# Resumos de ASE Bombadões

# Cenas de C

**Ponteiros e Estruturas de Dados:**

```
typedef struct {
    int data1;
    int data2;
} data_t;
void modify(int* p, void* data) {
    *p = 20;
    data_t* recv = (data_t*)data;
}
int main() {
    int x = 12;
    data_t *send = malloc(sizeof(data_t));
    send->data1 = 10;
    send->data2 = 11;
    int x = 10;
    modify(&x, (void*)send);
    return 0;
}
```

**Alocações de variáveis:**

- **Variáveis Globais**: As variáveis globais são alocadas na memória de dados, que é uma seção da memória reservada para o armazenamento de variáveis globais e estáticas. As variáveis globais podem ser acessadas por qualquer parte do programa e persistem durante toda a vida do programa.
- **Variáveis Estáticas**: As variáveis estáticas, tanto locais como globais, são alocadas na memória de dados, assim como as variáveis globais
- **Variáveis Locais**: As variáveis locais são alocadas na pilha (stack).
- **Alocação Dinâmica** (**Heap**): A alocação dinâmica de memória acontece na heap. Isto ocorre quando se usa funções como malloc(), calloc(), realloc() e free(). Ao contrário da pilha, a alocação e desalocação de memória na heap não é automática, e é responsabilidade do programador garantir que a memória seja adequadamente libertada quando não for mais necessária.

**Macros:**

- Durante a fase de pré-processamento, antes da compilação, o pré-processador C procura a macro no código e substitui todas as suas ocorrências pela definição fornecida.
- **Pré-processamento**: A fase de pré-processamento é a **primeira etapa** no pipeline de compilação. O pré-processador C é responsável pela expansão de todas as diretivas de pré-processamento, que incluem macros, #include e condições de compilação (#if, #ifdef, etc.).
- Armazenamento: As macros não são "armazenadas" no sentido tradicional. Elas existem apenas no código fonte e durante a fase de pré-processamento. Uma vez que a fase de pré-processamento é concluída e a macro é expandida/substituída, a macro em si não existe mais no código que é passado para a próxima etapa da compilação.

**Keyword Static:**

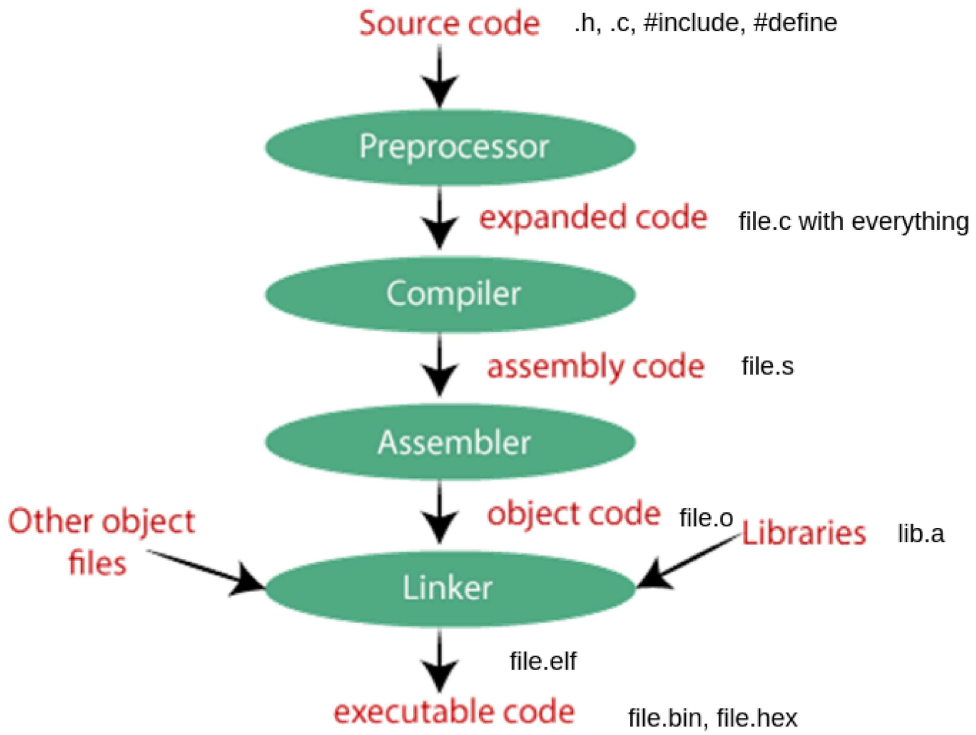
- **Variáveis Locais Estáticas**: Se uma variável local é definida como static, o valor da variável persiste entre chamadas de função. Variáveis locais são geralmente alocadas no stack e são destruídas quando a função retorna. No entanto, as variáveis locais estáticas são alocadas na memória de dados e persistem durante toda a vida do programa.
- **Variáveis Globais Estáticas**: Uma variável global que é definida como static é visível apenas no ficheiro onde é definida. ( se fizermos *extern* da variável é possível “ importar a variável" )
- **Funções Estáticas**: Uma função que é definida como static é visível apenas no ficheiro onde é definida.

**Estrutura da Memória:**

(TOP DOWN ORGANIZATION):

- STACK -> cresce para baixo (variáveis ‘normais’)
- HEAP -> cresce para cima (variaveis de
- DATA -> variáveis estáticas/globais
- TEXT -> codigo em si
- RSVD -> memoria reservada

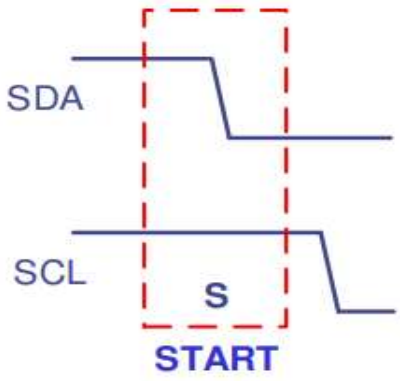
**Processo de compilação do programa:**

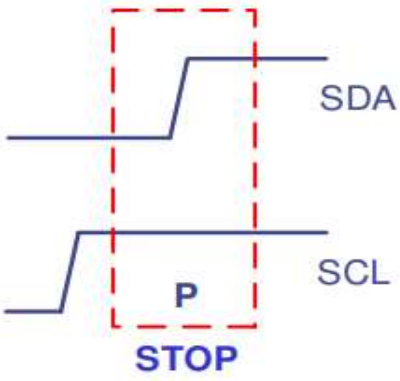



# Protocolos

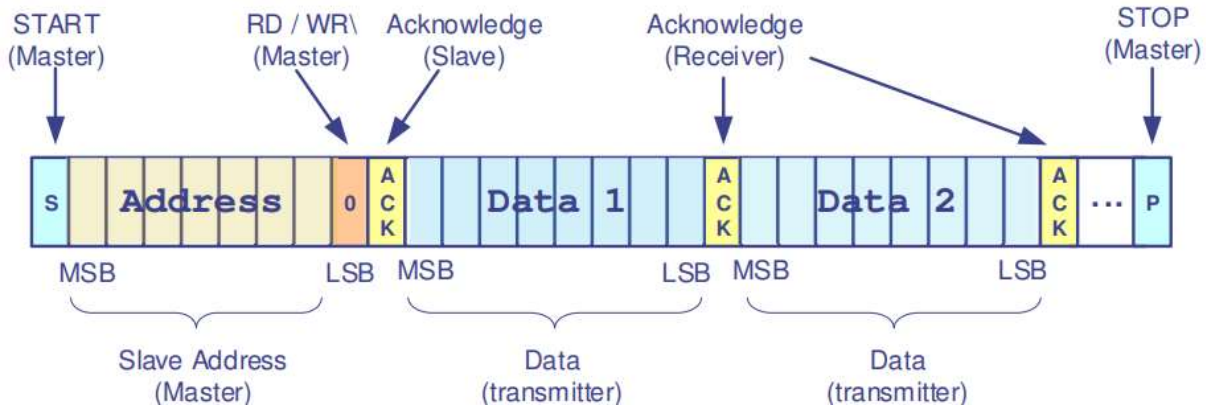
**I2C:**

- 2 linhas de barramento SCL (Clock) e SDA (Dados)
- Protocolo síncrono
- Half-duplex, orientada ao byte
- 128 endereços disponíveis para os slaves ou então 1024, dependendo se tem 7 ou 10 bits de endereçamento (depende do fabricante)
- Linha de dados tanto para ler e escrever nos dispositivos slaves
- Necessita sempre de pull-up resistors
- Manda comandos e Dados



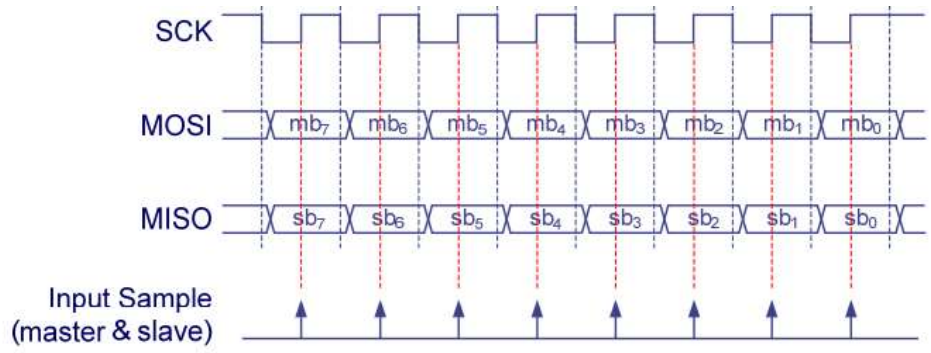
Para dar debug no osciloscópio colocar trigger no flanco descendente (por causa do start bit) no SDA.




**SPI:**

Arquitetura Master-Slave com ligação ponto a ponto (uma ligação slave - master para cada sensor)

- Shift Registers
- Bidirecional (Full-duplex)
- Comunicação Síncrona
    - Clock gerado pelo master e disponibiliza-o para todos os slaves
    - Não necessita precisão do relógio
- Usa 4 linhas SCK (clock), MOSI (Master Out Slave In), MISO (Master In Slave Out) e SS (Slave Select)




- A transição negativa do relógio é usada pelo *master* e pelo *slave* para colocar na respetiva linha de saída um bit de informação
- A transição positiva seguinte é usada pelo *master* e pelo *slave* para armazenar o bit presente na respetiva linha de entrada
- Ao fim de oito ciclos de relógio:
    - o valor inicialmente armazenado no *shift-register* do *master* foi transferido para o *shift-register* do *slave*
    - o valor inicialmente armazenado no *shift-register* do *slave* foi transferido para o *shift-register* do *master*

Para dar debug no osciloscópio colocar trigger no flanco descendente (por causa do start bit) no Slave Select.

**UART:**

- Como não partilham CLK têm de saber o speed de transferências (baud rate)
- UART Frames: Start Bit, Data Bits, Parity Bit, Stop Bit
- Dispositivos com UART têm de ter 2 buffers um para Tx e outro para Rx

  FLUXO SETUP:
- Single Step Conf (Multiple Steps is possible)

```
const uart_port_t uart_num = UART_NUM_2;
uart_config_t uart_config = {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_CTS_RTS,
    .rx_flow_ctrl_thresh = 122,
};
// Configure UART parameters
ESP_ERROR_CHECK(uart_param_config(uart_num, &uart_config));
```

- Set Communication Pins
- Install Drivers, specify:
  - Size of Tx and Rx ring buffer
  - Event queue handle and size
  - Flags to allocate an interrupt
- Run UART Communication:
  - Transmit Data uart_write_bytes
  - Receive Data uart_read_bytes

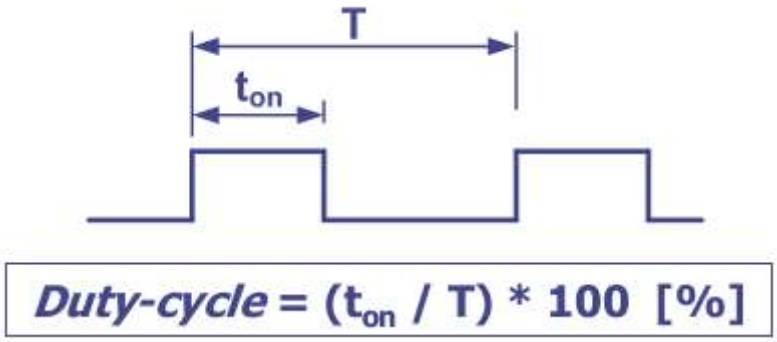
**Interrupt Service Routine:**

Dá apoio quando há um pedido de interrupção

- Uma ISR deve se pequena (sucinta)
- Salvaguarda o contexto (Registos, stack, ...)
- Desativa interrupções (algumas, dependendo da utilização, prioridade, etc.)
- Repor o contexto Callback
- Pode se chamada dentro da ISR pana executar ene uma determinada Tarefa

**PWM:**

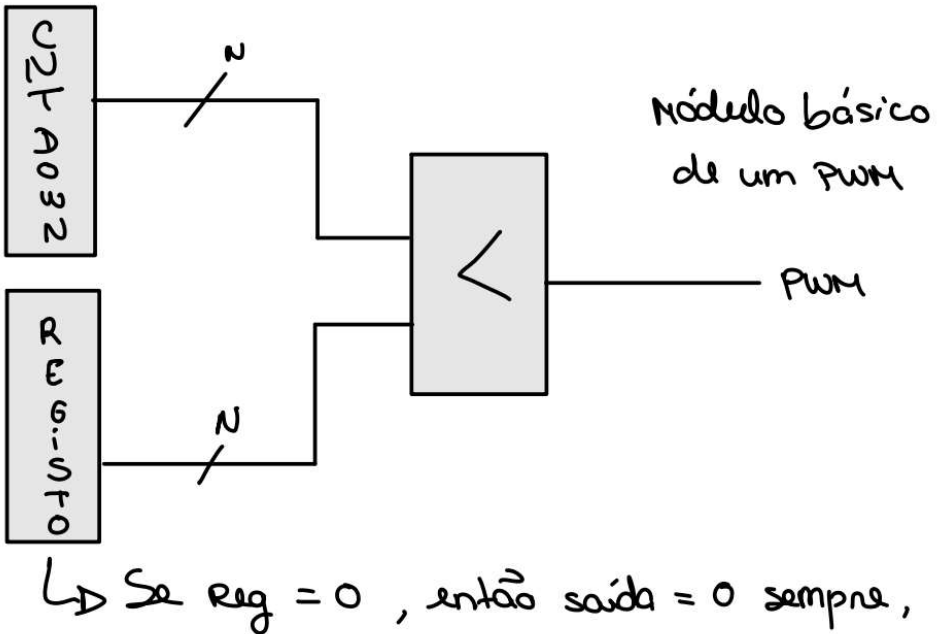
Um timer é um dispositivo periférico que permite a medição de tempo partindo de uma referência de tempo conhecida.




Geração de um evento periódico com período e duração controlados.

T_on = tempo durante o qual o nível lógico está a 1, num período.

A possibilidade de alterar o valor de T_on sem alterar o valor de T é útil em muitas situações e designa-se PWM(Pulse width modulation).\



**ADC:**

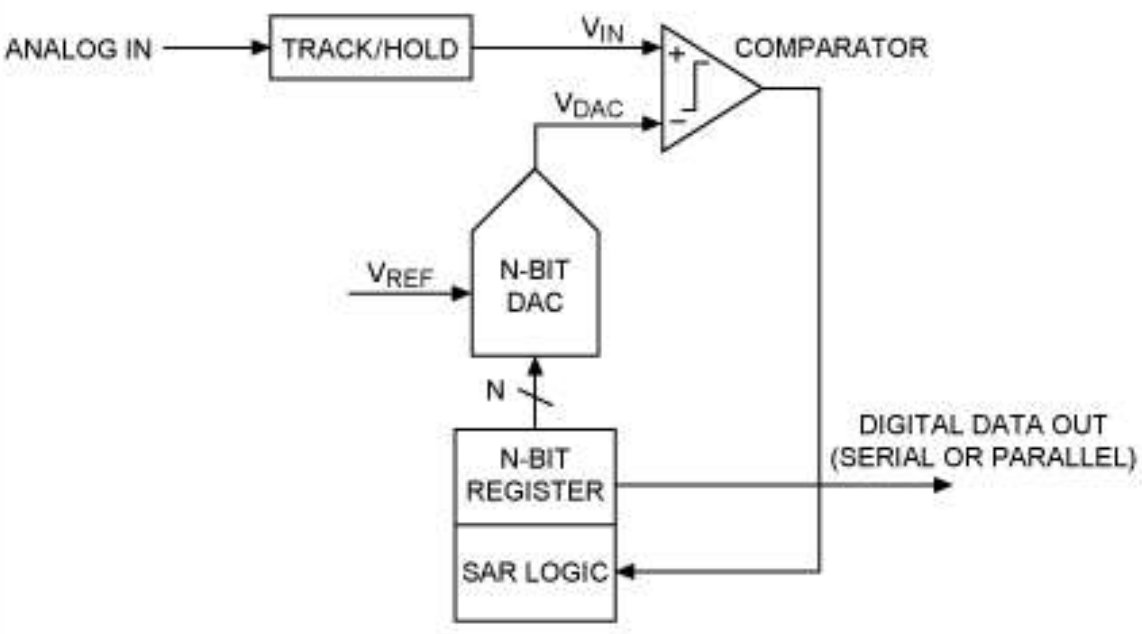
(One shot, Continuous, Calibration? (cagar só))

Fluxo Setup:

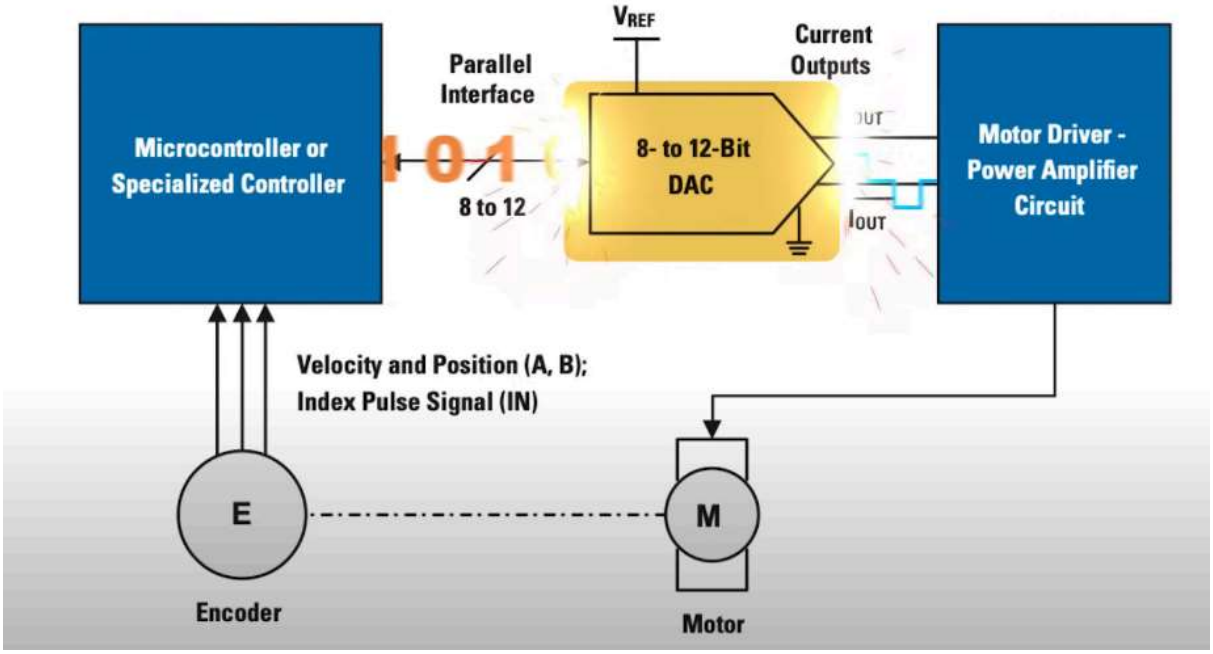
- Resource Allocation: buff size to store ADC conversion
- ADC Configurations: Sample Hz, modo, output format

ADC Control:

- Start
- Register Event Callbacks
  - Conversion Done Event
  - Pool Overflow Event
  - Read Conversion Event
- Stop



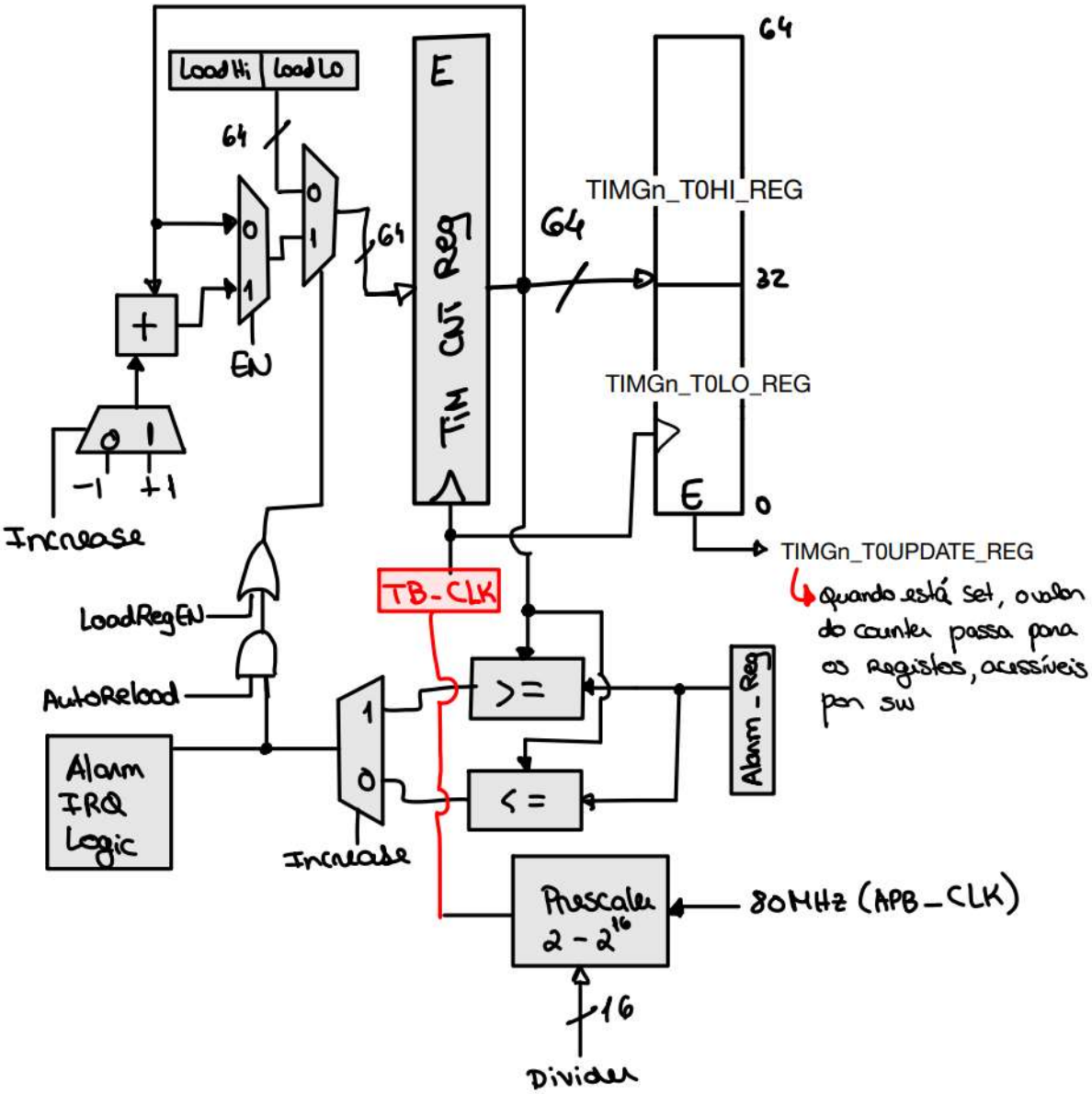
**DAC:**




Não sei quem meteu isto mas obrigado pelo knowledge divino

**Timers:**

- **ESP:** Por software, não pode ter resolução maior que o TICK do sistema, callbacks para interrupções.
  ARQ: Prescaler + Counter + Comparador + Registo com valor a Comparar
  De um modo geral, tem menos precisão, menos resolução que os timers GPT, porque apesar de usar os timers de hardware, leva muito software por cima.




Fluxo Setup:

- Criar timer com func esp_timer_create()
- Definir modo (One-shot ou periodic)
- Callback
- Start Timer

- **GPT**: Hardware Timer, mais flexível, arquitetura toda mamada RIP Trigger One-shot ou Trigger Period Events

No entanto, ao contrário dos timers ESP, apesar de ter uma maior resolução e precisão, é mais difícil de usar e dar setup.

Fluxo Setup

- Resource Allocation: qual clk usar, resolution (kinda like prescaler), …
- Set up Alarm Action
- Register Eventos Callback
- Enable no Timer
- Start Timer

- **Watchdog:**

  Watchdog Timers (WDTs) are crucial components in embedded systems, especially for ensuring reliability and stability. ESP32, a widely used microcontroller for IoT projects, incorporates both Task Watchdog Timer (TWDT) and the Main Watchdog Timer (MWDT). Here's an overview of Watchdog Timers in ESP32:

  **Purpose**:

  Watchdog Timers are used to detect and recover from malfunctions. During normal operation, software periodically resets the watchdog timer, often called "kicking" or "feeding" the watchdog. If, due to a software bug or hardware anomaly, the program hangs, or enters an unintended state, it will stop resetting the watchdog timer. After a predefined time, the watchdog timer expires and performs a corrective action, typically a system reset or a call to a specific interrupt service routine.

  **Task Watchdog Timer (TWDT):**

  - This is mainly used in FreeRTOS tasks on the ESP32. You can set the Task Watchdog Timer to monitor specific tasks.
  - If a task doesn't reset the TWDT within a specified time frame, the watchdog timer concludes that the system has crashed or is not responding, and it takes corrective action.
  - The action taken by the TWDT when it expires can be configured (e.g., reset, interrupt).

**Main Watchdog Timer (MWDT)**:

- The MWDT is designed to be a more general watchdog timer. It's essentially checking if your main program is still running and hasn't crashed.
- Like the TWDT, you have to periodically reset the MWDT. If the MWDT isn’t reset within a certain time, it will take corrective action.
- The MWDT on the ESP32 is hardware-based and is designed to catch issues like deadlocks and infinite loops.

**Configuration**:

- With ESP32, you can configure the watchdog timers to have different timeout values.
- You can also configure what happens when the WDTs expire, such as resetting the microcontroller, raising an interrupt, or logging the issue.
- ESP-IDF, the official development framework for ESP32, provides API functions to work with watchdog timers.

Example Use-Case:

A common use case for WDTs is in systems that require high levels of reliability or in scenarios where the device may be remotely located and manual intervention for a reset is not possible. By using a watchdog timer, the system can attempt an automatic recovery without human intervention.

In conclusion, Watchdog Timers in ESP32 are essential for enhancing system reliability, by monitoring the application for anomalies and taking corrective action in case the application becomes unresponsive or enters an invalid state.

**DMA:**

O problema da (possível) longa latência apresentada pelos periféricos é adequadamente tratada pela técnica de E/S por interrupção.

Esta técnica não resolve, contudo, a questão da transferência a taxas elevadas, uma vez que o limite é sempre imposto pelo facto de o CPU ter que executar um programa para efetuar a transferência.

A solução para mitigar o problema identificado é designada por DMA e consiste na transferência de informação do periférico diretamente para a memória, sem intervenção do processador.

**DMAC:**

Um controlador de DMA é um periférico que, do ponto de vista do modelo de programação, é semelhante a qualquer outro periférico.

Acede a um conjunto de registos internos que o CPU disponibiliza para permitir uma transferência de dados:

- Endereço de leitura
- Endereço de escrita
- Nº de bytes

Durante a transferência o DMAC controla os barramentos de **endereços**, **dados** e **controlo RD, WR** como se fosse um CPU.

Por default o CPU é o bus master.
O DMAC tem que pedir para ser o Bus Master.

**Passos para uma transferência:**

- Lê uma palavra (byte ou word) do dispositivo fonte (do source address) para um registo interno
- Escreve a palavra guardada no registo interno, no passo anterior, no dispositivo destino (no destination address)
- Incrementa source address e destination address
- Incrementa o número de palavras transferidas
- Se não transferiu a totalidade do bloco, repete

Retira o pedido para bus master ser libertado

Para se tornar o BUS MASTER, o DMAC:

Ativa o BUS REQ e assim que o CPU poder libertar o barramento ativa o BUS GRANT.

**Modos de operação:**

**Bloco**: Assume o barramento até todos os dados serem transferidos

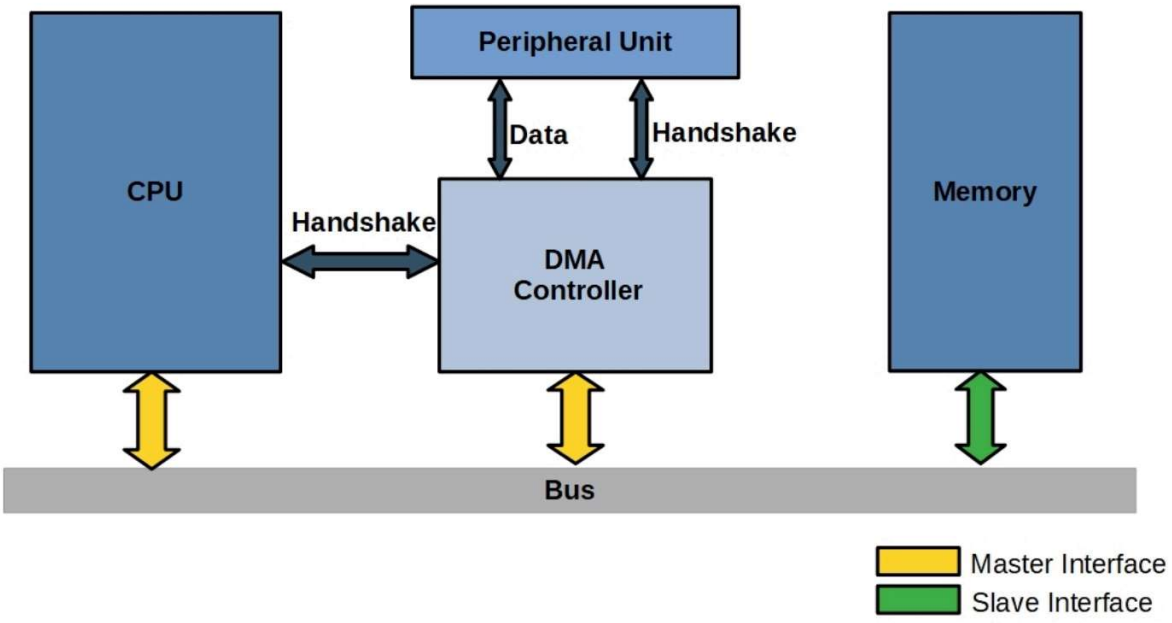
**Burst:** O DMAC transfere até atingir o número de palavras pré-programado ou até o periférico não ter mais informação pronta para ser transferida

**Cycle Stealing**:

O DMAC assume o controlo dos barramentos durante 1 bus cycle e liberta-os de seguida ("rouba" 1 bus-cycle ao CPU) - transfere parcialmente 1 palavra (fetch ou deposit)

- O CPU só liberta os barramentos nos ciclos em que não acede à memória (por exemplo, no estágio MEM de uma instrução aritmética na arquitetura MIPS, pipelined)
- A transferência é mais lenta, mas o impacto do processo de DMA no desempenho do CPU é nulo - o DMAC aproveita os ciclos que não são, de qualquer modo, usados pelo CPU

Um periférico pode ter o seu próprio controlador de DMA (DMAC dedicado).



# FreeRTOS

O FreeRTOS é uma ferramenta popular para sistemas embutidos que pode fazer a gestão de tarefas, e de escalonamento. Além disso, também tem noção de tempo real.

## FreeRTOS Tasks:

Tasks in FreeRTOS are akin to threads in general-purpose operating systems. A task is a small, independent block of code that performs a specific job. In FreeRTOS, tasks are scheduled by the kernel and executed in a pseudo-parallel fashion (on single-core processors).

Each task in FreeRTOS has its own stack and state. The state could be running, ready, or blocked. There are also different priorities associated with tasks. The FreeRTOS scheduler typically uses a preemptive scheduling algorithm but can be configured to use co-operative scheduling.

Tasks can communicate with each other through mechanisms such as queues, semaphores, or events.

## FreeRTOS Delays:

Delays are used in FreeRTOS to block a task for a specified number of ticks (see below). This is useful for timing, synchronisation, and allowing other tasks to execute. There are different types of delays in FreeRTOS:

- vTaskDelay: It delays a task for a specified number of ticks.
- vTaskDelayUntil: It is used to get a task to execute periodically. It delays a task until a specified time.

By using delays, you can effectively allow tasks of lower priority to get processor time, or just make a task wait for some time-critical operations to complete.

## FreeRTOS Ticks:

In FreeRTOS, a tick is a unit of time that is used by the scheduler to switch between tasks. The tick rate is configurable and defines the frequency at which the scheduler's tick interrupt occurs. Every time a tick interrupt occurs, the scheduler can potentially switch to a different task.

The tick rate and mechanism are crucial for real-time performance. It's a trade-off – a high tick rate may allow for more precise scheduling but consumes more CPU cycles in context switching and handling the tick interrupt; a low tick rate may have the opposite effect.
Ticks are also used as a time base for task delays, timeouts, and other time-based functions within FreeRTOS.

# Conceitos Gerais

**O que é um sistema embutido?**

Um sistema embutido é um sistema computacional direcionado a realizar tarefas muito específicas que normalmente necessitam de ser em tempo-real. Ao contrário de um computador que pode realizar diversas tarefas ao mesmo tempo, um sistema embutido tem um limite baixo de tarefas
Um sistema embutido pode ser encontrado em vários tipos de dispositivos e produtos, como:

- Telemóveis
- Sistemas de domótica
- Automóveis
- Aviação
- Eletrodomésticos

Estes sistemas são caracterizados por integrar hardware e software otimizado para realizar as tarefas específicas, considerando limitações como consumo de energia, custo e tamanho.

**O que são Micro Controladores?**

**Explicar o DMA**

DMA, which stands for "Direct Memory Access," is a technique employed in embedded systems that enables hardware devices, like disk controllers or network cards, to access the computer's main memory (RAM) directly without involving the central processing unit (CPU).

**How it works:**

When a device needs to transfer data to or from memory,

1. It sends a request to the DMA controller.
2. The DMA controller then manages the data transfer by coordinating access to the main memory and allowing the device to read from or write directly to that memory. During this time, the CPU is freed up to do something else.
3. Once the data transfer is complete, the DMA controller sends an interrupt signal to the CPU, indicating that the task has been finished.
4. The CPU can then process the data as needed.

**Architecture:**

- I/O Devices: These are the devices requiring memory access to read from or write data.
- DMA Controller: This central component manages the data transfer between the I/O devices and the main memory.
- Data Bus: This physical pathway facilitates data transfer. In many systems, the DMA controller and CPU share the same data bus, but not simultaneously.
- Main Memory: This location stores the data during the transfer.

It's important to note that there are various DMA methods available, such as single-cycle DMA, block DMA, channel DMA, and demand-driven direct memory access (DD-DMA). Each method possesses distinct characteristics and is suited for specific systems and applications.

# Pseudocódigo

**Tarefa 2**

```
Define the constants and PWM configuration.
pwm_init(){
        Set up the PWM timer configuration and apply it.
        Set up the PWM channel configuration and apply it.
}
pwm_set_duty(){
        Set the PWM duty cycle and update it.
```

```
}
app_main(){

        var amplitude, frequency, sampling_period, number_of_samples.

        init array for samples and time.

        generate the time array and samples array.

        Init DAC and ADC.

        Initialize PWM.

        while(true){
                For each sample:
                        Write to DAC.
                        Read from ADC.
                        Adjust ADC reading to match DAC's bit range.
                        Set PWM duty based on ADC reading.
                        Get current time.
                        Write current time, DAC value, and ADC value to log
                wait(sampling_period)
        }
}
```

In essence, the program is controlling a PWM output based on the readings from an ADC which is being fed by a DAC output. The DAC is outputting a sine wave of a specified amplitude and frequency. The PWM, ADC, and DAC are being controlled at a specified sampling rate, and at each sample the current time, DAC output, and ADC reading are being logged.

**EEPROM**

```
Set up constants for the GPIO pins

define SPI device handle

spi_init(){
  Set up and initialize the SPI bus with certain parameters (like clock speed, pin numbers)
  Add the EEPROM device to the SPI bus
}

spi_eeprom_read_byte(){
```

```
  Set up a SPI transaction with the EEPROM's read command and the address we're
interested in
  Tell the SPI bus to send this request to the EEPROM and get the result
  return the result
}

spi_eeprom_write_byte(){
  Set up a SPI transaction with the EEPROM's write command, the address we're interested
in, and the data we want to write
  Tell the SPI bus to send this request to the EEPROM
}

app_main(){
  Start the SPI bus and connect to the EEPROM using spi_init

  Choose an address in the EEPROM and some data we want to write there

  Spi_eeprom_write_byte

  Spi_eeprom_read_byte
}
```

**UART**

```
setup_uart() {
        uart_config_t uart_config {
                baud_rate,
                data_bits,
                parity,
                stop_bits,
                …
        }
        uart_param_config(uart_config)
        uart_set_pin()
        uart_set_driver_install()
}
// use uart_read_bytes from espressif
get_char() {
        while(1) {
                len = uart_read_bytes(data)
                if (len > 0)
                        return data
        }
}
// use get_char from above
```

```
get_string(data) {
        while((c = get_char()) != '\n') {
                data[i++] = c;
        }
}
```

**Contributos de:**

João ***Pexugo Teludo*** Viegas

André ***Génis Crande de Paracol*** Clérigo

Tiago ***Areguiça Smbulante Pingela*** Marques

Bruno ***Enguia Elétrica*** Lemos

Hugo ***Sanana Bplit*** Domingos

Claudio ***Lodi da Jena*** *Ascenção*

Pedro ***Mirgem Vacho*** Rocha

Diogo ***Razi Nacista*** Correia

João ***VO TE CHUPAR TU PALO*** **AMARELO** Amaral

Joana ***LA CHUPADORA DE DECABANANA*** Cunha

Diana ***EL CULO DE MELANCIA(zinha xd)*** Rocha  bruvvv wut? Grizei que fode