

# Arquiteturas para Sistemas Embutidos Apontamentos

Universidade de Aveiro

Sebastian D. González



# **Arquiteturas para Sistemas Embutidos Apontamentos**

Dept. de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

sebastian.duque@ua.pt(103690)Arquiteturas para Sistemas  
Embutidos

8 de julho de 2024

### **Warning!!**

Isto são apenas uns apontamentos realizados por uma pobre alma de MIECT, feitas a partir dos slides da disciplina e outras fontes 😈. Por favor, não usem apenas estes apontamentos como material de estudo.

Dito isto, boa sorte a todos e ámen CT 🙏.


Agradecimentos ao Professor Arnaldo Silva Rodrigues de Oliveira [arnaldo.oliveira@ua.pt](mailto:arnaldo.oliveira@ua.pt) por todo o material fornecido.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
	(Resumo de AC2)	
1.1	Microprocessador vs Microcontrolador . . . . .	1
1.1.1	Microprocessador . . . . .	1
1.1.2	Microcontrolador . . . . .	1
1.1.3	Técnicas de transferência de informação . . . . .	3
1.2	Revisão protocolos de comunicação em série . . . . .	4
1.2.1	RS-232C . . . . .	4
1.2.2	SPI . . . . .	4
1.2.3	I <sup>2</sup> C . . . . .	5
1.3	Timers . . . . .	6
1.3.1	WatchDog Timer . . . . .	6
<b>2</b>	<b>Compilação Software de ficheiros .c ou .cpp</b>	<b>7</b>
2.1	Make e Makefiles . . . . .	10
<b>3</b>	<b>Variáveis em C e C++</b>	<b>11</b>
3.1	Hierarquia de memória . . . . .	11
3.2	Tipos de variáveis: . . . . .	12
3.2.1	Variáveis locais ou automáticas . . . . .	12
3.2.2	Variáveis globais . . . . .	12
3.2.3	Disposição da memória em um programa C . . . . .	13
3.2.4	Variáveis externas . . . . .	14
3.2.5	Variáveis estáticas . . . . .	14
3.2.6	Variáveis signed e unsigned . . . . .	15
3.2.7	Ponteiros . . . . .	15
3.2.8	Aritmética de ponteiros . . . . .	16
<b>4</b>	<b>Timers</b>	<b>17</b>
4.1	Periférico vs Coprocessador . . . . .	17
4.2	Tipos de Timers . . . . .	17
4.3	Interrupções . . . . .	18

<b>5</b>	<b>ESP32-C3-DevKitM-1</b>	<b>19</b>
5.1	Principais características . . . . .	19
5.2	Driver TC74 . . . . .	20
5.2.1	Driver . . . . .	21
5.2.1.1	tc74_init: . . . . .	21
5.2.1.2	tc74_standby: . . . . .	22
5.2.1.3	tc74_read_temp_after_cfg: . . . . .	22
5.2.2	tc74_read_temp_after_temp . . . . .	22

# Lista de Figuras

1.1	Esquema de um barramento de um sistema com modelo Von Neumann [1]	2
1.2	CrossBar Interconnect	2
1.3	Simplified DMA block diagram [2]	3
1.4	Cabo RS-232C	4
1.5	Arquiteturas de ligação do SPI	5
1.6	Comunicação I <sup>2</sup> C	5
1.7		6
2.1	Pré processador do C	7
2.2	Dependência na compilação	7
2.3	Compilados C	8
2.4	Linker	9
2.5	Esquema completo da compilação do c em Linux	10
2.6	Esquema completo da compilação do c no Windows	10
3.1	Hierarquia de memória [4]	11
3.2	C program memory layout [5]	13
3.3	Variável externa [6]	14
4.1	Timer de Hardware	17
4.2	Interrupt Controller [7]	18
5.1	ESP32-C3-DevKitM-1 Pin Layout	19
5.2	Esquema de ligação entre o TC74 e o ESP32-C3-DevKitM-1	21

# Introdução

## (Resumo de AC2)

### **O que é um Sistema Embutido?**

Um sistema embutido é um sistema de computação dedicado ao dispositivo ou sistema que ele controla. Distintamente de computadores de propósito geral, um sistema embutido realiza um conjunto de tarefas predefinidas e específicas de forma eficiente. São sistemas compactos e consomem pouca energia comparativamente a um sistema computacional de uso geral.

## **1.1 Microprocessador vs Microcontrolador**

### **1.1.1 Microprocessador**

É um sistema computacional de uso geral utilizado para executar instruções e realizar operações lógicas e aritméticas. Um microprocessador é composto por uma unidade central de processamento (CPU), que interpreta e executa instruções, uma unidade de controle, que coordena as operações do processador, e uma unidade de aritmética e lógica, que realiza operações matemáticas e lógicas.

### **1.1.2 Microcontrolador**

É um sistema computacional usado em tarefas específicas que, geralmente, contém um microprocessador embutido. Os microcontroladores têm a característica de serem dispositivos programáveis que integram, num único circuito integrado, 3 componentes fundamentais:

1. Uma Unidade de Processamento
2. Memória (volátil e não volátil)
3. Portos de I/O (E/S) que disponibilizam uma grande variedade de periféricos e interfaces com o exterior.

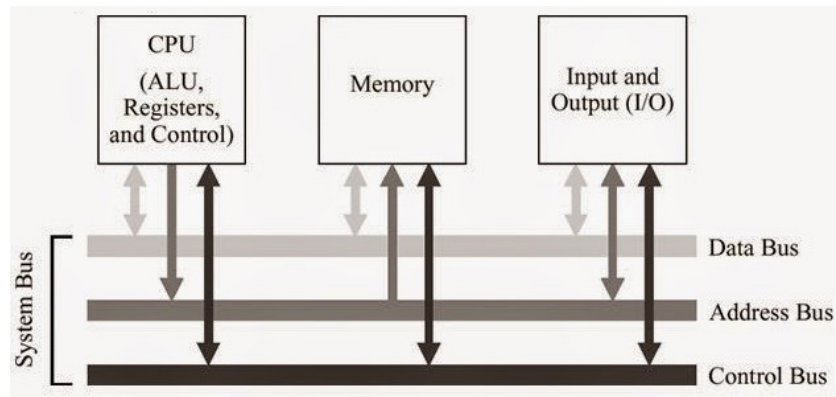


Figura 1.1: Esquema de um barramento de um sistema com modelo Von Neumann [1]

Existem barramentos (dados, endereços e controle) interligam todos estes dispositivos.

- **Data Bus:** Responsável pela troca de dados no computador, tanto enviados quanto recebidos.
- **Address Bus:** Indica a localização dos processos na memória(ou I/O) e para onde devem ser enviados após serem processados.
- **Control Bus:** Agrupa todo o conjunto de sinais elétricos de controle do sistema necessários para o bom funcionamento do sistema como, por exemp, sinais de escrita e leitura, definição de timers etc.

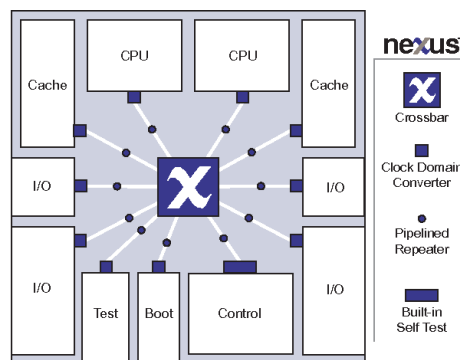


Figura 1.2: CrossBar Interconnect

Também existe outro tipo de organização de sistema chamado de CrossBar Interconnect em que todos os dispositivos em vez de estarem ligados diretamente pelo bus existe um intermediário chamado de Crossbar que controla e distribui as informações dos Busses.



O Interconnect aumenta o débito de instruções e permite um melhor desempenho porque 1 cpu pode aceder a multiplas memorias e vice versa. Podem não operar todos á mesma frequencia (vantagens do ponto de vista do consumo energético).

### 1.1.3 Técnicas de transferência de informação

- **SW- Software Oriented**
  - **Programmed I/O:** O CPU tem que esperar que o periférico esteja disponível para a troca de informação. Essa espera é efetuada num ciclo de verificação da informação de status do periférico, designado por **POLLING**.
  - **Interrupt driven I/O:** Periférico sinaliza o CPU de que está pronto para trocar informação (leitura ou escrita). O CPU inicia e controla a transferência.
- **HW- Hardware Oriented**
  - **Direct Memory Access:** Um dispositivo externo ao CPU (**DMA**) assegura a transferência de informação diretamente entre a memória e o periférico; o CPU não toma parte no processo de transferência.

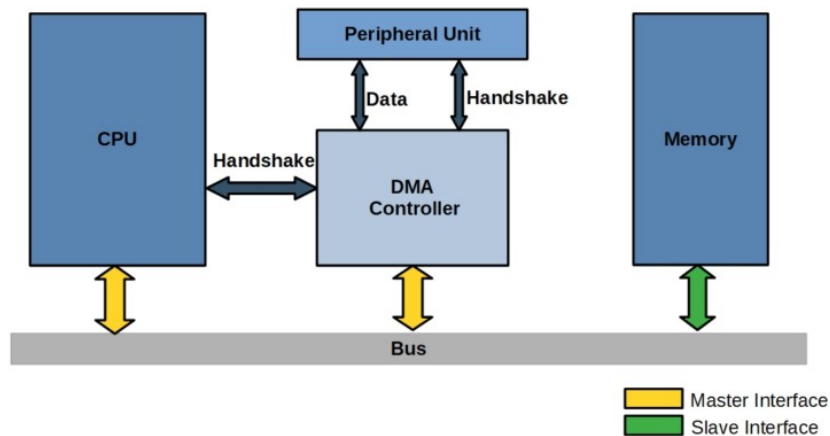


Figura 1.3: Simplified DMA block diagram [2]

## 1.2 Revisão protocolos de comunicação em série

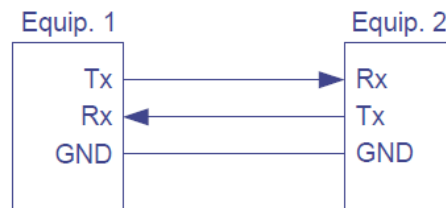
### 1.2.1 RS-232C

É um padrão de protocolo para troca série de dados entre dispositivos, definindo níveis de tensão para transmissão de dados e sinais de controle. É caracterizado por ser uma comunicação bidirecional **half-duplex**<sup>1</sup> assíncrona.



Figura 1.4: Cabo RS-232C

Na sua forma mais simples, a implementação da norma RS-232C requer apenas a utilização de 2 linhas de sinalização e uma linha de massa:



Podem ser usadas linhas adicionais para protocolar a troca de informação entre os dois equipamentos (handshake) incluindo

### 1.2.2 SPI

Serial Peripheral Interface é um protocolo de comunicação em série usado para trocas de dados em alta velocidade entre dispositivos tipicamente utilizado em microcontroladores para comunicar com uma grande variedade de dispositivos (ex: sensores, cartões de memória, etc).

Possui comunicação bidirecional e **full-duplex**<sup>2</sup> operando num paradigma **master-slave** em que o sistema apenas pode ter um master em que este controla o clock e inicia e controla a transferência de dados.

---

<sup>1</sup>**half-duplex** - comunicação nos dois sentidos apenas um de cada vez (usando apenas 1 linha)

<sup>2</sup>**full-duplex** - comunicação em simultâneo bidirecional usando duas linhas separadas uma para recepção e outra para transmissão

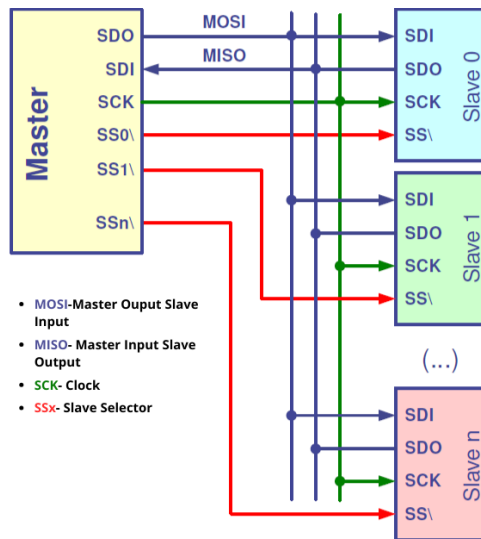


Figura 1.5: Arquiteturas de ligação do SPI

### 1.2.3 I<sup>2</sup>C

Inter-Integrated Circuit caracteriza-se por ter uma transferência bidirecional, half-duplex byte-oriented projetado para circuitos integrados. O ESP32 possui um controlador I<sup>2</sup>C que é o master.

- Master/Slave, em que o master pode ser transmissor ou recetor
- O barramento apenas necessita 2 fios
  - **SDA** - Serial Data Line
  - **SCL** - Serial Clock Line
- Barramento multi-master com deteção de colisões

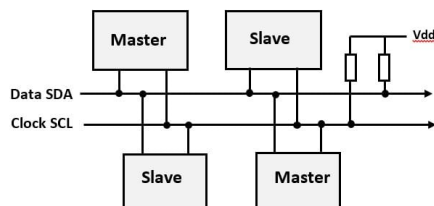
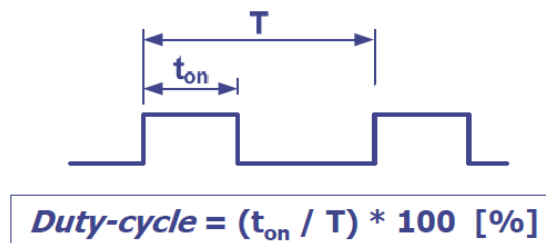


Figura 1.6: Comunicação I<sup>2</sup>C

## 1.3 Timers

**Timer** - é um dispositivo periférico que permite a **medição do tempo** partindo de uma referência temporal conhecida. Alguns exemplos de aplicações típicas de timers:

- Geração de um evento periódico com período e duração controlados. Exemplo: geração de um sinal periódico com um período de 10 ms e um **duty-cycle**<sup>3</sup> de 40%:



### 1.3.1 WatchDog Timer

Também conhecido como temporizador "cão de guarda"  $\Rightarrow$  🐕 tem como função monitorizar a operação do microprocessador e, em caso de falha, forçar o seu reinício.

Se o CPU não atuou na entrada reset do 🐕 antes de um período determinado, o 🐕 força o **reset** do microprocessador, garantindo que em caso de crash do microprocessador não comprometa o funcionamento global do sistema.

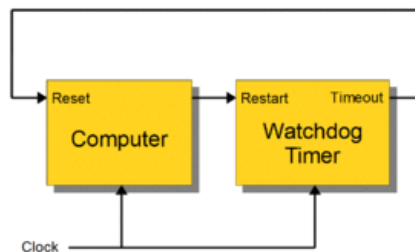


Figura 1.7: 🐕

<sup>3</sup>é a fração de tempo em que um sinal está em nível alto (ou ativo) comparado ao período total do sinal.

# Compilação Software de ficheiros .c ou .cpp

1. A primeira Ferramenta a ser invocada na compilação de um ficheiro .c é o **Pré processador**. Ele processa todas as diretivas de pré-processamento presentes no arquivo fonte (.c e .h) e gera um código intermediário, que que será compilado posteriormente.

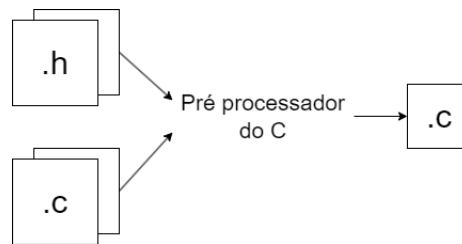


Figura 2.1: Pré processador do C

Durante esta etapa, as diretivas como `#include`, `#ifdef`, `#define`, entre outras são processadas. Por exemplo, quando o pré-processador encontra uma diretiva `#include`, ele substitui-a pelo conteúdo do arquivo incluído. Isto poderá gerar problemas no seguinte cenário:

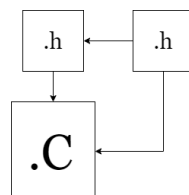


Figura 2.2: Dependência na compilação

Podemos observar na imagem 2.2 que temos um ficheiro `.c` que possui um include de um ficheiro `.h`, e de outro ficheiro `.h` que por sua vez também está a incluir o primeiro ficheiro `.h`. Logo, quando o pré processador for fazer as substituições de código vai gerar conflitos internos por repetir se o mesmo bloco de código.

Para resolver este problema é possível recorrer a um método chamado `#ifndef` ("if not defined") que permite verificar se uma determinada macro não foi definida anteriormente no código. Se a macro não estiver definida, o bloco de código entre `#ifndef` e `#endif` será incluído na compilação. Isto garante que o conteúdo do ficheiro `meu_header.h` seja incluído apenas uma vez em cada ficheiro fonte durante a sua compilação.

```
1 // meu_header.h
2 #ifndef MEU_HEADER_H
3 #define MEU_HEADER_H
4
5 // Protótipo da função
6 void minhaFuncao();
7
8 #endif // MEU_HEADER_H
```

Listing 2.1: Ficheiro `meu_header.h` que inclui o método `#ifndef`

```
1 // meu_programa.c
2 #include "meu_header.h"
3
4 int main() {
5     minhaFuncao(); // Chamando a função definida no cabeçalho
6     return 0;
7 }
```

Listing 2.2: Ficheiro `meu_programa.c` que inclui o ficheiro `meu_programa.c`

2. A segunda ferramenta a ser invocada é o próprio **Compilador** que pega nos files `.c` e produz um `.o` para cada `.c` os `.h` são incluídos de início pelo pré compilador).

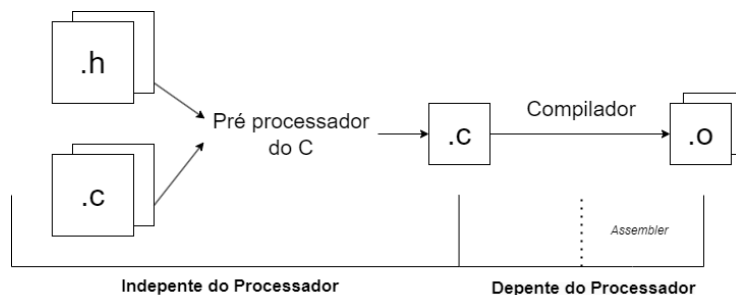


Figura 2.3: Compilados C

- ⇒ Um ficheiro `.o` é um ficheiro de **object code**. Este tipo de ficheiros contém código de máquina compilado, mas ainda não está finalizado para ser executado como um programa completo pois ainda faltam elementos para ele realmente se tornar executável.
- ⇒ Um ficheiro `.h` faz a assinatura das funções, protótipos, tipos de dados que sejam públicos.

3. A terceira ferramenta a ser usada é o **Linker**. Este é um programa que combina um ou mais ficheiros `.o` gerados pelo compilador num único ficheiro executável `.elf` (Executable and Linkable Format). Ele desempenha a função de juntar o código binário produzido pelo compilador, resolvendo questões de ligação como o uso de símbolos ou identificadores definidos em diferentes partes do programa.

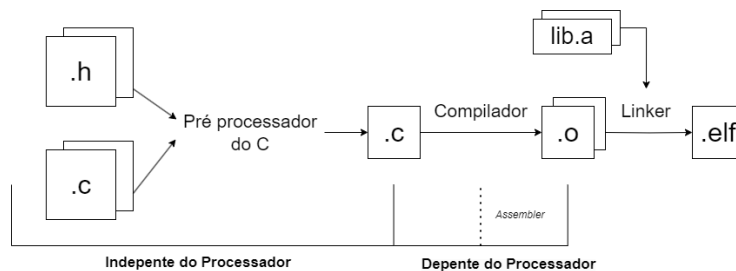


Figura 2.4: Linker

Além disso, o Linker também inclui todos os ficheiros de biblioteca **lib.a** durante a sua execução. Um ficheiro de biblioteca contém, internamente ficheiros, `.o` (trata-se de um arquivo gerado pela compilação de múltiplos `.c` que gerou múltiplos `.o` e foram arquivados num `.a` pronto a ser usado externamente. Este tipo de ficheiros são conhecidos como **bibliotecas estáticas**.

4. Finalmente, temos o **loader** que é responsável por carregar o ficheiro executável na memória principal para sua execução. As suas principais tarefas incluem ler o tamanho dos dados e código a ser carregado alocar espaço na memória para o programa, entre outras funções[3]. Uma biblioteca estática é utilizada por via de um linker. Uma biblioteca dinâmica utiliza o sistema operativo para verificar se a biblioteca está ou não carregada em memória, havendo 1 só cópia da biblioteca em causa poupando assim os recursos disponíveis.

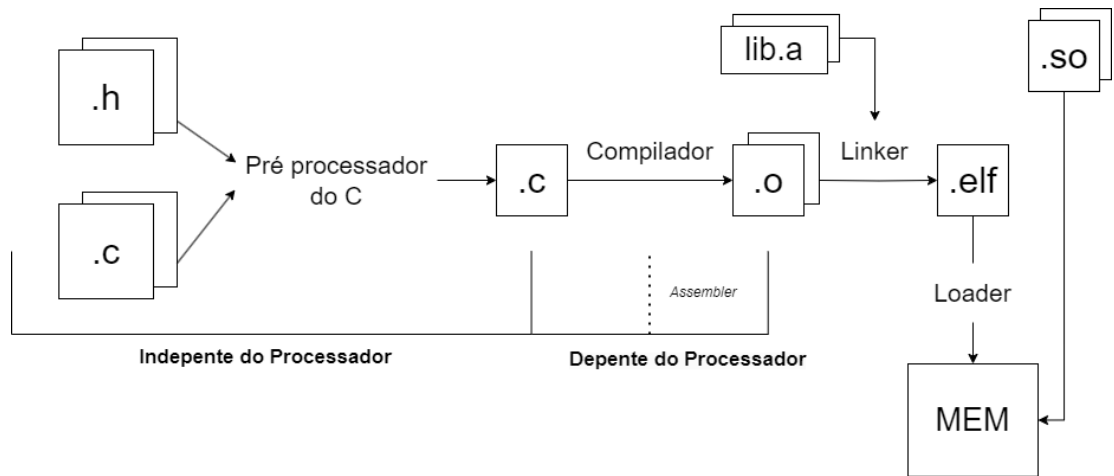


Figura 2.5: Esquema completo da compilação do c em Linux

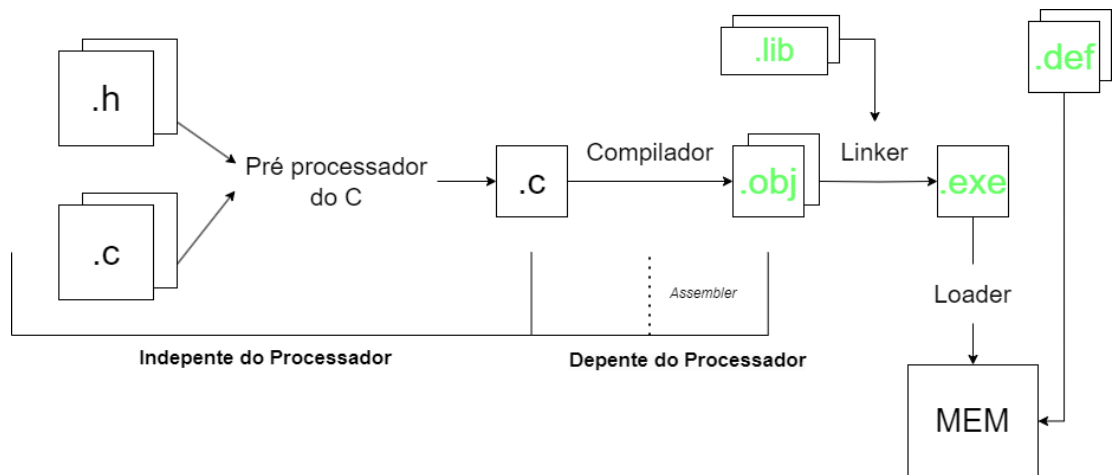


Figura 2.6: Esquema completo da compilação do c no Windows

## 2.1 Make e Makefiles

Permitem especificar dependências e regras durante o processo de compilação. Graças a isto quando se compila um dado excerto de código o Make apenas compila as alterações realizadas não recompilando o código todo de novo.



# Variáveis em C e C++

## O que é um bloco de código em C?

Em C, um bloco de código é uma secção contínua de declarações e comandos que são executados como uma unidade. É delimitado por chavetas `{}`.

```
1 void f(void) {  
2     int x;  
3     x = 10;  
4 }
```

Listing 3.1: Exemplo Bloco de código

## 3.1 Hierarquia de memória

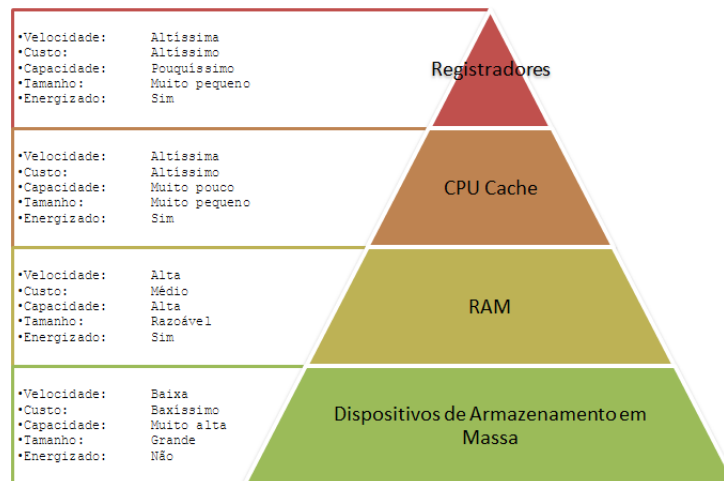


Figura 3.1: Hierarquia de memória [4]

## 3.2 Tipos de variáveis:

### 3.2.1 Variáveis locais ou automáticas

São todas as variáveis declaradas dentro de uma função. Como só existem enquanto a função estiver a ser executada, são criadas quando tal função é chamada e destruídas quando termina a execução desta. Parâmetros formais, que são os valores passados para a função como argumentos, também são considerados variáveis locais. Somente podem ser referenciadas pela função onde foram declaradas e os seus valores se perdem entre chamadas da função.

No exemplo 3.1 a variável `x` é uma variável local da função `f`.

### 3.2.2 Variáveis globais

São variáveis declaradas e/ou definidas fora de qualquer função do programa. Podem ser acessadas por qualquer função dentro ou fora do módulo e os seus valores existem durante toda a execução do programa.

⇒ São criadas quando o programa é alocado em memória e destruídas quando o programa finaliza e é retirada da memória.

```
1 #include <stdio.h>
2
3 int n = 100;
4
5 void func1() {
6     // Acessar a variável global dentro de uma função
7     n = 99;
8 }
9
10 int main() {
11     // Acessar a variável global dentro da função main
12     printf("Valor da variável global dentro de main(): %d\n", n);
13     func1();
14     printf("Valor da variável global depois de f1: %d\n", n);
15     return 0;
16 }
```

Listing 3.2: Exemplo de uma variável Global `n`

O output da função é:

```
1 Valor da variável global dentro de main(): 100
2 Valor da variável global depois de f1: 99
```

Listing 3.3: Output de 3.2

### 3.2.3 Disposição da memória em um programa C

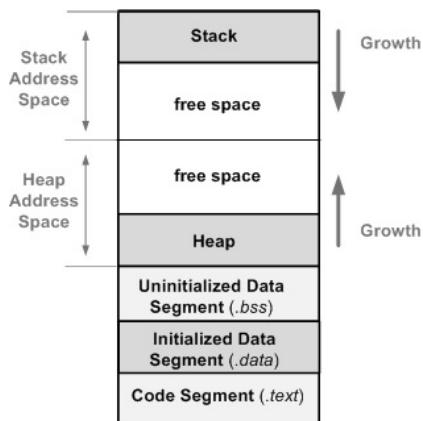


Figura 3.2: C program memory layout [5]

- **.text** - Também chamado **Code Segment** é uma área de memória que armazena o código executável do programa. Inclui instruções de máquina, constantes e alguns dados inicializados estaticamente.
- **.data** - Corresponde aos **Dados inicializados estaticamente** e é uma área de memória que armazena variáveis globais e estáticas que são inicializadas com um valor constante no momento da compilação.
- **.bss** - Corresponde aos **Dados não inicializados estaticamente** onde se armazena variáveis globais e estáticas que não são inicializadas com um valor constante no momento da compilação. Estas variáveis são automaticamente inicializadas com o valor zero.
- **Heap** - É um segmento de memória gerida explicitamente pelo utilizador. A alocação dinâmica de memória é realizada através de funções como `malloc()`, `calloc()` e `realloc()` <sup>1</sup>.

```
1 int *ptr;  
2 ptr = malloc(sizeof(int)); // Aloca memória para um único inteiro  
3 *ptr = 10;                // Atribui um valor  
4 free(ptr);                // Libertar a memória alocada
```

Listing 3.4: `malloc()`

- **Stack** - É uma área de memória utilizada para armazenar dados locais de funções e parâmetros de chamada de função. Cada chamada de função gera um novo quadro de pilha, que é removido da memória quando a função retorna. Arrays são guardados na stack.

<sup>1</sup>Se não se liberar a memória alocada dinamicamente usando a função `free()`, ocorre um memory leak.

### 3.2.4 Variáveis externas

Um programa em C pode ser composto por um ou mais ficheiros-fonte, compilados separadamente e posteriormente ligados, gerando um ficheiro executável final. Como as várias funções do programa estão distribuídas pelos arquivos-fonte, variáveis globais de um ficheiro não serão reconhecidas por outro, a menos que estas variáveis sejam declaradas como **externas**.

A variável externa deve ser definida somente num dos ficheiros-fonte e em quaisquer outros ficheiros deve ser referenciada mediante a declaração com a seguinte sintaxe:

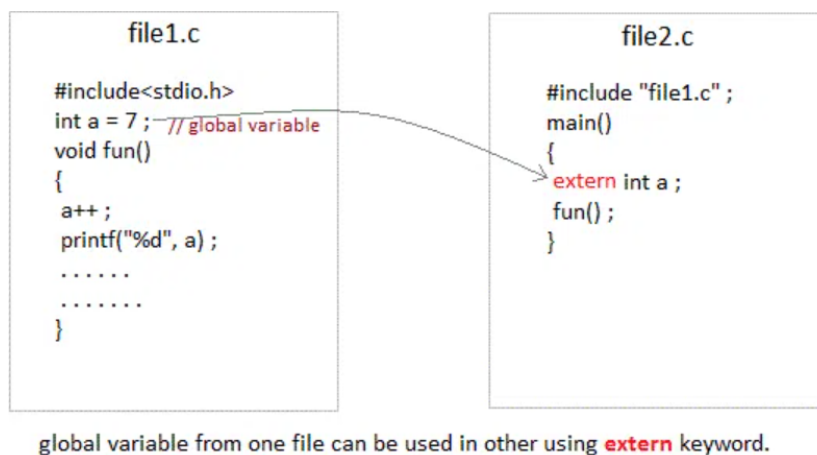


Figura 3.3: Variável externa [6]

### 3.2.5 Variáveis estáticas

Existem dois tipos de variáveis estáticas em C: variáveis estáticas de função e as variáveis estáticas de ficheiro.

#### Variáveis estáticas de função

São um tipo de variáveis que são apenas visíveis dentro da função em que foram declaradas e só são inicializadas em memória 1 vez, ou seja, como não são inicializadas sempre que a função é chamada são preservadas entre as chamadas dessa função.

```
1 #include <stdio.h>
2
3 void exemplo() {
4     static int contador = 0; // Variável estática de função
5
6     contador++;
7     printf("0 contador agora é: %d\n", contador);
8 }
```

```

9
10 int main() {
11     exemplo(); // Saída: 0 contador agora é: 1
12     exemplo(); // Saída: 0 contador agora é: 2
13     exemplo(); // Saída: 0 contador agora é: 3
14
15     return 0;
16 }

```

Listing 3.5: Variáveis estáticas de função

## Variáveis estáticas de ficheiro

Têm um scope global dentro do ficheiro onde são declaradas. Elas são declaradas fora de qualquer função e são visíveis para todas as funções desse ficheiro. No entanto, elas não são visíveis para funções em outros arquivos.

```

1 #include <stdio.h>
2
3 static int n = 10; // variável estática de ficheiro
4
5 void function1() {
6     printf("Valor da variável estática de arquivo em function1(): %d\n", n);
7 }

```

Listing 3.6: Variáveis estáticas de ficheiro

### 3.2.6 Variáveis signed e unsigned

Em C, e na maioria das linguagens de programação, **signed** e **unsigned** são especificadores que definem se um tipo de dado inteiro que pode representar números negativos e positivos ou apenas números não negativos.

Tipo	signed	Largura em Bits	Faixa de Valores
int8	Sim	8	-128 a 127
uint8	Não	8	0 a 255
int16	Sim	16	-32.768 a 32.767
uint16	Não	16	0 a 65.535

Tabela 3.1: Tabela de tipos de dados

### 3.2.7 Ponteiros

Os **Ponteiros** em C são variáveis que armazenam endereços de memória. Eles permitem acessar e manipular dados diretamente na memória do computador. Para declarar um ponteiro em C, usa-se o tipo de dado seguido de um asterisco (\*) e o nome da variável ponteiro:

```
1 int *ptr; // declara um ponteiro para um inteiro
```

Listing 3.7: Declaração de um ponteiro

Para atribuir o endereço de uma variável a um ponteiro, usa-se o operador de endereço (&). Por exemplo:

```
1 int a = 10;
2 int *ptr;
3 ptr = &a; // ptr agora aponta para a variável a
4
5 printf("Valor de *ptr: %d\n", *ptr); // imprime o valor apontado por
   pt que é 10
```

Listing 3.8: Declaração de um ponteiro

### 3.2.8 Aritmética de ponteiros

Ponteiros podem ser incrementados ou decrementados para apontar para o próximo ou anterior elemento de um array:

```
1 #include <stdio.h>
2
3 int main() {
4     // Define um array de inteiros
5     int arr[] = {10, 20, 30, 40, 50};
6
7     // Declara um ponteiro para um inteiro
8     int *ptr;
9
10    // Faz o ponteiro apontar para o primeiro elemento do array
11    ptr = &arr[0]; // ou simplesmente ptr = arr;
12
13    // Imprime o valor do primeiro elemento do array usando o ponteiro
14    printf("Conteúdo do primeiro índice do array: %d\n", *ptr);
15    // Imprime o endereço de memória do ponteiro
16    printf("Endereço apontado por ptr: %p\n", (void*)ptr);
17
18    return 0;
19 }
```

Listing 3.9: Ponteiro para o primeiro índice de um array

```
1 int v[5] = {10, 20, 30, 40, 50};
2 int *p;
3
4 p = v; // p aponta para o primeiro elemento de v
5 printf("Valor de v[0]: %d\n", *p); // imprime 10
6
7 p++; // p agora aponta para v[1]
8 printf("Valor de v[1]: %d\n", *p); // imprime 20
```

Listing 3.10: p é incrementado para apontar para o próximo elemento do array

# Timers

## 4.1 Periférico vs Coprocessador

Para aceder a um periférico é através de acessos á memoria(loads e stores). Os coprocessadores são acedidos através de registos dedicados e instruções específicas, sem mapeamento nos endereços de memória principais.

## 4.2 Tipos de Timers

**Timers de Hardware:** Os timers de hardware são implementados como parte do hardware do sistema, geralmente como um contador ou um circuito dedicado. Podem ser utilizados para tarefas sensíveis ao tempo, como controlo de dispositivos em tempo real, captura de eventos ou geração de pulsos.

⇒ 1 timer de Hardware pode ter vários Timers de Software

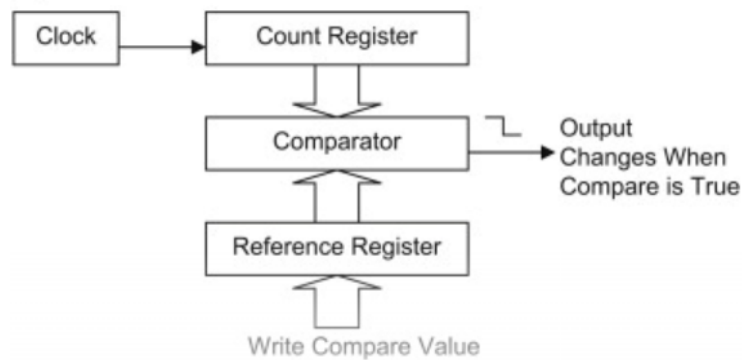


Figura 4.1: Timer de Hardware

Normalmente, são compostos por três componentes principais:

- **Counter:**
  - O contador é um componente do timer que conta ciclos de clock ou unidades de tempo, dependendo da configuração. Ele é responsável por acompanhar o tempo decorrido desde que foi iniciado ou dado **reset**.
- **Comparator:**
  - O comparador é usado para comparar o valor atual do contador com um valor de referência predefinido. Quando o valor do contador atinge ou ultrapassa o valor do comparador, uma ação é desencadeada, como uma interrupção ou uma operação específica.
- **Max Count Register(Reference Register):**
  - Este registo armazena o valor máximo que o contador pode atingir antes de ser reiniciado. Uma vez que o contador atinge esse valor, ele pode ser reiniciado automaticamente para começar a contagem novamente.

**Timers de Software:** Os timers de software são implementados através de instruções de programação num software ou sistema operativo e dependem do relógio interno do processador para medir o tempo. Geralmente, são menos precisos em comparação com os timers de hardware devido a atrasos de execução devido ao agendamento do sistema operacional e outras tarefas em segundo plano.

## 4.3 Interrupções

Um timer pode gerar interrupções através de um mecanismo específico. Em muitos sistemas, especialmente em arquiteturas de computadores modernas, as interrupções geradas pelo timer são tratadas pelo **Interrupt Controller**.

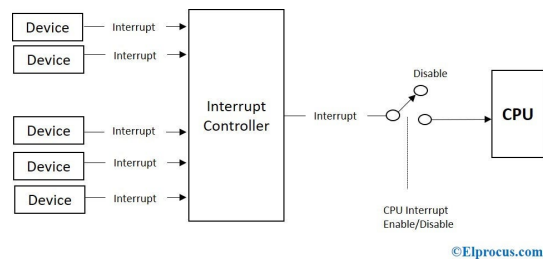


Figura 4.2: Interrupt Controller [7]

⇒ O **Interrupt Controller** é responsável por receber, priorizar e encaminhar interrupções para o processador.



# ESP32-C3-DevKitM-1

O **ESP32-C3-DevKitM-1** é uma placa de desenvolvimento compacta da [Espressif Systems](#) projetada em torno do módulo ESP32-C3-MINI-1.

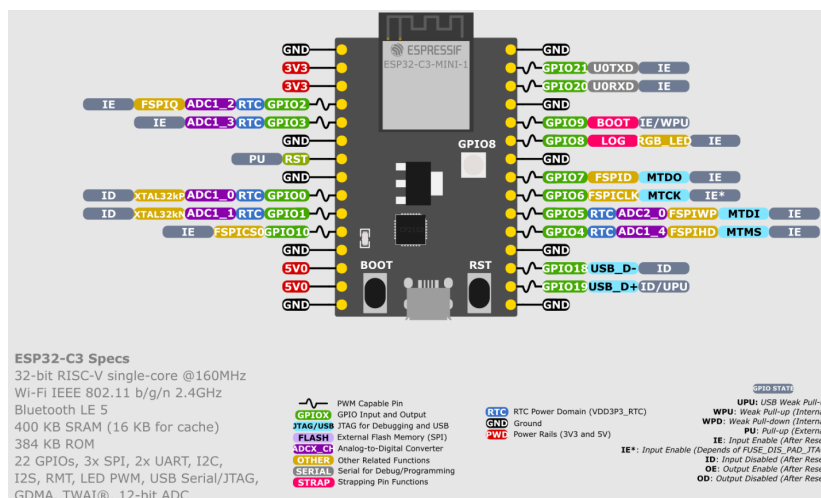


Figura 5.1: ESP32-C3-DevKitM-1 Pin Layout

## 5.1 Principais características

- **Processador:** Possui um núcleo RISC-V de 32 bits
- **Conectividade:**
  - **Wi-Fi:** Suporta 2.4 GHz com compatibilidade IEEE 802.11b/g/n.
  - **Bluetooth LE:** Bluetooth 5.0 Low Energy, permitindo comunicação de baixo consumo de energia e maior alcance.

- **Memória:**

- **RAM:** 400 KB de SRAM, sendo que 16 KB são configurados para cache
- **Flash:** suporta um flash externo com capacidade máxima de 16 MB, cujo espaço de endereço da memória de instrução e dados pode ser mapeado pela CPU.

- **GPIOs e Periféricos:** A placa oferece múltiplos pinos de entrada e saída (GPIO), ADC, UART, SPI, I<sup>2</sup>C, PWM e outros periféricos, facilitando a interface com sensores e outros dispositivos periféricos.

- Pode-se usar os pinos **0,1,2,3,4,5,6,7,10** para programar pois
- Não se podem usar os pinos **8,9,18,19,20,21** a menos que seja para as tarefas específicas às que foram desenhadas

- **Programação e Desenvolvimento:** Compatível com a ESP-IDF (Espressif IoT Development Framework) e outras plataformas como [Arduino IDE](#), oferecendo flexibilidade para os programadores.

No ESP32-C3-DevKitM-1, a maioria dos pinos são **multiplexados**<sup>1</sup>, permitindo uma grande flexibilidade na sua utilização. Alguns pinos, no entanto, têm funções dedicadas ou restritas para garantir a operação correta e eficiente do dispositivo como por exemplo os **GND**, **3V3**, **BOOT** e **RESET**.

## 5.2 Driver TC74

O **TC74** é um sensor de temperatura fabricado pela [Microchip Technology](#) que utiliza o protocolo I<sup>2</sup>C([1.2.3](#)) para comunicação, o que facilita a integração com microcontroladores e sistemas embutidos. O Datasheet pode-se encontrar no seguinte [link](#).

A ligação física do **TC74** ao **ESP32-C3-DevKitM-1** é feita diretamente através dos fios de comunicação do barramento I<sup>2</sup>C, que são normalmente partilhados entre múltiplos dispositivos I<sup>2</sup>C.

Isto significa que, para utilizar o TC74, é necessário ligar os fios SDA e SCL do barramento I<sup>2</sup>C ao microcontrolador tanto como a alimentação(3V3) e o ground(GND).

---

<sup>1</sup>Um pino multiplexado quer dizer que existe uma configuração interna do micro-controlador que indica que cada pino físico pode suportar várias funções como SPI, UART, I<sup>2</sup>C, etc.

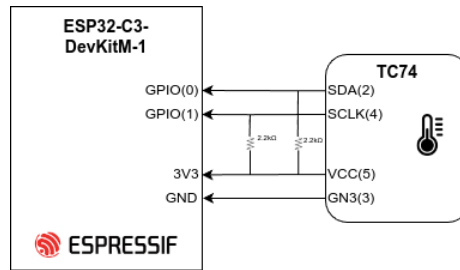


Figura 5.2: Esquema de ligação entre o TC74 e o ESP32-C3-DevKitM-1

### 5.2.1 Driver

O seguinte driver é projetado para inicializar e comunicar-se com um sensor de temperatura **TC74** utilizando o protocolo I<sup>2</sup>C numa plataforma ESP-IDF.

#### 5.2.1.1 tc74\_init:

```

1 esp_err_t tc74_init(i2c_master_bus_handle_t* pBusHandle,
2                     i2c_master_dev_handle_t* pSensorHandle,
3                     uint8_t sensorAddr, int sdaPin, int sclPin,
4                     uint32_t clkSpeedHz) {
5
6     i2c_master_bus_config_t i2cMasterConfig = {
7         .clk_source = I2C_CLK_SRC_DEFAULT,
8         .i2c_port = I2C_NUM_0,
9         .scl_io_num = sclPin,
10        .sda_io_num = sdaPin,
11        .glitch_ignore_cnt = 7,
12        .flags.enable_internal_pullup = true,
13    };
14    ESP_ERROR_CHECK(i2c_new_master_bus(&i2cMasterConfig, pBusHandle));
15
16    i2c_device_config_t i2cDevCfg = {
17        .dev_addr_length = I2C_ADDR_BIT_LEN_7,
18        .device_address = sensorAddr,
19        .scl_speed_hz = clkSpeedHz,
20    };
21
22    ESP_ERROR_CHECK(i2c_master_bus_add_device(*pBusHandle, &i2cDevCfg,
23        pSensorHandle));
24
25    return ESP_OK;
26 }
```

Esta função configura o barramento I<sup>2</sup>C usando os pinos do ESP e a velocidade de comunicação, adicionando o sensor **TC74** ao barramento I<sup>2</sup>C.

#### tc74\_wakeup

```

1 esp_err_t tc74_wakeup(i2c_master_dev_handle_t sensorHandle) {
2     uint8_t buffer[2] = {0x01, 0x00};
3     ESP_ERROR_CHECK(i2c_master_transmit(sensorHandle, buffer, sizeof(
4         buffer), -1));
5     return ESP_OK;
6 }

```

Envia um comando ao sensor para acordá-lo escrevendo 0x00 no registo de configuração 0x01. Após isso, há uma espera para garantir que o sensor estabilize antes de realizar leituras.

#### 5.2.1.2 tc74\_standby:

```

1 esp_err_t tc74_standby(i2c_master_dev_handle_t sensorHandle) {
2     uint8_t buffer[2] = {0x01, 0x80};
3     ESP_ERROR_CHECK(i2c_master_transmit(sensorHandle, buffer, sizeof(
4         buffer), -1));
5     return ESP_OK;
6 }

```

Envia um comando ao sensor para entrar em modo **standby** escrevendo 0x80 no registo de configuração 0x01. Tal como descrito no [Datasheet](#)

#### 5.2.1.3 tc74\_read\_temp\_after\_cfg:

```

1 esp_err_t tc74_read_temp_after_cfg(i2c_master_dev_handle_t sensorHandle
2     , uint8_t* pTemp) {
3     uint8_t txBuf[1] = {0x00};
4     ESP_ERROR_CHECK(i2c_master_transmit_receive(sensorHandle, txBuf,
5         sizeof(txBuf), pTemp, sizeof(uint8_t), -1));
6     return ESP_OK;
7 }

```

Envia um comando ao sensor para ler o registo de temperatura 0x00 e recebe o valor da temperatura.

#### 5.2.2 tc74\_read\_temp\_after\_temp

```

1 esp_err_t tc74_read_temp_after_temp(i2c_master_dev_handle_t
2     sensorHandle, uint8_t* pTemp) {
3     ESP_ERROR_CHECK(i2c_master_receive(sensorHandle, pTemp, sizeof(
4         uint8_t), -1));
5     return ESP_OK;
6 }

```

Lê diretamente um byte do sensor, que se assume ser o valor da temperatura.

# Bibliografia

- [1] L. Melo, *O Modelo de Barramento de Sistema*, <http://arquiteturadecomputadoresalgoritimo.blogspot.com/2015/03/o-modelo-de-barramento-de-sistema.html>.
- [2] Open4Tech, *Direct Memory Access (DMA) in Embedded Systems*, <https://open4tech.com/direct-memory-access-dma-in-embedded-systems/>.
- [3] A. J. Proença, *Arquitetura de Computadores, Notas de estudo*, <http://gec.di.uminho.pt/discip/TextoAC/cap3.html>.
- [4] StonesJunior, *Pendrive x HD x Nuvem: Como usar a hierarquia de memória para gerenciar seus arquivos*, <https://stonesjunior.blogspot.com/2015/12/hierarquia-de-memoria-e-suas-aplicacoes.html>, 2015.
- [5] Open4Tech, *Memory Layout of Embedded C Programs*, <https://open4tech.com/memory-layout-embedded-c-programs/>.
- [6] studytonight, *Storage classes in C*, <https://www.studytonight.com/c/storage-classes-in-c.php>.
- [7] K. Madhumal, *Implement Your Own Operating System (week 05)*, <https://kasun98.medium.com/implement-your-own-operating-system-week-05-92fba4a3c104>, 2021.