo:

(1) int entrada_analogica=3,val; *// Declaração de uma variável do tipo integer, com o nome "entrada_analogica" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val"*
(2) void setup() {
(3) Serial.begin(9600); *// Permite a inicialização da comunicação Série*
(4) pinMode(entrada_analogica,INPUT); *// Permite definir o pino 3 como "INPUT"*
(5) pinMode(13,OUTPUT); *// Permite definir o pino 13 como "OUTPUT"*
(6) }
(7) void loop(){
(8) val=analogRead(entrada_analogica); *// Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"*
(9) switch(val){ *// Ciclo switch, sendo "val" a variável a "testar"*
(10)case 500: *// condição de "teste", se val == 500 (igual a 500)*
(11) digitalWrite(13,HIGH); *// Se a condição da linha 10 se verificar, esta instrução é executada*
(12) break; *// Instrução que permite terminar o ciclo*
(13) default: *// Se nenhuma das condições se verificar (o seu uso é opcional)*

(14) digitalWrite(13,LOW); *// Instrução que é executada se a condição do ciclo switch não se verificar*
(15) }
(16) }

3.2.4 Ciclo while

Descrição: É normalmente utilizado para efectuar um "*loop*" sucessivo até uma determinada condição se verificar. Caso essa condição se deixe de verificar o ciclo termina. Em casos em que a condição não se verifica, estamos perante um ciclo infinito (semelhante à função "void loop()").

Sintaxe:

```
while(condição){
Instrução 1;
Instrução 2;
(.....)
}
```

É necessário fazer com que a condição se torne falsa, caso se queira finalizar o ciclo para executar outras instruções. Pois caso contrário o ciclo vai ser executado indefinidamente.

Exemplo:

(1) int entrada_analogica=3,val,i=0,f[10]; *// Declaração de uma variável do tipo integer, com o nome "entrada_analogica" inicializada com o valor "3", de variáveis do mesmo tipo com o nome "val" não inicializada, "i" inicializada com o valor "0" e de um vector "f[]" com 11 posições de memória do mesmo tipo (integer).*
(2) void setup() {
(3) Serial.begin(9600); *// Permite a inicialização da comunicação Série*
(4) pinMode(entrada_analogica,INPUT); *// Permite definir o pino 3 como "INPUT"*
(5) }
(6) void loop(){
(7) while(i<=10){ *// ciclo while, com a condição "i<=10"*
(8) val=analogRead(entrada_analogica); *// Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"*
(9) f[i]=val; *//Atribui o valor da variável "val" ao vector "f[i]", ou seja, permite preencher o vector com valores provenientes do pino analógico 3*
(10) i++; *// Incremento do valor de "i" inicialmente a zero, instrução semelhante a "i=i+1"*
(11) }
(12) }

3.2.5 Ciclo do….while

Descrição: Bastante semelhante ao ciclo "*while*", sendo a principal diferença o facto da condição ser testada apenas no fim do ciclo e não no início do ciclo. Ou seja, mesmo que a condição seja falsa o ciclo vai ser sempre executado uma vez.

Sintaxe:

```
do{
Instrução 1;
Instrução 2;
(.....)
}
while(condição);
```

Exemplo:

```
(1) int entrada_analogica=3,val,i=0,f[10]; // Declaração de uma variável do tipo integer, com o nome "entrada_analogica" inicializada com o valor "3", de uma variável do mesmo tipo com o nome "val" não inicializada, "i" inicializada com o valor "0" e de um vector "f[]" com 11 posições de memória do mesmo tipo (integer).
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(entrada_analogica,INPUT); // Permite definir o pino 3 como "INPUT"
(5) }
(6) void loop(){
(7) do{ // Permite definir o inicio do ciclo
(8) val=analogRead(entrada_analogica); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(9) f[i]=val: //Atribui o valor da variável "val" ao vector "f[i]", ou seja, permite preencher o vector com valores provenientes do pino analógico 3
(10) i++; // Incremento do valor de "i" inicialmente a zero, instrução semelhante a "i=i+1"
(11)}
(12) while(i<=10); // Se a condição se verificar repete o ciclo, caso não se verifique o ciclo termina
(13) }
```

3.3. Tipos de variáveis disponíveis

3.3.1 Variável do Tipo B*oolean*

Descrição: Uma variável deste tipo apenas pode tomar dois valores distintos - "true" ou "false". Esta variável reserva 1 Byte de memória para a sua utilização.

Sintaxe:

boolean variável = "valor";

Quando é referido acima a "valor", está-se a referir a "true" ou "false".

Exemplo:

```
(1) int entrada_analogica=3,val,i=0,f[10]; // Declaração de uma variável do tipo integer, com o nome "entrada_analogica" inicializada com o valor "3", de uma variável do mesmo tipo com o nome "val" não inicializada, "i" inicializada com o valor "0" e de um vector "f[]" com 11 posições de memória do mesmo tipo (integer).
(2) boolean teste = false; // Definição de uma variável do tipo boolean inicializada com "false"
(3) void setup() {
(4) Serial.begin(9600); // Permite a inicialização da comunicação Série
(5) pinMode(entrada_analogica,INPUT); // Permite definir o pino 3 como "INPUT"
(6) }
(7) void loop(){
(8) do{ // Permite definir o inicio do ciclo
(9) val=analogRead(entrada_analogica); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(10) f[i]=val: //Atribui o valor da variável "val" ao vector "f[i]", ou seja, permite preencher o vector com valores provenientes do pino analógico 3
(11) i++; // Incremento do valor de "i" inicialmente a zero, instrução semelhante a "i=i+1"
(12) teste = !teste; //modifica o valo atribuído à variável booleana "teste" o seu oposto ( se "true" fica "false" e vice-versa)
(13)}
(14) while(i<=10); // Se a condição se verificar repete o ciclo, caso não se verifique o ciclo termina
(15) }
```

3.3.2. Variável do tipo *char* vs *unsigned char*

Descrição: Uma variável do tipo ***char*** apenas pode guardar um único caracter, ocupando este tipo de variável 1 Byte de memória. Este tipo de variável pode armazenar os valores em decimal, correspondentes ao valor decimal (tabela ASCII) do caracter a guardar. Podendo ter valores em decimal no intervalo de -128 a 127. Ao declarar uma variável do tipo **char** como **unsigned**, estamos a abdicar da parte negativa. Podendo armazenar valores correspondendo entre 0 a 255.

Sintaxe:

```
char variável = 'Caracter':
char variável = "valor em decimal - tabela ASCII";
unsigned char variável = 'Caracter';
unsigned char variável = "valor em decimal - tabela ASCII";
```

Exemplos:

```
char exemplo_char = 'N'; // Atribuido à variável do tipo char o caracter "N"
char exemplo_char_1 = 78; // Corresponde ao caracter "N", segundo a tabela ASCII
```

3.3.3. Variável do tipo byte

Descrição: Armazena um número do tipo ***unsigned***, compreendido entre 0 e 255. Ocupando, como o próprio nome indica, 1 Byte na memória (8 Bits).

Sintaxe:

```
byte variável = valor;
```

Exemplos:

```
byte exemplo = B0000001; // Armazena o valor em binário que corresponde ao valor "1" em decimal
byte exemplo_2 = 1; // Semelhante ao exemplo anterior, mas armazenando o valor directamente em decimal
```

3.3.4. Variável do tipo *int* vs *unsigned int*

Descrição: Este tipo de variável (integer) permite guardar um valor inteiro de 2 bytes, ou seja, um número inteiro com valor compreendido entre -32768 e 32767. Podemos também declarar a variável ***int***, como sendo do tipo ***unsigned***. Ao declarar uma variável do tipo ***unsigned***, deixamos de ter parte negativa passando a poder armazenar valores compreendidos entre 0 e 65535.

Sintaxe:

```
int variável = valor;
unsigned int variável = valor;
```

Exemplos:

```
int exemplo = -150; // Declara um variável do tipo "int" e atribui-lhe o valor de "-150"
unsigned int exemplo_1 = 65000; // Declara um variável do tipo "unsigned int" e atribui-lhe o valor de "65000"
```

3.3.5. Variável do tipo *long* vs *unsigned long*

Descrição: É um tipo de variável que pode guarda valores numéricos até 32 bits, o que correspondente a 4 bytes, ou seja, valores compreendidos entre -2147483648 e 2147483647. Este tipo de variável pode também ser declarado como *unsigned*, podendo armazenar valores compreendidos entre 0 e 4294967295.

Sintaxe:

```
long variável = valor;
unsigned long variável = valor;
```

Exemplos:

```
long exemplo = -1500000000; // Declara um variável do tipo "long" e atribui-lhe o valor de "-1500000000"
unsigned long exemplo_1 = 4000000000; // Declara um variável do tipo "unsigned long" e atribui-lhe o valor de "4000000000"
```

3.3.6. Variável do tipo *float* vs *double*

Descrição: A variável do tipo ***float*** apresenta uma maior resolução, face às variáveis do tipo integer. São reservados em memória 4 Bytes (32 bits), para armazenar o seu conteúdo. Este tipo de variável pode conter valores no intervalo de $-3.4028235 * 10^{38}$ e $3.4028235 * 10^{38}$. Paralelamente o conceito de ***double*** leva-nos a considerar uma variável que possua o dobro da precisão, de uma variável do tipo ***float***, apesar de no contexto de estudo a declaração de uma variável do tipo ***float*** ou do tipo ***double*** ser igual. Ou seja, não existe um incremento de precisão mantendo o mesmo espaço reservado de memória para ambos os tipos utilizando o *Arduino*.

Sintaxe:

```
float variável = valor;
double variável = valor;
```

Exemplos:

```
float exemplo = 1.589; // Declara um variável do tipo "float" e atribui-lhe o valor de "1.589"
double exemplo_1 = 1.589; // Declara um variável do tipo "double" e atribui-lhe o valor de "1.589"
```

3.3.7. Variável do Tipo *array* e a Noção de *string*

Descrição: É considerado um *array*, um vector de variáveis do mesmo tipo ao qual se pode aceder através do seu respectivo índice. O conceito de *string* é comummente utilizado para designar um vector de variáveis do tipo ***char***.

Sintaxe:

```
tipo_variável nome_variável[índice] = valor;
```

Ao declarar um índice de valor 10, estamos na verdade a reservar 11 espaços na memória para a variável do tipo declarado. Pois, também, temos de contar com o índice zero, este factor torna-se muitas vezes objecto de erro.

Exemplos:

```
float exemplo[10]; // Declaração de um vector com 11 "espaços", do tipo "float"
float exemplo_2[]={1,2,3,4,5,6,7,8,9,10}; // Declaração de um vector do tipo" float", que vai ter um índice compreendido entre 0 e 9
char exemplo_3[11]="Hello World"; // Declaração de uma string
char exemplo_4[11]={'H','E','L','L','O',' ','W','O','R','L','D'}; // Declaração de uma string, com o conteúdo semelhante ao exemplo anterior, mas com outra forma de colocar o mesmo conteúdo
```

3.4. Converter tipos de variáveis

3.4.1. char(x)

Descrição: Converte um valor de x, que pode ser de qualquer tipo, para uma variável do tipo ***char***.

Sintaxe:

```
char variável = char(valor);
```

3.4.2. byte(x)

Descrição: Converte um valor de x, que pode ser de qualquer tipo, para uma variável do tipo **byte**.

Sintaxe:

```
byte variável = byte(valor);
```

3.4.3. int(x)

Descrição: Converte um valor de x, que pode ser de qualquer tipo, para uma variável do tipo **integer**.

Sintaxe:

```
int variável = int(valor);
```

3.4.4. long(x)

Descrição: Converte um valor de x, que pode ser de qualquer tipo, para uma variável do tipo ***long***.

Sintaxe:

```
long variável = long(valor);
```

3.4.5. float(x)

Descrição: Converte um valor de x, que pode ser de qualquer tipo, para uma variável do tipo ***float***.

Sintaxe:

```
float variável = float(valor);
```

3.5. Funções

3.5.1. *Input/Output* digital

3.5.1.1. pinMode()

Descrição: Ao recorrer a esta instrução, é possível configurar o modo de comportamento de um determinado pino. Possibilitando assim defini-lo como ***input*** ou ***output***, esta definição normalmente é efectuada recorrendo à função "void setup()".

Sintaxe:

pinMode(Número do pino, Modo);

O "Modo" acima descrito pode ser definido da seguinte forma:

- "**INPUT**"
- "**OUTPUT**"

Exemplo:

```
(1) int botao=3,val; // Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val" não inicializada com nenhum valor
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(botao,INPUT); // Permite definir o pino 3 como "INPUT"
(5) pinMode(13,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(6) }
(7) void loop(){
(8) val=analogRead(botao); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(9) if(val>=500) // Ciclo if que testa a condição – "val>=500" (maior ou igual a 500)
(10) digitalWrite(13,HIGH); // Se a condição for verificada, é atribuído ao pino digital 13 a condição "HIGH" (5 V)
(11) }
(12) }
```

3.5.1.2 digitalWrite()

Descrição: Possibilita, nos pinos configurados como output através da instrução "**pinMode**", estabelecer a saída dos respectivos pinos com o valor lógico 1 (HIGH – 5 V) ou com o valor lógico 0 (LOW – 0V)

Sintaxe:

digitalWrite(Número do pino, Modo);

O "Modo" acima descrito, pode ser definido como:

- "**HIGH**"
- "**LOW**"

Exemplo:

```
(1) int botao=3,val; // Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val" não inicializada com nenhum valor
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(botao,INPUT); // Permite definir o pino 3 como "INPUT"
(5) pinMode(13,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(6) }
(7) void loop(){
(8) val=analogRead(botao); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(9) if(val>=500) // Ciclo if que testa a condição – "val>=500" (maior ou igual a 500)
(10) digitalWrite(13,HIGH); // Se a condição for verificada, é atribuído ao pino digital 13 a condição "HIGH" (5 V)
(11) }
(12) }
```

3.5.1.3. digitalRead()

Descrição: Possibilita a leitura de uma entrada digital específica, retornando um valor no formato integer (int). Se obtivermos um valor de retorno de "1", estamos perante uma leitura do tipo "HIGH" (valor lógico 1). Se tal não se verificar, e tivermos um valor de retorno igual a "0", estamos perante uma leitura do tipo "LOW" (valor lógico 0).

Sintaxe:

Variável do tipo integer = digitalRead(Número do pino);

Exemplo:

(1) int botao=3,val; *// Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val" não inicializada com nenhum valor*
(2) void setup() {
(3) Serial.begin(9600); *// Permite a inicialização da comunicação Série*
(4) pinMode(botao,INPUT); *// Permite definir o pino 3 como "INPUT"*
(5) pinMode(13,OUTPUT); *// Permite definir o pino 13 como "OUTPUT"*
(6) }
(7) void loop(){
(8) val=digitalRead(botao); *// Permite a leitura do valor do pino 3 digital atribuindo o seu valor à variável "val"*
(9) if(val==1) *// Ciclo if que testa a condição – "val>=500" (maior ou igual a 500)*
(10) digitalWrite(13,HIGH); *// Se a condição for verificada, é atribuído ao pino digital 13 a condição "HIGH" (5 V)*
(11) }
(12) else{ *// Caso a condição do ciclo if não se verifique*
(13) digitalWrite(13,LOW); *// Se a condição não for verificada, é atribuído ao pino digital 13 a condição "LOW" (0 V)*
(14) }
(15) }

3.5.2. *Input/Output* analógico

3.5.2.1 analogRead()

Descrição: Possibilita a leitura do valor analógico do pino especificado, com um conversor A/D possuindo uma resolução de 10 bits. O que leva a que um valor compreendido entre 0 e 5 V, esteja compreendido entre os valores inteiros (int) 0 e 1023.

Sintaxe:

Variável do tipo integer = analogRead(Número do pino);

Exemplo:

```
(1) int botao=3,val; // Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val" não inicializada com nenhum valor
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(botao,INPUT); // Permite definir o pino 3 como "INPUT"
(5) pinMode(13,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(6) }
(7) void loop(){
(8) val=analogRead(botao); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(9) if(val>=500) // Ciclo if que testa a condição – "val>=500" (maior ou igual a 500)
(10) digitalWrite(13,HIGH); // Se a condição for verificada, é atribuído ao pino digital 13 a condição "HIGH" (5 V)
(11) }
(12) }
```

3.5.2.2 analogWrite()

Descrição: Possibilita a utilização dos pinos PWM (Pulse Width Modulation) da placa *Arduino*. O sinal PWM mantém-se até ser modificado através de uma outra instrução que afecte esse pino, a frequência do sinal PWM criado é de $\cong 490\ Hz$.

Sintaxe:

analogWrite(Número do pino, valor);

O "valor" referido anteriormente varia entre 0 (sempre desligado), até ao valor 255 (que representa um sinal de 5 V constante).

Exemplo:

```
(1) int botao=3,val; // Declaração de uma variável do tipo integer, com o nome "botao" inicializada com o valor "3" e de uma variável do mesmo tipo com o nome "val" não inicializada com nenhum valor
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(botao,INPUT); // Permite definir o pino 3 como "INPUT"
(5) pinMode(13,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(6) }
(7) void loop(){
(8) val=analogRead(botao); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(9) if(val>=500) // Ciclo if que testa a condição – "val>=500" (maior ou igual a 500)
(10) analogWrite(9,255); // Instrução com a mesma função que "digitalWrite(9,HIGH)"
(11) }
(12) else{
(13) analogWrite(9,0); // Instrução com a mesma função que "digitalWrite(9,LOW)"
(14) }
(15) }
```

3.5.3. Tempo

3.5.3.1. millis()

Descrição: Possibilita o retorno da quantidade de tempo, em milisegundos na forma de uma variável do tipo "*unsigned long*". O valor retornado representa o tempo que, passou desde que o programa actual começou a ser executado. O *overflow* (voltar ao início, valor zero) do contador ocorre passado um tempo de ≅ 50 dias.

Sintaxe:

unsigned long tempo = millis();

Exemplo:

(1) unsigned long tempo; *// Declarada uma variável do tipo unsigned long com o nome "tempo"*
(2) void setup() {
(3) (.....)
(4) }
(5) void loop(){
(6) tempo = millis(); *// Atribui à variável "tempo" o valor em milisegundos desde que o sketch actual começou a ser executado*
(7) }

3.5.3.2. micros()

Descrição: Possibilita o retorno da quantidade de tempo, em microsegundos na forma de uma variável do tipo "unsigned long". O valor retornado representa o tempo que, passou desde que o programa actual começou a ser executado. O overflow (voltar ao início, valor zero) do contador ocorre passado um tempo de ≅ 70 minutos.

Sintaxe:

unsigned long tempo = micros();

Exemplo:

(1) unsigned long tempo; *// Declarada uma variável do tipo unsigned long com o nome "tempo"*
(2) void setup() {
(3) (.....)
(4) }
(5) void loop(){
(6) tempo = micros(); *// Atribui à variável "tempo" o valor em microsegundos desde que o sketch actual começou a ser executado*
(7) }

3.5.3.3. *delay* (milisegundos)

Descrição: Possibilita efectuar uma pausa ao programa em execução, por uma quantidade de milisegundos especificada. Útil para manter um estado durante uma certa quantidade de tempo.

Sintaxe:

delay(tempo que deseja efectuar a pausa – ms);

Exemplo:

```
(1) int led_pin=13; // Declaração de uma variável do tipo integer com o nome "led_pin", sendo-lhe atribuída o valor "13"
(2) void setup() {
(4) pinMode(led_pin,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(5) Serial.begin(9600); // Permite a inicialização da comunicação Série
(6) }
(7) void loop(){
(8) digitalWrite(led_pin,HIGH); // É atribuído ao pino digital 13 a condição "HIGH" (5 V)
(9) delay(200); // É efectuado um delay de 200 ms, antes de efectuar a próxima instrução
(10) digitalWrite(led_pin,LOW); // É atribuído ao pino digital 13 a condição "LOW" (0 V)
(11) delay(600); // É efectuado um delay de 600 ms, antes de efectuar a próxima instrução, neste caso a função "void loop()" recomeça
(12) }
```

3.5.3.4. delayMicroseconds (microsegundos)

Descrição: Possibilita efectuar uma pausa ao programa em execução, por uma quantidade de microsegundos especificada.

Sintaxe:

delayMicroseconds(tempo que deseja efectuar a pausa – μs);

Exemplo:

```
(1) int led_pin=13; // Declaração de uma variável do tipo integer com o nome "led_pin", sendo-lhe atribuída o valor "13"
(2) void setup() {
(4) pinMode(led_pin,OUTPUT); // Permite definir o pino 13 como "OUTPUT"
(5) Serial.begin(9600); // Permite a inicialização da comunicação Série
(6) }
(7) void loop(){
(8) digitalWrite(led_pin,HIGH); // É atribuído ao pino digital 13 a condição "HIGH" (5 V)
```

```
(9) delayMicroseconds(200); // É efectuado um delay de 200 µs, antes de efectuar a próxima instrução
(10) digitalWrite(led_pin,LOW); // É atribuído ao pino digital 13 a condição "LOW" (0 V)
(11) delayMicroseconds(600); // É efectuado um delay de 600 µs, antes de efectuar a próxima instrução, neste caso a função "void loop()" recomeça
(12) }
```

3.5.4. Funções Matemáticas

3.5.4.1. min(valor1,valor2)

Descrição: Instrução que retorna o menor de dois valores.

Sintaxe:

Variável menor = min(Valor 1, Valor 2);

Exemplo:

int variavel_menor=0; *// declaração de uma variável do tipo integer de nome "variavel_menor" inicializada com o valor 0*
variavel_menor = min(5,9); *// O valor atribuído à variável "variável_menor" seria o valor "5"*

3.5.4.2. max(valor1,valor2)

Descrição: Instrução que retorna o maior de dois valores.

Sintaxe:

Variável maior = max(Valor 1, Valor 2);

Exemplo:

int variavel_maior=0; *// declaração de uma variável do tipo integer de nome "variavel_maior" inicializada com o valor 0*
variavel_maior = max(5,9); *// O valor atribuído à variável "variável_maior" seria o valor "9"*

3.5.4.3. abs(valor)

Descrição: Instrução que retorna o modulo de um número.

Sintaxe:

Módulo = abs(valor);

Sendo "Módulo" um valor que tem de respeitar a seguinte condição:

$$\text{Módulo} = \begin{cases} \text{valor} \Rightarrow \text{valor} \geq 0 \\ -\text{valor} \Rightarrow \text{valor} < 0 \end{cases}$$

Exemplo:

int modulo=0;
modulo = abs(-5) *// O valor atribuído à variável "modulo" seria o valor "5"*

3.5.4.4. constrain(valor, valor1.,valor2)

Descrição: Limita o valor de retorno a um intervalo definido pelo utilizador.

Sintaxe:

Valor = constrain(Valor a testar, Valor inferior do intervalo, Valor superior do intervalo);

Sendo "Valor" dado pela seguinte condição:

$$\text{Valor} = \begin{cases} \text{valor} \Rightarrow \text{Se valor} \in [\text{valor1}, \text{valor2}] \\ \text{valor1} \Rightarrow \text{valor} < valor1 \\ \text{valor2} \Rightarrow \text{valor} > valor2 \end{cases}$$

Exemplo:

int valor=0; *// declaração de uma variável do tipo integer de nome "valor" inicializada com o valor 0*
valor = constrain(5,2,10) *// O valor atribuído à variável "valor" seria o valor "5"*

3.5.4.5. map(X,valor1,valor2,para valor1,para valor2)

Descrição: Instrução útil para modificar o intervalo de valores esperados por uma determinada aplicação. Possibilitando assim um ajuste na escala de valores a utilizar. Este tipo de instrução é muito utilizado para gerar sinais PWM através de sinais analógicos de entrada, podendo assim a partir de um sinal que varia de 0 a 1023 gerar um sinal PWM que varia de 0 a 255 com relativa facilidade.

Sintaxe:

Valor num novo intervalo = map(Valor, valor1, valor2, Para valor1, Para valor2);

"valor1" – O limite inferior do intervalo actual
"valor2" – O limite superior do intervalo actual
"Para valor1" – O limite inferior do novo intervalo a considerar
"Para valor2" – O limite superior do novo intervalo a considerar

Exemplo:

```
1) int entrada_analogica=3,val,i=0,f[10]; // Declaração de uma variável do tipo integer, com o nome "entrada_analogica" inicializada com o valor "3", de variáveis do mesmo tipo com o nome "val" não inicializada com nenhum valor, "i" inicializada com o valor "0" e de um vector "f[]" com 11 posições de memória do mesmo tipo (integer)
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) pinMode(entrada_analogica,INPUT); // Permite definir o pino 3 como "INPUT"
(5) }
(6) void loop(){
(7) for(i=0;i<=10;i++){ //Inicio do ciclo for
(8) val=analogRead(entrada_analogica); // Permite a leitura analógica do valor do pino 3 atribuindo o seu valor à variável "val"
(10) map(val,0,1023,0,500); // Passa o valor da variável "val" que se encontra numa escala de valores entre 0 e 1023 (entrada analógica), para um valor entre 0 e 500
(11) f[i]=val;
(12) }
(13) }
```

3.5.4.6. pow(valor, expoente)

Descrição: Permite calcular o resultado ao elevar um número a um determinado expoente.

Sintaxe:

Valor obtido = pow(valor, expoente);

Exemplo:

```
int valor=0;
valor = pow(2,2) // O valor atribuído à variável "valor" seria o valor "4", pois 2^2 = 4.
```

3.5.4.7. sqrt(valor)

Descrição: Permite o cálculo da raiz quadrado de um determinado valor.

Sintaxe:

Valor obtido = sqrt(valor);

Exemplo:

```
int valor=0;
valor = sqrt(9) // O valor atribuído à variável "valor" seria o valor "3", pois √9 = 3.
```

3.5.5. Funções Trigonométricas

3.5.5.1. sen(valor)

Descrição: Instrução que retorna o cálculo do seno, de um valor em radianos.

Sintaxe:

Resultado = sen(Valor em Radianos);

Com "Resultado ϵ $[-1; 1]$

3.5.5.2. cos(valor)

Descrição: Instrução que retorna o cálculo do coseno, de um valor em radianos.

Sintaxe:

Resultado = cos(Valor em Radianos);

Com "Resultado ϵ $[-1; 1]$

3.5.5.1. tan(valor)

Descrição: Instrução que retorna o cálculo da tangente, de um valor em radianos.

Sintaxe:

Resultado = tan(Valor em Radianos);

Com "Resultado ϵ $[-\infty; +\infty]$

3.5.6. Números Aleatórios

3.5.6.1 randomSeed(valor)

Descrição: Possibilita variar o ponto onde o algoritmo de sequência aleatória começa, pois apesar de este ser muito longo (possuir muitos valores no seu intervalo), o seu intervalo apresenta-se constante. Ao fazer variar o ponto onde o algoritmo começa, menos probabilidades temos de repetir um número.

Sintaxe:

randomSeed(Valor);

Exemplo:

(1) void setup() {
(2) (……)
(3) randomSeed(analogRead(0)); *//Possibilita fazer variar o ponto onde o algoritmo de sequência aleatória começa. Se colocar-mos uma entrada aleatória, cada vez que a função "void setup()" é executada o valor atribuído é diferente.*
(4) (……)
(5) }
(6) void loop(){
(7) (…..)
(8) }

3.5.6.2. random()

Descrição: Instrução que permite gerar números aleatórios (ou melhor "escrevendo" - pseudo-aleatórios).

Sintaxe:

Resultado = random(Valor máximo possível);
Resultado = random(Valor mínimo possível, Valor máximo possível);

O "Valor máximo possível" não está incluído no intervalo a considerar e a inclusão do "Valor mínimo possível" é opcional. Ficando a variável "Resultado" compreendida no seguinte intervalo:

"Resultado" ϵ [Valor mínimo possível; $(Valor\ máximo\ possível - 1)$].

Exemplo:

(1) long valor_random; *// Declaração de uma variável do tipo long de nome "valor_random"*
(2) void setup() {
(3) (……)

```
(4) randomSeed(analogRead(0)); //Possibilita fazer variar o ponto onde o algoritmo de sequência aleatória começa. Se colocar-mos uma entrada aleatória, cada vez que a função "void setup()" é executada o valor atribuído é diferente.
(5) (.....)
(6) }
(7) void loop(){
(8) valor_random=random(10,20); // Valor de "valor_random" será um valor compreendido entre o valor "10" e o valor "20"
(9) (....)
(10) }
```

3.5.7. Interrupts

3.5.7.1. attachInterrupt(interrupt, função,modo)

Descrição: Instrução que permite definir o funcionamento de interrupts externos. Ocorrendo a definição através desta instrução da função a executar, do pino onde vai ocorrer o interrupt e o modo como se deve responder à variação do sinal de entrada no pino a considerar.

Sintaxe:

attachInterrupt(interrupt, função a executar, modo);

O parâmetro "interrupt" pode tomar os seguintes valores:

- *Arduino duemilinove*
  - Valor "0" -corresponde ao pino digital 2;
  - Valor "1" -corresponde ao pino digital 3.
- *Arduino Mega*
  - Valor "0" -corresponde ao pino digital 2;
  - Valor "1" -corresponde ao pino digital 3;
  - Valor "2" -corresponde ao pino digital 21;
  - Valor "3" -corresponde ao pino digital 20;
  - Valor "4" -corresponde ao pino digital 19;
  - Valor "5" -corresponde ao pino digital 18.

Nota: A "função" a executar, ao ser verificado o interrupt externo, não poderá retornar qualquer valor (Ou seja, deverá ser do tipo "void").

O parâmetro "modo" define o modo como ocorre o "reconhecimento", de que ocorreu um interrupt externo. Podendo ter os seguintes valores:

LOW	O interrupt é efectuado sempre que o valor lógico no pino é 0.
CHANGE	O interrupt é efectuado sempre que o valor no pino muda de valor.
RISING	O interrupt é efectuado sempre que o valor lógico no pino muda de 0 para 1.
FALLING	O interrupt é efectuado sempre que o valor lógico no pino muda de 1 para 0.

Tabela 5 – Condições possíveis do parâmetro "modo"

3.5.7.2. detachInterrupt(interrupt)

Descrição: Possibilita desabilitar um interrupt previamente estabelecido utilizando a instrução "attachInterupt()".

Sintaxe:

detachInterrupt(interrupt);

Os valores possíveis para "interrupt" encontram-se enumerados no subcapítulo anterior (3.5.7.1.).

3.5.7.3. interrupts()

Descrição: Instrução que permite voltar a activar o uso de interrupções, depois de esta funcionalidade ter sido desactivada. O uso de interrupts encontra-se por defeito activado.

Sintaxe:

interrupts();

Exemplo:

```
(1) void setup() {
(2) (.....)
(3) }
(4) void loop(){
(5) noInterrupts(); // Instrução que desactiva o uso de interrupts
(6) Instrução 1;
(7) (.....)
(8) interrupts(); // Instrução que activa o uso de interrupts
(9) (.....)
(10) }
```

3.5.7.4. noInterrupts()

Descrição: Instrução que permite desactivar o uso de interrupções. O uso de interrupts encontra-se por defeito activado.

Sintaxe:

noInterrupts();

Exemplo:

```
(1) void setup() {
(2) (.....)
```

```
(3) }
(4) void loop(){
(5) noInterrupts(); // Instrução que desactiva o uso de interrupts
(6) Instrução 1;
(7) (.....)
(8) interrupts(); // Instrução que activa o uso de interrupts
(9) (.....)
(10) }
```

3.5.8. Comunicação Série

3.5.8.1. Serial.available()

Descrição: Obtém o número de bytes que estão a ser recebidos por série, podendo assim saber quando estão a ser recebidos dados ou não.

Sintaxe:

Número de bytes a receber (int) = Serial.available();

Exemplo:

```
(1) int leitura_serie=0; //Declaração de uma variável do tipo integer com o nome "leitura_serie", inicializada com o valor "0"
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) }
(5) void loop(){
(6) if(Serial.available()>0) // Condição if que verifica se estão a ser recebidos dados por Série
(7) leitura_serie=Serial.read(); // Caso se estejam a receber dados por série, o seu valor é guardado na variável integer "leitura_serie"
(8) }
(9) }
```

3.5.8.2. Serial.begin(int baud rate)

Descrição: Instrução necessária para iniciar a comunicação série, permitindo definir qual a "baud rate" da comunicação. Os valores de velocidade de comunicação mais comuns para comunicação com um computador são: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 e 115200 (No entanto podem definir-se outros valores).

Sintaxe:

Serial.begin(int baud rate);

Exemplo:

```
(1) int leitura_serie=0; //Declaração de uma variável do tipo integer com o nome "leitura_serie", inicializada com o valor "0"
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) }
(5) void loop(){
(6) if(Serial.available()>0) // Condição if que verifica se estão a ser recebidos dados por Série
(7) leitura_serie=Serial.read(); // Caso se estejam a receber dados por série, o seu valor é guardado na variável integer "leitura_serie"
(8) }
(9) }
```

3.5.8.3. Serial.read()

Descrição: Permite a leitura dos dados recebidos por série. Esta função retorna o valor "-1" se não houver dados disponíveis.

Sintaxe:

Valor recebido (int) = Serial.read();

Exemplo:

```
(1) int leitura_serie=0; ; //Declaração de uma variável do tipo integer com o nome "leitura_serie", inicializada com o valor "0"
(2) void setup() {
(3) Serial.begin(9600); // Permite a inicialização da comunicação Série
(4) }
(5) void loop(){
(6) if(Serial.available()>0) ) // Condição if que verifica se estão a ser recebidos dados por Série
(7) leitura_serie=Serial.read(); // Caso se estejam a receber dados por série, o seu valor é guardado na variável integer "leitura_serie"
(8) }
(9) }
```

3.5.8.4. Serial.flush()

Descrição: Efectua o "*flush*" (apaga) de todos os dados presentes no buffer de entrada no momento de execução da instrução. O buffer de entrada tem uma capacidade de armazenamento de 64 bytes.

Sintaxe:

Serial.flush();

Exemplo:

```
(1) void setup() {
(2) Serial.begin(9600); // Permite a inicialização da comunicação Série
(3) }
(4) void loop(){
(5) Serial.flush(); //Apaga o conteúdo do buffer de entrada
(6) (.....)
(7) }
```

3.5.8.5. Serial.print() vs Serial.println()

Descrição: Instrução que permite o envio de dados pela porta série. A única diferença entre "Serial.print" e "Serial.println()" é que a segunda instrução adiciona ao dado enviado o caracter "\r" ("carriage return") e o caracter "\n" ("new line").

Sintaxe:

Serial.print(dado a enviar);
Serial.print(dado a enviar, formato);

O formato de envio pode ser decimal (DEC), hexadecimal (HEX), octal (OCT) ou binário (BIN). O dado a enviar também pode tomar o formato de uma string.

Exemplo:

```
(1) void setup() {
(2) Serial.begin(9600); // Permite a inicialização da comunicação Série
(3) }
(4) void loop(){
(5) Serial.flush();//Apaga o conteúdo do buffer de entrada
(6) Serial.println("CADETES AEL"); //Envio da string "CADETES AEL" seguido do caracter "\r" e "\n"
(7) }
```

3.5.8.1. Serial.write()

Descrição: Permite enviar dados em binário pela porta série. Esses dados podem ser enviados como um único byte ou um conjunto de bytes.

Sintaxe:

Serial.write(valor);
Serial.write(string);
Serial.write(vector, comprimento);

A variável "valor" acima define o valor a ser enviado em binário pela porta série, o mesmo pode ser efectuado enviando uma "string" como um conjunto de bytes. Na última instrução apresentada acima, é enviado um vector ("vector") de tamanho "comprimento".

4. Recursos Avançados

Este capítulo tem como objectivo explorar algumas bibliotecas existentes para a plataforma de desenvolvimento *Arduino*, possibilitando assim expandir as suas possibilidades. As bibliotecas a ser expostas neste capítulo são as mais usadas nos projectos de desenvolvimento, mas não representam a totalidade das bibliotecas existentes. Será também aprofundado neste capítulo a temática "taxa de amostragem" no nosso contexto de estudo, tentando que a sua compreensão se torne maior.

4.1. Flash

A quantidade de memória SRAM disponível é normalmente muito mais pequena do que a memória flash disponível e em programas em que é necessário recorrer à utilização de muitas variáveis facilmente a memória disponível se torna uma limitação. Uma solução óbvia, tendo em conta o que foi referido anteriormente, seria gravar o valor de algumas variáveis em *flash*. Mas existe uma limitação, na medida que a gravação de dados na memória flash apenas pode ser efectuada quando o programa se encontra a ser carregado (durante o *uploading*) para a memória *flash*. O que leva a que apenas os valores de variáveis constantes devam ser armazenados na memória flash, sendo utilizadas só quando necessário (não se encontrando já a ocupar espaço em SRAM).

Para utilizar esta funcionalidade vais ser utilizada a biblioteca "<Flash.h>", devendo a mesma ser declarada no nosso *sketch*, esta biblioteca não se encontra disponível no software de desenvolvimento devendo o seu *download* ser efectuado do site "Arduiniana" (http://arduiniana.org/libraries/flash/). Esta biblioteca baseia-se na biblioteca "<avr/progmen.h>", tentado simplificar assim o seu uso.

4.1.1. Sintaxe a Utilizar para Guardar Dados em *Flash*

FLASH_STRING(Nome da string, String a guardar);

FLASH_ARRAY(Tipo de variável a ser guardada, Nome do vector, Valores a colocar no vector);
FLASH_TABLE(Tipo de variáveis a guardar na tabela, nome da tabela, número de colunas, valores a armazenar na tabela);
FLASH_STRING_ARRAY(Nome do vector de strings, PSTR("string a armazenar"), PSTR("String a armazenar 2"), PSTR(.....));

Qualquer dúvida em relação ao tipo de variáveis possíveis de utilizar, pode consultar o subcapítulo - "**Tipos de variáveis disponíveis**" (Página 23 à 26).

Nota: A biblioteca "<Flash.h>" é baseada na biblioteca "<avr/progmen.h>", como esta não suporta a criação de um vector de *strings*, o que esta biblioteca faz é criar um vector com os endereços guardados em SRAM de *strings* individuais guardadas em *Flash*. Isto faz com que o uso desta instrução ocupe 2 bytes por cada *string* armazenada.

4.1.2. Sintaxe a Utilizar para Ler Dados da Memória *Flash*

Nome usado (String, vector,tabela, vector de strings)[indice];

O acesso aos dados guardados em Flash é em tudo semelhante à utilização de vectores (*arrays*). O que é um dos pontos fortes da utilização desta biblioteca.

Existe mais sintaxe possível por parte desta biblioteca, mas sem dúvida foi aqui apresentado o mais importante (Guardar e ler). Para mais informações pode ser consultado o site oficial desta biblioteca, já referido anteriormente.

4.2. EEPROM

O microcontrolador disponível na placa de desenvolvimento *Arduino Duemilinove* (ATMega168) possui 512 bytes de memória EEPROM, segundo o *datasheet* do respectivo microcontrolador.

Este tipo de memória tem a vantagem de manter a sua informação armazenada, mesmo após desligarmos a sua alimentação, visto que pode ser de grande importância conseguir manter certos dados, dependo da aplicação a implementar. Uma outra vantagem é a possibilidade de guardar conteúdo nesta memória enquanto executamos o nosso *sketch*, o que não acontece com o armazenamento em *flash*. Esta biblioteca já se encontra disponível no *Software* de desenvolvimento *Arduino*, não necessitando de ser adicionada.

A principal limitação da memória EEPROM está na existência de um limite de ciclos de leitura/escrita. O que pode ser uma limitação a curto prazo em aplicações que recorram muito a este tipo de armazenamento (este tipo de informação encontra-se discriminada no datasheet do microcontrolador). O tempo de escrita num ATMega168 é de $\cong 3.3$ ms (informação retirada do seu datasheet).

4.2.1. Sintaxe de Escrita em EEPROM

EEPROM.write(Endereço, Valor a guardar);

Com "Endereço"(int) $\epsilon[0,511]$ e "Valor a guardar"(byte) $\epsilon[0,255]$.

4.2.2. Sintaxe de Leitura em EEPROM

EEPROM.read(Endereço);

Esta instrução, e colocando em "Endereço" o mesmo valor que na instrução "EEPROM.write", permite obter o valor guardado em memória.

Exemplo:

```
(1) #include<EEPROM.h> //Declaração da biblioteca <EEPROM.h>
(2)(.....)
(3) EEPROM.write(1,1); // Escrita no endereço 1 do valor inteiro 1
(4) void setup(){
(5) Instrução 1;
(6) (.....)
(7) }
(8) void loop(){
(9) int i=0; //Declaração de uma variável do tipo integer de nome "i", inicializada com o valor "0"
(10) i = EEPROM.read(1); //Feita a leitura do endereço 1 e guardado o seu conteúdo na variável de nome "i"
(11) (.....)
(12) }
```

4.3. Servomotor

Um servomotor é um pequeno dispositivo cujo veio pode ser posicionado numa determinada posição angular de acordo com um sinal de entrada. Enquanto esse sinal se mantiver constante e for enviado para o servomotor, o servo irá manter a sua posição angular. Ao variar o sinal de entrada possibilita uma variação a posição angular do veio. Os servomotores são muito usados no controlo de aviões telecomandados, robots, barcos telecomandados, helicópteros telecomandados, entre outras possíveis aplicações.

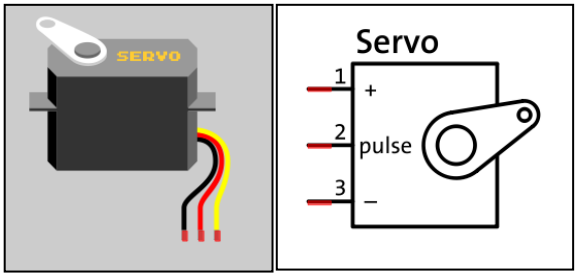


Fig. 19 – Representação de um servomotor
Fonte: Obtido a partir do Software open source *Fritzing*

Um servomotor, como se pode ver na figura 19, possui três entradas. Uma entrada é a alimentação (normalmente 5V para servomotores standard), outra será a massa e a última que falta referir será o sinal de entrada. Este sinal de entrada será um impulso PWM, em que fazendo variar o seu "*duty cicle*", podemos variar a posição angular do servomotor. Isto em servomotores de posição, pois também existem servomotores de rotação contínua. Nesses servomotores a variação do "*duty cicle*" fará variar não a posição angular, mas a velocidade e sentido de rotação do servomotor. Existe possibilidade de modificar, sem grande dificuldade, um servomotor de posição para rotação contínua. Esta modificação é permanente, não podendo ser reversível. No entanto, o objectivo deste subcapítulo não se centra nisso, mas sim em como fazer a sua interacção com a plataforma de desenvolvimento *Arduino*.

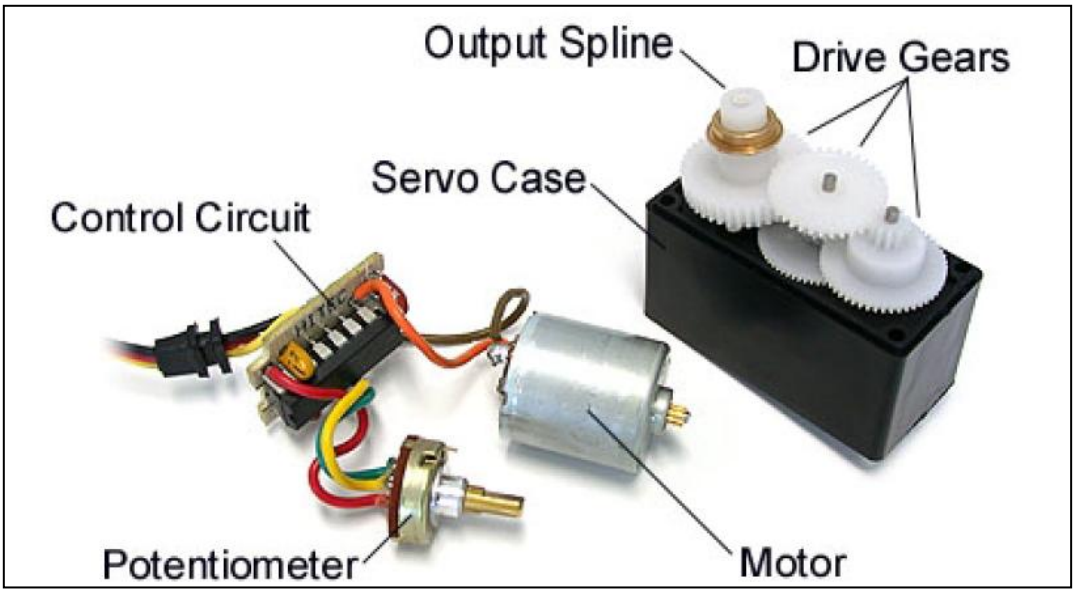


Fig. 20 – Representação de um servomotor de posição mais pormenorizada
Fonte: Retirado de Santos 2007, pág. 3

Pela análise da figura 20, podemos constatar os componentes mais importantes de um servomotor de posição são os seguintes:

- Circuito de controlo (Control Circuit)
- Potenciómetro (Potentiometer)
- Motor DC (Motor)

O circuito de controlo, como o próprio nome indica, é responsável por controlar e mover o motor DC, com base nas informações obtidas pela leitura do potenciómetro e do sinal de entrada. O potenciómetro, neste contexto, é utilizado acoplado ao eixo de rotação para conseguir obter a posição angular do eixo.

A enorme aplicabilidade dos servomotores leva a que, tendo em conta o universo da electrónica, seja uma temática obrigatória. Dessa necessidade surge a criação de uma biblioteca específica para facilitar a operação com servomotores, que até já se encontra disponível no *Software* de desenvolvimento *Arduino*.

A biblioteca "Servo.h" utiliza os pinos 9 e 10 para fazer o controlo dos respectivos servomotores, ao utilizar esta biblioteca (ao associar um servomotor a umas das duas saídas) não poderá utilizar os pinos 9 e 10 para outras aplicações. De seguida, vai ser explorada a sintaxe possível para o facilitar o uso desta biblioteca.

4.3.1. Sintaxe das Instruções da Biblioteca <Servo.h>

4.3.1.1. attach()

Descrição: Permite atribuir a um pino (pinos disponíveis – 9 ou 10), uma variável do tipo "Servo". A atribuição de uma variável deste tipo vai ser necessária para controlar o respectivo servomotor.

Sintaxe:

Servo.attach(Pino);

Servo.attach(Pino, Valor do ângulo mínimo, Valor do ângulo máximo);

- "***Servo***" – Variável do tipo "Servo" pode ser declarada da seguinte forma:

Servo nome_a_atribuir;

Podendo atribuir qualquer nome à escolha, o exemplo abaixo é esclarecedor em relação a esta questão.

- "Valor do ângulo mínimo" – Permite definir qual a largura mínima do impulso a utilizar, em microsegundos, que vai corresponder ao ângulo mínimo (0°).
- "Valor do ângulo máximo" – Permite definir qual a largura máxima do impulso a utilizar, em microsegundos, que vai corresponder ao ângulo máximo (180°).

Exemplo:

```
(1) #include<Servo.h> // Declaração da biblioteca <Servo.h>
(2) Servo Exemplo; // Criação de uma variável do tipo Servo, com o nome de "Exemplo"
(3) void setup(){
(4) Exemplo.attach(10); //Atribuição da variável do tipo Servo "Exemplo", ao pino digital 10
(5) (.....)
```

(6) void loop(){
(7) (.....)
(8) }

4.3.1.2. detach()

Descrição: Permite eliminar a ligação entre uma variável do "*Servo*" a um dos pinos possíveis de utilizar pela biblioteca "<Servo.h>" (Pinos 9 e 10). Se nenhum dos pinos possíveis estiver atribuído a uma variável do tipo "*Servo*", os pinos 9 e 10 podem ser utilizados normalmente.

Sintaxe:

Servo.detach();

- ***Servo"*** – Variável do tipo Servo pode ser declarada da seguinte forma:

Servo nome_a_atribuir;

Exemplo:

(1) #include<Servo.h> *// Declaração da biblioteca <Servo.h>*
(2) Servo Exemplo; *// Criação de uma variável do tipo Servo, com o nome de "Exemplo"*
(3) void setup(){
(4) Exemplo.attach(10); *// Atribuição da variável do tipo Servo "Exemplo", ao pino digital 10*
(5) (.....)
(6) void loop(){
(7) Exemplo.detach(); *// Elimina a ligação entre a variável do tipo Servo "Exemplo"e pino digital 10*
(8) analogWrite(10, 255); *// Como nenhuma variável do tipo Servo se encontra atribuída, podem-se usar as funcionalidades PWM dos pinos 9 e 10*
(9) }

4.3.1.3. write()

Descrição: Permite movimentar o eixo de um servomotor de posição para o ângulo pretendido. No caso de um servomotor de rotação contínua o valor 0 corresponderá à velocidade de rotação máxima num sentido, o valor 180 a velocidade de rotação máxima noutro sentido e o valor 90 será o ponto em que o servomotor se encontrará parado.

Sintaxe:

Servo.write(Ângulo);

- ***Servo"*** – Variável do tipo "Servo" pode ser declarada da seguinte forma:

Servo nome_a_atribuir;

- "Ângulo" $\epsilon[0,180]$

Exemplo:

(1) #include<Servo.h> *// Declaração da biblioteca <Servo.h>*
(2) Servo Exemplo; *// Criação de uma variável do tipo Servo, com o nome de "Exemplo"*

```
(3) Servo Exemplo_2; // Criação de uma variável do tipo Servo, com o nome de "Exemplo_2"
(4) void setup(){
(5) Exemplo.attach(10); // Atribuição da variável do tipo Servo "Exemplo", ao pino digital 10
(6) Exemplo_2.attach(9); // Atribuição da variável do tipo Servo "Exemplo_2", ao pino digital 9
(7) (.....)
(8) void loop(){
(9) Exemplo.write(0); // Faz com que um servomotor p.ex. de posição se movimente para a posição correspondente ao ângulo "0"
(1) Exemplo_2.write(90); // Faz com que um servomotor p.ex. de rotação contínua pare a sua rotação
(11) }
```

4.3.1.4. read()

Descrição: Instrução que retorna o valor do último ângulo utilizado, recorrendo à instrução "*Servo*.write(Ângulo)".

Sintaxe:

Ângulo (int) = ***Servo***.read();

- ***Servo"*** – Variável do tipo Servo pode ser declarada da seguinte forma:

Servo nome_a_atribuir;

Exemplo:

```
(1) #include<Servo.h> // Declaração da biblioteca <Servo.h>
(2) int angulo = 0; // Declaração de uma variável do tipo integer de nome "angulo", atribuindo-lhe o valor de "0"
(3) Servo Exemplo; // Criação de uma variável do tipo Servo, com o nome de "Exemplo"
(4) void setup(){
(5) Exemplo.attach(10); // Atribuição da variável do tipo Servo "Exemplo", ao pino digital 10
(6) (.....)
(7) void loop(){
(8) Exemplo.write(angulo); // Faz com que um servomotor p.ex. de posição se movimente para a posição correspondente ao valor da variável "angulo"
(9) angulo = (Exemplo.read() + 1); // Vai incrementado uma unidade ao último valor de ângulo utilizado recorrendo à instrução "Exemplo.write(angulo)"
(10) }
```

4.3.1.5. attached()

Descrição: Instrução que permite verificar se uma variável do tipo "*Servo*" se encontra atribuída a um pino específico.

Sintaxe:

Estado (int) = ***Servo***.attached();

- ***Servo"*** – Variável do tipo "Servo" pode ser declarada da seguinte forma:

Servo nome_a_atribuir;

- "Estado" $\epsilon \{0,1\}$, ou seja, só pode tomar o valor "true" (valor "1") ou "false" (valor "0").

Exemplo:

```
(1) #include<Servo.h> // Declaração da biblioteca <Servo.h>
(2) int atribuido=0; // Declaração de uma variável do tipo integer de nome "atribuido", atribuindo-lhe o valor de "0"
(3) Servo Exemplo; // Criação de uma variável do tipo Servo, com o nome de "Exemplo"
(4) void setup(){
(5) Exemplo.attach(10); // Atribuição da variável do tipo Servo "Exemplo", ao pino digital 10
(6) (.....)
(7) void loop(){
(8) atribuido = Exemplo.attached(); // Atribui o valor 1 à variável atribuído caso a variável do tipo Servo "Exemplo" se encontra associada a algum pino, caso contrário retorna o valor 0
(9) if (atribuido ==1){ //Se a variável estiver do tipo Servo "Exemplo" estiver atribuída a condição é verificada
(10) Exemplo.write(180); // Faz com que um servomotor p.ex. de posição se movimente para a posição correspondente ao valor "180"
(11) }
(12) (.....)
(13) }
```

O uso desta biblioteca simplifica a operação com servomotores, o que não implica que a movimentação dos servomotores não seja feita com recurso directamente à instrução "analogWrite()", ou seja, gerar os sinais PWM directamente. Tal é, obviamente, possível mas mais trabalhoso, existindo esta biblioteca para simplificar esse trabalho de implementação.

Com este subcapítulo espera-se dar noções básicas de funcionamento e uso de servomotores, estando a partir daqui apenas dependente da criatividade de cada um na implementação e aprofundamento do conhecimento sobre esta temática.

4.4. Software Serial

A necessidade de utilização de outros pinos para efectuar a Tx e a Rx que não os pinos digitais 0 e 1,levou à criação desta biblioteca. Através do uso desta biblioteca existe um expandir das capacidades do *Arduino* em termos de possibilidades de comunicação, podendo assim comunicar e receber em mais que um pino.

A comunicação "*standard*", por assim dizer, efectua a sua comunicação através de uma peça de *Hardware* denominada **UART** (universal asynchronous receiver/transmitter). Esta peça permite implementar uma comunicação série assíncrona.

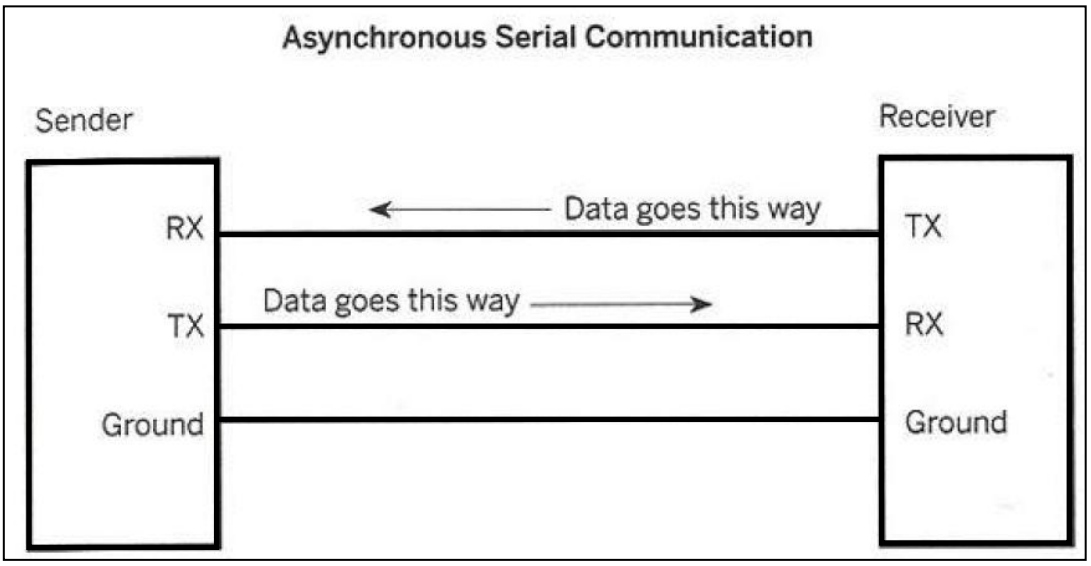


Fig. 21 – Representação de uma comunicação série assíncrona
Fonte: Retirado de Igoe 2007 pág. 51

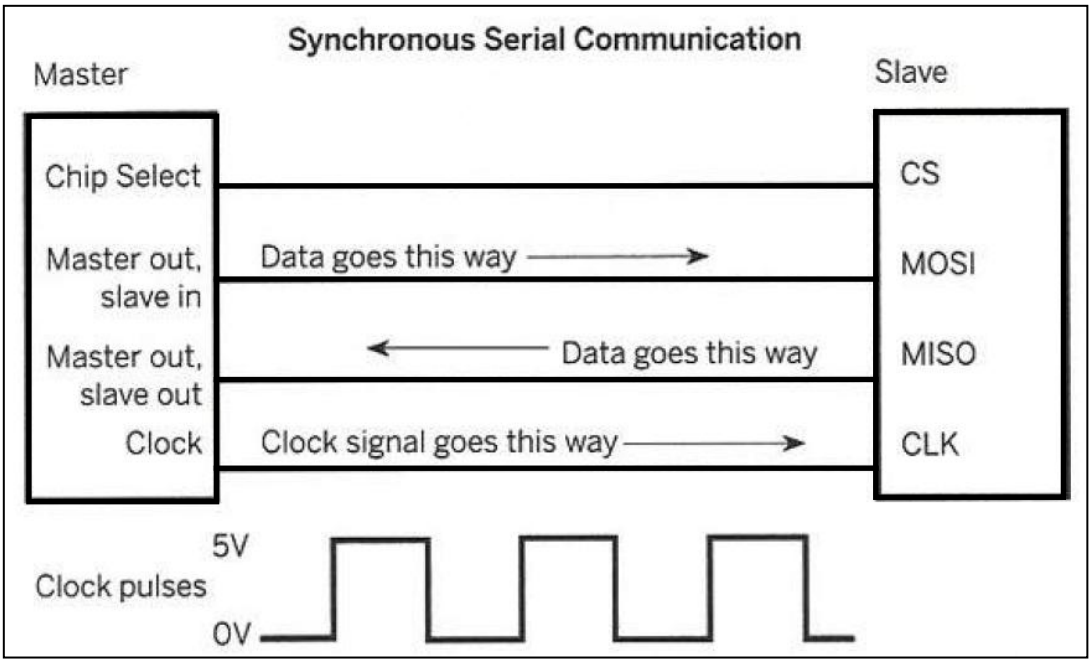


Fig. 22 – Representação de uma comunicação série síncrona
Fonte: Retirado de Igoe 2007, pág. 51

As formas de comunicação série assíncrona e síncrona (Fig. 19 e 20, respectivamente), são bastante distintas. A comunicação síncrona necessita de uma "Master" e de um "Slave", em que o "Master" é que estabelece a velocidade de transmissão e vai regulando o fluxo dos dados. Na comunicação assíncrona apenas existe um "Sender" e um "Receiver", sem haver um "controlo de fluxo" da comunicação por alguma das partes.

A bilbioteca <SoftwareSerial.h> permite simular por *Software* (daí o nome) uma comunicação série assíncrona. A não utilização de uma peça de *Hardware* para efectuar a comunicação, face aos pinos "*standard*" de comunicação, faz com que não se tenha a capacidade de ter um *buffer* de entrada (64 bytes de buffer de entrada utilizando **UART**). Ou seja, todos os dados que forem enviados para pinos de Rx configurados por *Software*, e se não se encontrarem a ser "lidos", serão perdidos.

4.4.1. Sintaxe das Instruções da Biblioteca <SoftwareSerial.h>

4.4.1.1. SoftwareSerial(Pino de Rx, Pino de Tx)

Descrição: Permite a criação de uma variável do tipo "SoftwareSerial", conseguindo definir o respectivo pino de Tx e Rx.

Sintaxe:

SoftwareSerial Nome_variável = ***SoftwareSerial***(Pino de Rx, Pino de Tx);

Nota: É importante ter em conta que os pinos definidos anteriormente para Rx e Tx devem ser declarados na função "void setup()" como "INPUT" e "OUTPUT" respectivamente.

4.4.1.2. begin(Baud Rate)

Descrição: Possibilita definir qual a **baud rate** a utilizar para a respectiva comunicação série assíncrona, devendo o valor da baud rate situar-se abaixo de 9600. Pois valores superiores a este, não obtêm boa performance na comunicação recorrendo a este "método" de comunicação.

Sintaxe:

SoftwareSerial.begin(Baud Rate);

- ***"SoftwareSerial"*** – Variável do tipo "SoftwareSerial", sendo definida da seguinte forma:

SoftwareSerial Nome_variável = ***SoftwareSerial***(Pino de Rx, Pino de Tx);

Tomando depois a seguinte sintaxe:

Nome_variável.begin(Baud Rate);

Exemplo:

```
(1) #include<SoftwareSerial.h> // Declaração da biblioteca <SoftwareSerial.h>
(2) SoftwareSerial Exemplo = SoftwareSerial(4,5); // Declaração de uma variável do tipo SoftwareSerial de nome "Exemplo", atribuindo-lhe o pino de Rx "4'' e o pino de Tx "5''
(3) void setup(){
(4) pinMode(4, INPUT); // Define o pino 4 como "INPUT"
(5) pinMode(5, OUTPUT); // Define o pino 5 como "OUTPUT"
```

(6) Exemplo.begin(9600); *// Define a velocidade de comunicação com uma Baud rate de 9600*
(7) void loop(){
(8) (……)
(9) }

4.4.1.3. read()

Descrição: Instrução que possibilita a leitura de um caracter proveniente do pino definido como Rx. A instrução "***SoftwareSerial***.read()" espera que chegue um caracter e retorna o seu valor, sendo preciso ter em atenção que os dados que cheguam sem haver uma "leitura"a aguardar serão perdidos (não existência de *buffer* de recepção).

Sintaxe:

Valor Leitura (int/char) = ***SoftwareSerial***.read();

- ***"SoftwareSerial"*** – Variável do tipo "SoftwareSerial", sendo definida da seguinte forma:

SoftwareSerial Nome_variável = ***SoftwareSerial***(Pino de Rx, Pino de Tx);

Tomando depois a seguinte sintaxe:

Valor Leitura (int/char) = Nome_variável.read();

Exemplo:

(1) #include<SoftwareSerial.h> *// Declaração da biblioteca <SoftwareSerial.h>*
(2) SoftwareSerial Exemplo = SoftwareSerial(4,5); *// Declaração de uma variável do tipo SoftwareSerial de nome "Exemplo", atribuindo-lhe o pino de Rx "4" e o pino de Tx "5"*
(3) char recebido; *//Declaração de uma variável do tipo char de nome "recebido"*
(4) void setup(){
(5) pinMode(4, INPUT); *// Define o pino 4 como "INPUT"*
(6) pinMode(5, IOUTPUT); *// Define o pino 5 como "OUTPUT"*
(7) Exemplo.begin(9600); *// Define a velocidade de comunicação com uma Baud rate de 9600*
(8) void loop(){
(9) recebido = Exemplo.read(); *//Possibilita guardar um valor recebido no pino de Rx, na variável do tipo char "recebido"*
(10) }

4.4.1.4. print(dados) vs println(dados)

Descrição: A instrução "***SoftwareSerial***.print()" tem um comportamento semelhante à instrução "Serial.print()". E, fazendo a mesma analogia, podemos afirmar que a instrução "***SoftwareSerial***.println()" apresenta um comportamento semelhante à instrução "Serial.println()". Qualquer dúvida em qual o tipo de comportamento das instruções da comunicação série "standard", estão disponíveis informações no subcapítulo **3.5.8** (página 44 à 46).

Sintaxe:

SoftwareSerial.print(dado a enviar);

SoftwareSerial.println(dado a enviar);

- ***"SoftwareSerial"*** – Variável do tipo "SoftwareSerial", sendo definida da seguinte forma:

SoftwareSerial Nome_variável = ***SoftwareSerial***(Pino de Rx, Pino de Tx);

Tomando depois a seguinte sintaxe:

Nome_variável. print(dado a enviar);

Nome_variável. println(dado a enviar);

Exemplo:

```
(1) #include<SoftwareSerial.h> // Declaração da biblioteca <SoftwareSerial.h>
(2) SoftwareSerial Exemplo = SoftwareSerial(4,5); // Declaração de uma variável do tipo SoftwareSerial de nome "Exemplo", atribuindo-lhe o pino de Rx "4'' e o pino de Tx "5''
(3) char recebido; //Declaração de uma variável do tipo char de nome "recebido"
(4) void setup(){
(5) pinMode(4, INPUT); // Define o pino 4 como "INPUT"
(6) pinMode(5, IOUTPUT); // Define o pino 5 como "OUTPUT"
(7) Exemplo.begin(9600); // Define a velocidade de comunicação com uma Baud rate de 9600
(8) void loop(){
(9) recebido = Exemplo.read(); //Possibilita guardar um valor recebido no pino de Rx, na variável do tipo char "recebido"
(10) Exemplo.print(recebido); // Envia a variável do tipo char de nome "recebido", através do pino definido com Tx (pino 5)
(11) }
```

4.5. FIRMATA vs Processing

Não é objectivo deste tutorial abordar a sintaxe utilizada pela biblioteca FIRMATA, mas sim dar a conhecer a sua existência. Esta biblioteca foi elaborada de forma a criar um protocolo genérico de comunicação entre o *Arduino* e um *Software* não específico para essa aplicação, bastando que o *Software* tenha capacidade de comunicação por porta série, desde que isso aconteça é possível haver um controlo por parte de *Software* do *Arduino*.

A distribuição padrão do *sketch* elaborado utilizando esta biblioteca toma o nome de "standard Firmata", podendo este ser completamente adaptado para a aplicação específica de cada utilizador. Uma implementação muito procurada por artistas e designers é a implementação do *Arduino* com o *Software open source* "Processing". Este *Software* permite uma programação visual, ou seja, permite através de instruções de programação fazer objectos visuais (p.ex. quadrados, círculos, linhas, entre outros). Permitindo, assim, desenhar o que se pretende, bem como criar verdadeiras obras de arte.

De acordo com Shiffman, 2008, este software foi desenvolvido por Benjamin Fry e Casey Reas, enquanto alunos no "MIT Media Lab" em 2001. Em Shiffman, 2008, o Software de desenvolvimento "Processing" é apenas abordado como uma ferramenta de ensino de programação, ou seja, o principal foco não é o *Software* em si mas os conceitos computacionais por detrás dele. O que leva a que seja uma leitura aconselhada para quem quer saber mais, não só sobre o *Software* em si, mas sobre a linguagem de programação java. Visto o "Processing" ser baseado na linguagem java, uma linguagem de programação largamente difundida e utilizada.

O "Processing" pode ser obtido gratuitamente através do seu site oficial (www.processing.org), na secção *downloads.* Estão disponíveis versões para Windows, Linux e Macintosh OS.

A interacção entre o "Processing" e o *Arduino* pode ser facilmente apreendida recorrendo ao separador "Playground", do site oficial *Arduino* (www.arduino.cc).

Este subcapítulo não refere obviamente a tudo o que há a saber sobre esta temática sendo o seu principal objectivo, como foi referido anteriormente, dar a conhecer novas possibilidades de implementação utilizando o *Arduino*.

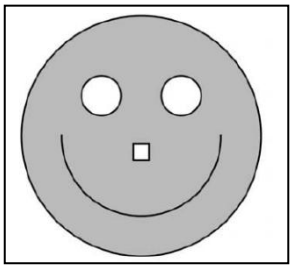


Fig. 23 – Representação possível de um "Hello World" utilizando o "Processing"
Fonte: Retirado de Shiffman 2008, pág. 10

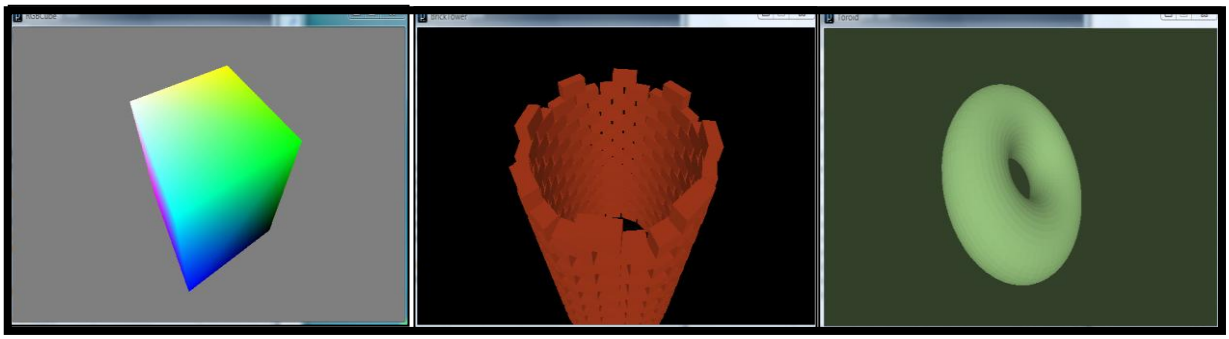


Fig. 24 – Alguns exemplos criados utilizando o "Processing"

4.6. Aquisição de Sinal – Conversão A/D

4.6.1. Alguns Conceitos Teóricos

Ao considerar um sinal discreto estamos, na maioria das vezes, a considerar um sinal contínuo no tempo que foi amostrado com uma determinada taxa de amostragem.

Para efectuar a amostragem de um sinal analógico para uma sequência digital, recorre-se a um conversor A/D (ADC – Analog-to-digital converter). Os constituintes de um conversor A/D são os seguintes:

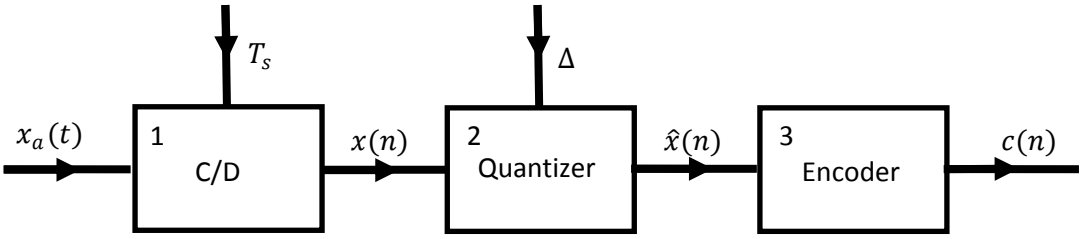


Fig. 25 – Constituintes de um conversor A/D
Fonte: Adaptado de Hayes 1999, pág. 101

No bloco 1 da Fig. 25, normalmente chamado de "Sampler", é permitida a conversão de um sinal analógico – $X_a(t)$ - para uma sequência discreta – X(n). Esta sequência discreta é criada ao guardar os valore de $X_a(t)$ em intervalos de tempo constantes T_s.

$$X(n) = X_a(n.T_s) \text{ com } n \epsilon N^+$$

No bloco 2 é efectuada uma quantificação dos valores de amplitude obtidos pelo "Sampler", ou seja, como as amostras X(n) têm múltiplos valores possíveis de amplitude é necessário atribuir valores de amplitude também discretos. Estes valores discretos de amplitude estão divididos em intervalos (Δ). Quantos mais intervalos de quantização houver e maior número de bits, mais precisão vou obter em relação ao valor real.

No 3º e último bloco é atribuída a respectiva sequência binária, para cada valor amostrado e quantificado, obtendo aqui os nossos valores digitais para a sequência de entrada analógica.

Para escolhermos a taxa de amostragem a utilizar em cada situação, de forma a ter uma representação discreta fiável do sinal original, temos de ter em conta o **Teorema de Nyquist** (Teorema da amostragem). Este teorema pode ser basicamente enumerado pela seguinte condição:

$$f_{amostragem} \geq 2 * f_{maior\ do\ sinal\ a\ amostrar}$$

Ao respeitarmos este teorema estão garantidas as condições para não ocorrer ***aliasing*** e o sinal original poderá ser reconstruído a partir da sua amostra, recorrendo a um simples filtro passa baixo (Hayes 1999). É referido como ***aliasing*** a distorção causada por uma taxa de amostragem insuficiente (Vasegui 2006).

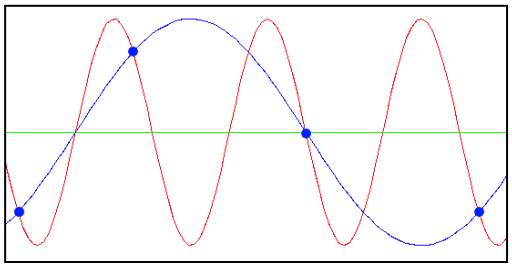


Fig. 26 – Exemplo de **aliasing**

Na figura 26, podemos ter um exemplo do que acontece quando o teorema da amostragem não é respeitado. O sinal a vermelho (sinal que se pretende amostrar) é amostrado como uma frequência de amostragem inferior à frequência mínima necessária para a reconstrução do sinal original ($f_{amostragem} \geq 2 * f_{máxima\ do\ sinal\ a\ amostrar}$). Sendo o resultado dessa amostragem a onda a azul, que não retrata claramente o sinal original (existência clara de ***aliasing***).

4.6.2. Arduino vs ADC

Após uma breve introdução de alguns dos conceitos associados a esta temática, torna-se agora importante abordar o nosso caso concreto (Uso do *Arduino*).

Os microcontroladores utilizados até ao momento, nos modelos disponíveis da plataforma de desenvolvimento *Arduino*, possuem conversores A/D com 10 bits de resolução (informação retirada do datasheet dos respectivos microcontroladores), ou seja, uma entrada analógica cujo valor varia entre 0 V e 5 V, terá a sua correspondência em binário a valores entre 0 (0000000000) e 1023 (1111111111) respectivamente. O que nos leva a resoluções, como foi referido no subcapítulo 2.1.4., na ordem dos 5 mV.

Segundo informação disponível no datasheet dos respectivos microcontroladores, a primeira conversão A/D demora 25 ciclos de Clock enquanto as restantes demoram 13. Esta diferença acontece pois a primeira conversão A/D precisa de efectuar a inicialização do ADC.

Analisando o referido anteriormente, e considerando um Clock de 1 MHz podemos facilmente podemos concluir o seguinte (Ignorando a conversão inicial):

$$Nº_{amostras\ \ por\ segundo} = \frac{10^6}{13} \cong 77000$$

Ou seja, estamos perante uma taxa de 77 kHz. Analisando a taxa de amostragem obtida, e tendo em conta o referido no subcapítulo anterior, a frequência máxima do sinal a amostrar deve ser menor que 38.5 kHz, de forma a respeitar o **Teorema da Amostragem**.

O *Arduino* possui um "Clock de sistema" de 16 MHz, mas no entanto não é este o Clock de entrada que vamos obter no nosso conversor A/D. O Clock de sistema é dividido por um "factor de divisão" (**Prescaler**), sendo este sim o Clock que vamos obter no nosso ADC como podemos constatar pela figura seguinte:

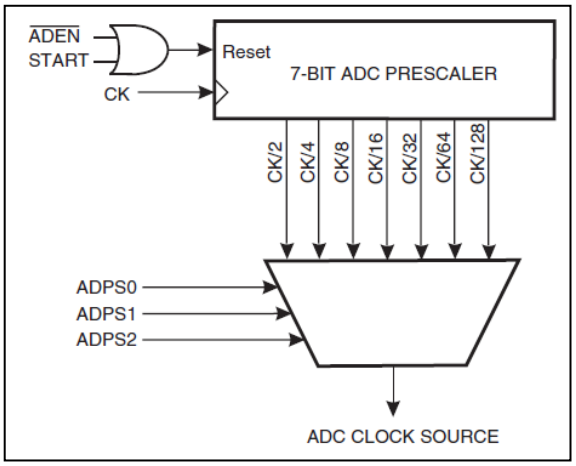


Fig. 27 – ADC Prescaler
Fonte: Retirado do datasheet do microcontrolador ATMega168

Mas no entanto existe uma forma de configurar este "factor de divisão", fazendo variar o conteúdo do bit **ADPS0**, **ADPS1** e **ADPS2** (Como se pode constatar pela análise da figura 27).

Estes bits são parte integrante do registo **ADCSRA**, que é constituído da seguinte forma:

Bit	7	6	5	4	3	2	1	0
(0x7A)	*ADEN*	*ADSC*	*ADATE*	*ADIF*	*ADIE*	*ADPS2*	*ADPS1*	*ADPS0*
Read/Write	*R/W*	*R/W*	*R/W*	*R/W*	*R/W*	*R/W*	*R/W*	*R/W*
Valor Inicial	0	0	0	0	0	0	0	0

Tabela 6 – Conteúdo do registo **ADCSRA**
Fonte: Retirado do datasheet do microcontrolador ATMega168

As combinações possíveis, para os bits referidos anteriormente, podem ser resumidas pela tabela seguinte:

ADPS2	ADPS1	ADPS0	Factor de divisão
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

Tabela 7 – Conteúdo do registo **ADCSRA**
Fonte: Retirado do datasheet do microcontrolador ATMega168

Ou seja, o Clock de entrada do conversor A/D é obtido através da seguinte condição:

$$f_{input\ \ clock} = \frac{Clock\ de\ Sistema\ (16\ MHz)}{Factor\ de\ divisão(\boldsymbol{Prescaler})}$$

Da equação anterior podemos retirar facilmente, e apenas efectuando o cálculo acima referido para cada condição, as seguintes conclusões:

Factor de divisão	Clock de entrada no ADC (MHz)
2	8
4	4
8	2
16	1
32	0.5
64	0.25
128	0.125

Tabela 8 – Valores possíveis de Clock de entrada no ADC

Pela análise da tabela anterior, podemos afirmar então que a maior frequência de amostragem que poderemos obter se situa nos 8 MHz. O que leva a que a que para haver uma reconstrução sem perda de informação de um possível sinal amostrado, o mesmo deverá ter uma frequência máxima de 4 MHz. No entanto os valores apresentados na tabela 8 são

apenas teóricos, correspondendo os valores reais a valores bastante inferiores aos apresentados.

4.6.3. Sintaxe para Alterar o Valor do "Factor de Divisão"

Depois do abordado, anteriormente, resta referir a forma de o fazer, indicando a sintaxe a utilizar para alterar os registos referidos anteriormente. A sintaxe utilizada não serve especificamente para alterar o conteúdo do registo **ADCSRA**, podendo a mesma sintaxe ser utilizada para alterar o conteúdo de outros registos com as respectivas adaptações é claro. Os registos disponíveis, e o seu conteúdo e função, encontram-se disponíveis no *datasheet* do respectivo microcontrolador. De seguida vai ser descrito quais as instruções a utilizar, dando um exemplo de implementação:

Código a utilizar:

```
(1)  #ifndef cbi
(2)  #define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
(3)  #endif
(4)  #ifndef sbi
(5)  #define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
(6)  #endif
(7)  void setup(void) {
(8)  Serial.begin(9600); ); // Inicializa o envio por Série, e define a Baud Rate
(9)  sbi(ADCSRA, ADPS2); // Neste caso concreto o bit ADPS2 será definido com o valor "1"
(10) cbi(ADCSRA, ADPS1); // Neste caso concreto o bit ADPS1 será definido com o valor "0"
(11) cbi(ADCSRA, ADPS0); // Neste caso concreto o bit ADPS0 será definido com o valor "0"
(12) }
(13) int a,b,c;
(14) void loop(){
(15) b = micros();); // Atribui à variável "b" o tempo actual de execução do programa actual
(16) a = analogRead(0);
(17) c = micros(); // Atribui à variável "c" o tempo actual de execução do programa actual
(18) Serial.print(c-b); // Envia (série) o valor necessário para executar a instrução "analogRead" (conversão A/D)
(19) }
```

Sintaxe utilizada:

sbi(Nome do registo, Nome do bit); *// Define o bit "Nome do bit" situado no registo "Nome do registo" a 1 (set bit)*

cbi(Nome do registo, Nome do bit); *// Define o bit "Nome do bit" situado no registo "Nome do registo" a 0 (clear bit)*

O código acima apresentado está elaborado de forma a, medir na realidade qual o tempo gasto a efectuar uma conversão A/D. Os valores obtidos por medidas experimentais, fazendo a média de 1000 amostras de tempo gasto na conversão A/D (Valores obtidos por comunicação série efectuando a aquisição e cálculo com o Software MatLab). Os resultados obtidos foram os seguintes:

Factor de divisão	Taxa de amostragem $(\boldsymbol{KHz})$
128	$\cong 8$
64	$\cong 16$
32	$\cong 31$
16	$\cong 50$
8	$\cong 63$
4	$\cong 83$
2	$\cong 125$

Tabela 9 – Valore obtidos na conversão A/D

O que corresponde, como foi referido anteriormente, a taxas de amostragem bastante inferiores às teoricamente esperadas.

Espera-se com este subcapítulo dedicado à conversão A/D, dar a entender a enorme importância desta temática. Pois a aquisição de sinal e respectivo processamento têm sido assunto de grande estudo. Espera-se assim ter aberto os horizontes a esta temática.

5. Conclusão

Espera-se com este tutorial "abrir as portas" para um entendimento mais aprofundado desta plataforma de desenvolvimento. Não correspondendo este documento a tudo o que existe para aprender e desenvolver sobre este tema. Muitos assuntos ficaram por abordar e caberá ao leitor caso tenha manifesto interesse (confunde-se muitas vezes com necessidade), pesquisar e desenvolver uma temática específica.

A placa de desenvolvimento Arduino encontra-se em franca expansão, estando a sua utilização generalizada quer na área da electrónica, artes e muitas mais. O que leva a que esta plataforma tenha muitos seguidores e estando a crescer e a desenvolver-se a cada minuto que passa. O que agora é uma novidade de implementação, muito provavelmente quando estiver a ler este tutorial será uma coisa comum. Pois o Arduino está a conquistar o seu espaço, e já é uma referência.

Quem diria que um projecto académico chegaria tão longe, provavelmente ninguém diria. O que faz com que as pessoas que acreditaram, estejam desde já de parabéns pelo magnífico trabalho. Deste exemplo real só podemos retirar que a persistência e dedicação são o que nos leva mais longe, pois os criadores desta plataforma de desenvolvimento não são os melhores. Mas sim os que acreditaram e trabalharam para isso, o que se revela sempre muito importante.

O que é preciso é ter uma atitude de contínua procura pelo conhecimento….

"Pois o saber não ocupa espaço de memória"

6. Bibliografia

Arduiniana. (2009). "Arduino software jewellry and wisdom." Retrieved 4 August 2009, 2009, from http://arduiniana.org/.

Arduino. (2006, 14 July 2009). Retrieved 21 July 2009, 2009, from http://www.arduino.cc.

Arroz, G., J. Monteiro, et al. (2007). Arquitectura de Computadores dos Sistemas Digitais aos Microprocessadores. Lisboa, IST Press.

AVR. (2009). "pgmspace reference." Retrieved 31 July 2009, 2009, from http://www.nongnu.org/avr-libc/user-manual/group__avr__pgmspace.html.

Hayes, M. H. (1999). Schaum´s Outline of Theory and Problems of Digital Signal Processing. Washington DC, USA, McGraw-Hill.

Igoe, T. (2007). Making Things Talk. Sebastopol, USA, O´REILLY.

Santos, A. (2007). "Servomotores." Retrieved 31 July 2009, 2009, from http://www.esen.pt/on/file.php/45/Cerqueira/Servo_Motor.pdf.

Santos, N. P. (2008) "Introdução ao Arduino." Revista PROGRAMAR, 39-44.

Santos, N. P. (2009) "Arduino e a Aquisição de dados." Revista PROGRAMAR, 24-26.

Shiffman, D. (2008). Learning Processing - A beginner´s Guide to Programming Images, Animation, and Interaction. Burlington, USA, Morgan Kaufmann.

Sousa, D. J. d. S. and N. C. Lavinia (2006). Conectando o PIC 16F877A Recursos Avançados. São Paulo, Brazil.

Tecnology, M. (1992). Advanced RISC Computing Specification. California, USA, MIPS Technology.

Vasegui, S. V. (2006). Advanced Signal Processing and Noise Reduction. Wiltshire, England, Wiley.