

INSTITUTO MAUÁ DE TECNOLOGIA



# Sistemas Operacionais de Tempo Real

Profº. Tiago Sanches da Silva

# Avaliação

- Projeto em grupo e apresentação do mesmo no último dia de aula da disciplina;
- Atividades para entrega;
- Atividades em laboratório.

# **FreeRTOS**



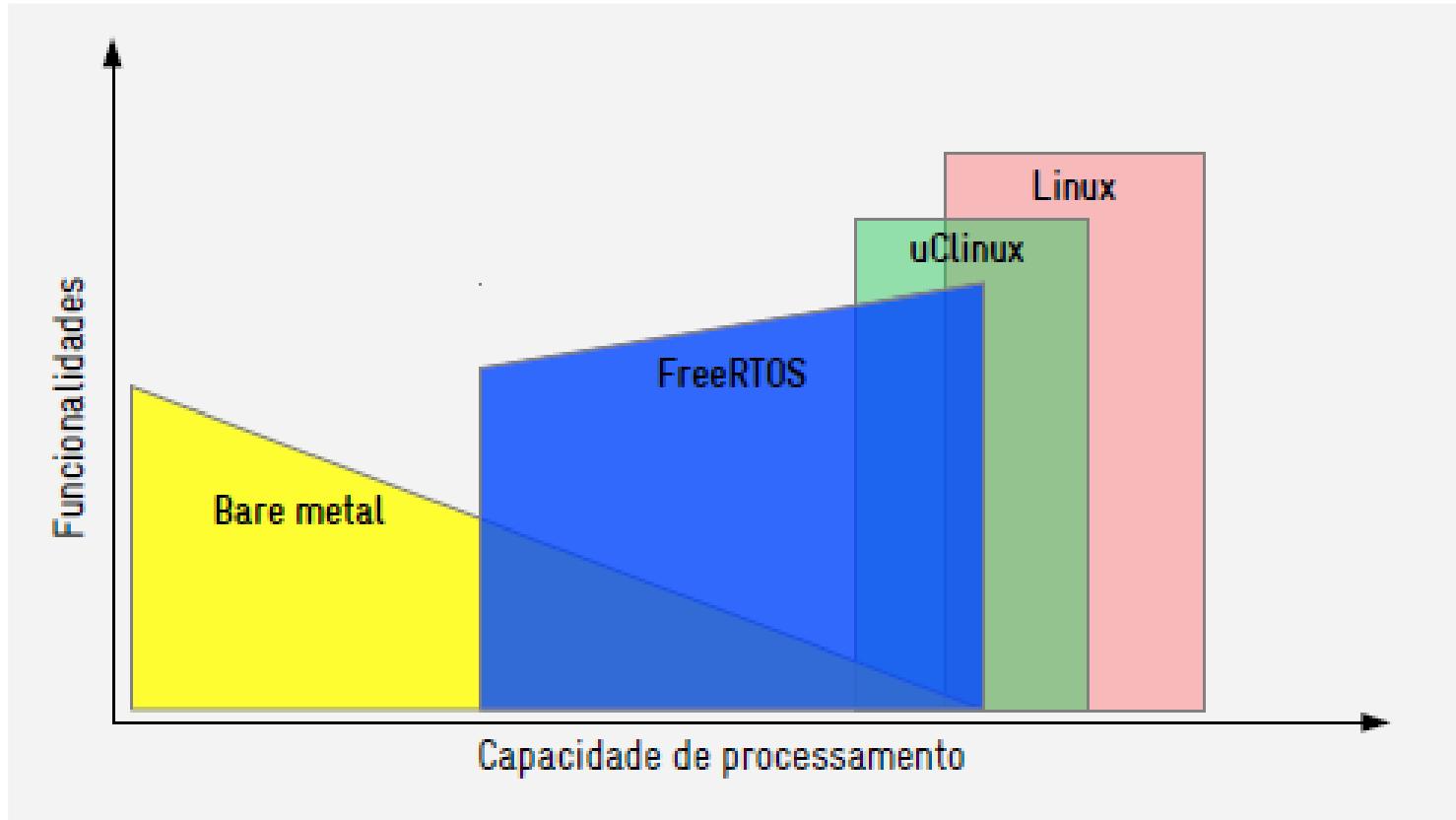
- Introdução

Criado por volta do ano 2000 por Richard Barry, e hoje mantido pela empresa Real Time Engineers Ltd.

- RTOS de código aberto mais utilizado no mundo;
- É simples, pequeno e extremamente portável;
- Site do projeto, com muita documentação disponível:  
<http://www.freertos.org/>
- Código-fonte pode ser baixado em:  
<http://sourceforge.net/projects/freertos/files/>



- Posição no mercado



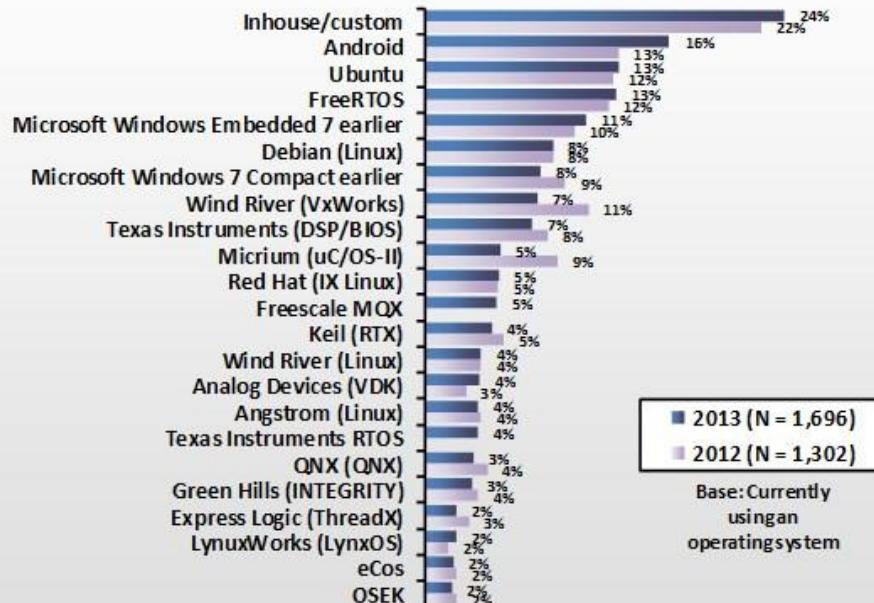


- Posição no mercado

2013 Embedded Market Study

23

Please select **ALL** of the operating systems you are currently using.



Only Operating Systems that had 2% or more are shown.



- Posição no mercado

Presented by UBM Electronics Group



April 2015



© 2015 Property of UBM Canon; All Rights Reserved



- Posição no mercado

2015 UBM Electronics Embedded Markets Study

## Purpose and Methodology

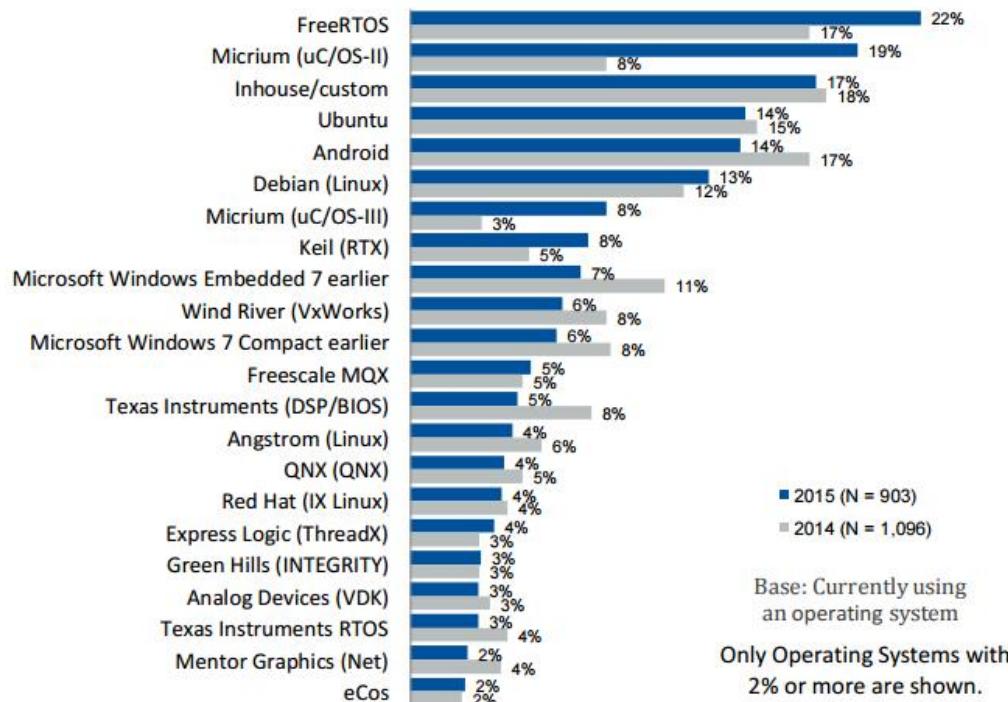
- **Purpose:** To profile the findings of the 2015 results of UBM Tech's annual comprehensive survey of the **embedded systems markets worldwide**. Findings include types of technology used, all aspects of the embedded development process, IoT emergence, tools used, work environment, applications, methods/ processes, operating systems used, reasons for using chips and technology, and brands and chips being considered by embedded developers. Many questions in this survey are trended over three to five years.
- **Methodology:** A web-based online survey instrument based on the previous year's survey was developed and implemented by independent research company Wilson Research Group from January 14, 2015 to March 31, 2015 by email invitation.
- **Sample:** E-mail invitations were sent to subscribers to UBM Tech Embedded Brands with reminder invitations sent later. Each invitation included a link to the survey.
- **Returns:** 1,807 valid respondents for an overall confidence of 95% +/- 2.29%. Confidence levels vary by question. As a guide, confidence for questions with:
  - 1807 respondents = 95% +/- 2.29% vs. 95% +/- 2.05% in 2014
  - 1050 respondents = 95% +/- 3.0%
  - 600 respondents = 95% +/- 4.0%
  - 400 respondents = 95% +/- 5.0%



- Posição no mercado

2015 UBM Electronics Embedded Markets Study

Please select ALL of the operating systems you are  
currently using



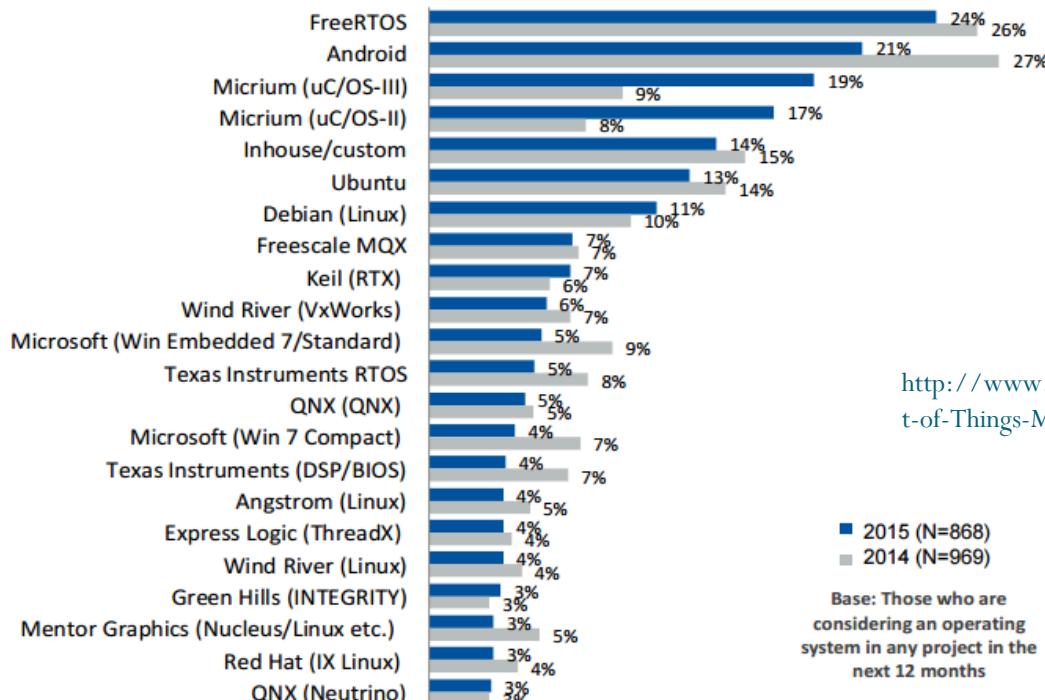
FreeRTOS was 30%,  
Micrium uC/OS-II was  
29% and Micrium  
uC/OS-III was 13% in  
**Asia**, influencing the  
ranking of this years  
OS leaders.



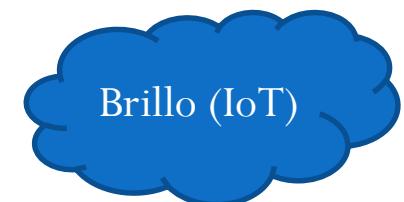
- Posição no mercado

2015 UBM Electronics Embedded Markets Study

Please select ALL of the operating systems you are considering using in the next 12 months.



<http://www.engineering.com/IOT/ArticleID/11307/Internet-of-Things-Microcontroller-is-Google-Weave-Ready.aspx>





- ## Características

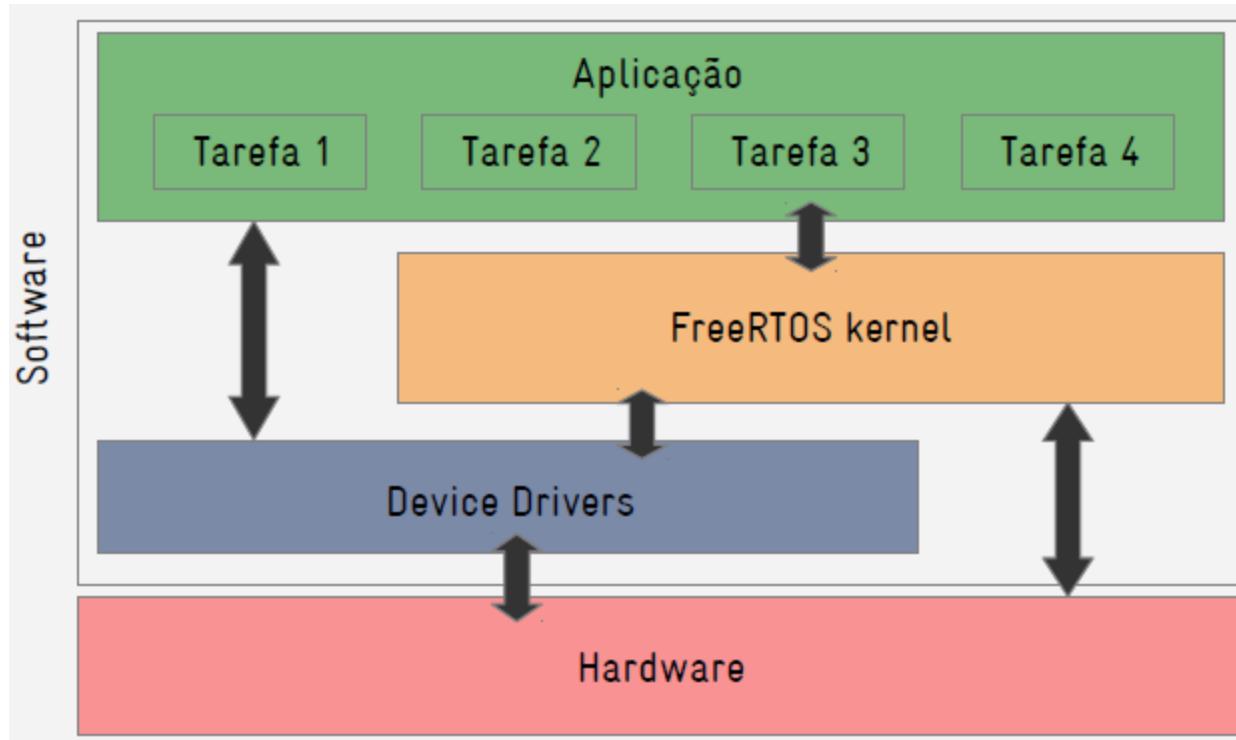
- Projetado para ser pequeno, simples e fácil de usar;
- Escrito em C, extremamente portável;
- Suporta 35+ arquiteturas diferentes;
- Em uma configuração típica, o kernel do FreeRTOS pode ocupar de 4KB a 9KB de código (ROM/flash) e em torno de 200 bytes de dados (RAM);



- Características
  - Kernel pode trabalhar de forma preemptiva ou colaborativa;
  - Mutex com suporte à herança de prioridade;
  - Capacidades de trace e detecção de stack overflow;
  - Sem restrição de quantidade de tarefas que podem ser criadas, ou da quantidade de prioridades que podem ser usadas;
  - Comunidade ativa.

# FreeRTOS

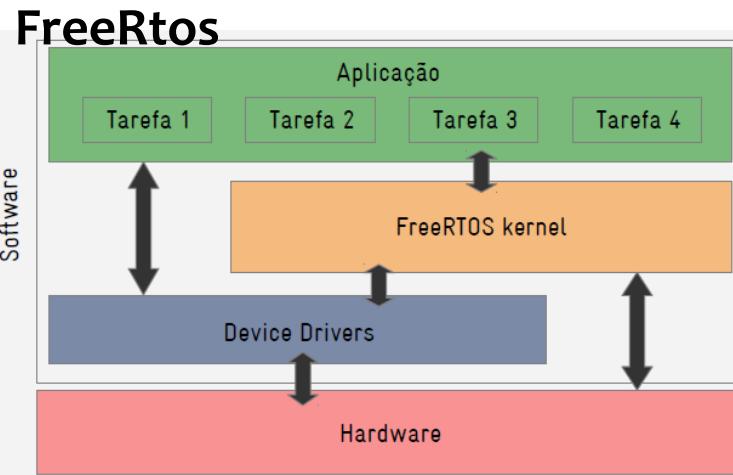
- Arquitetura



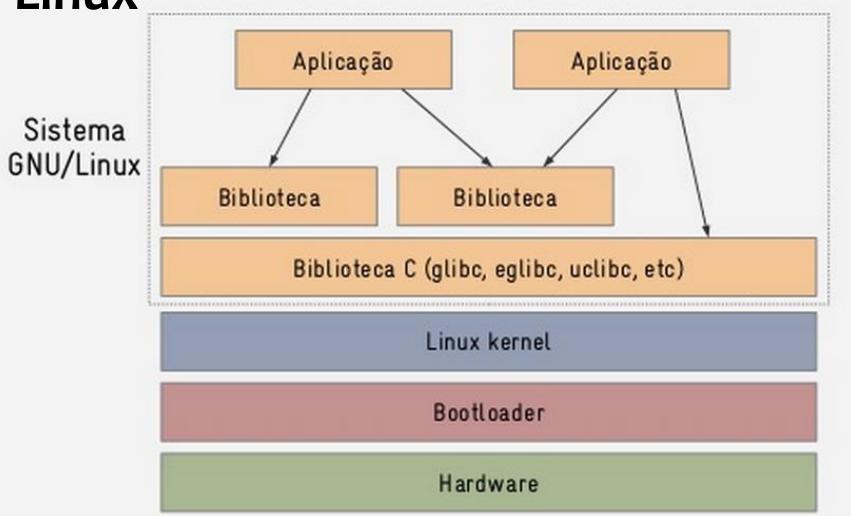
# FreeRTOS



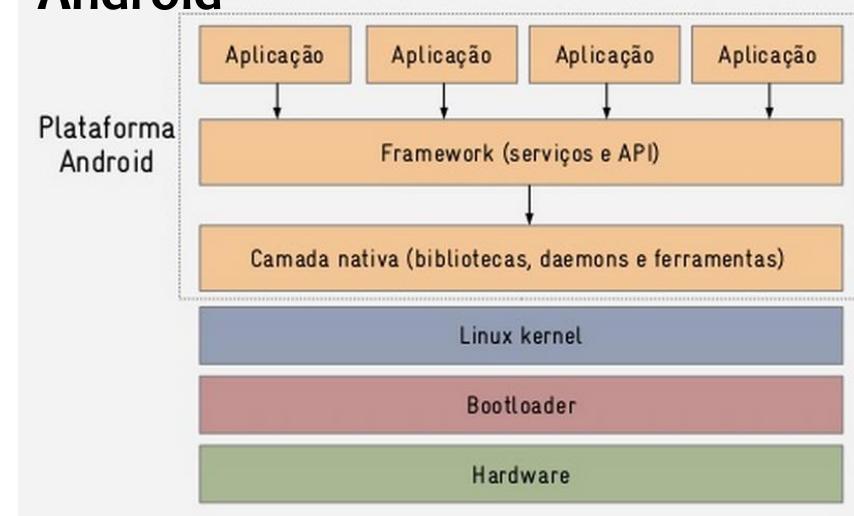
- Arquitetura - comparando



### Linux



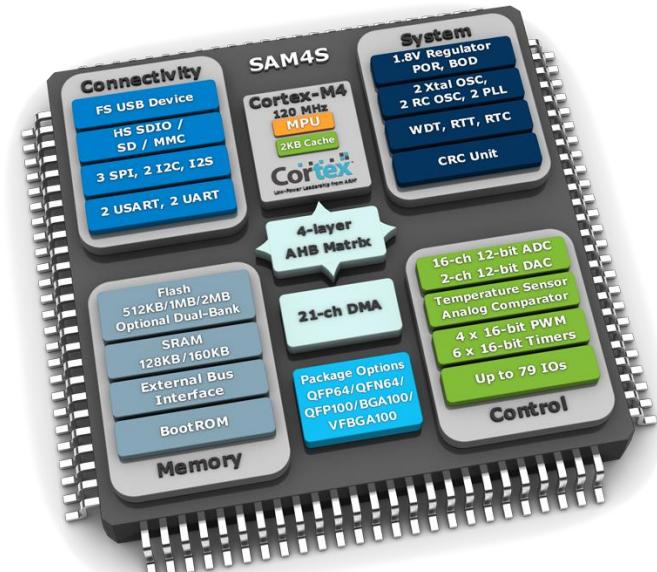
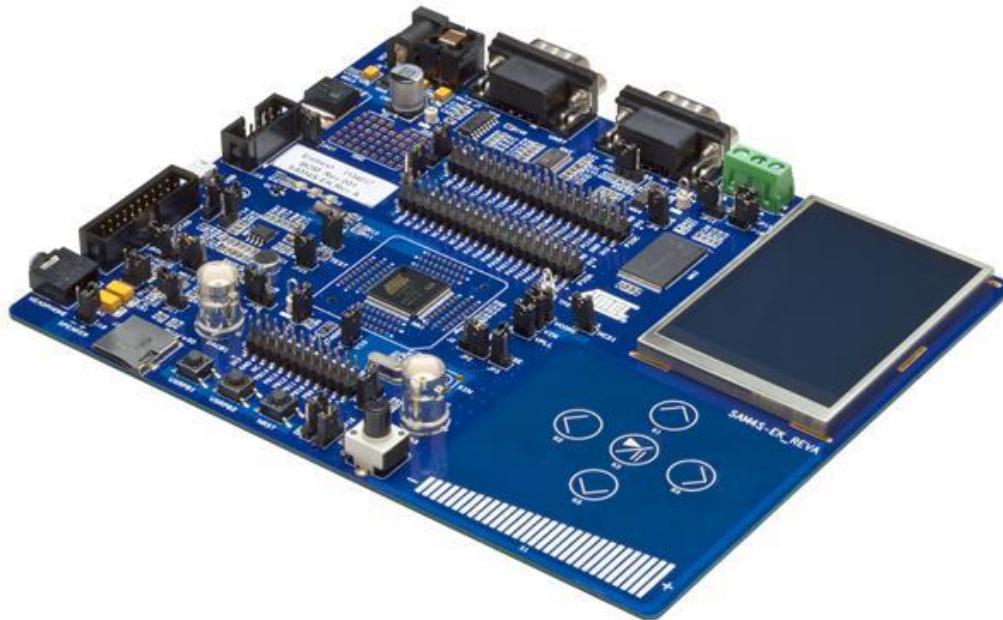
### Android



# **Kit de desenvolvimento**



# Kit de desenvolvimento



## Atmel AT91SAM

- Kit de desenvolvimento da Atmel
- SOC SAM4SD32 (ARM Cortex M4)

# **Sistemas de Tempo Real**

# SISTEMAS DE TEMPO REAL

## Introdução

No contexto de desenvolvimento de software, um sistema é projetado para receber estímulos (ou eventos), internos ou externos, a partir destes estímulos realizar um processamento e produzir uma saída.

Alguns sistemas trabalham com eventos que possuem **restrições temporais**, ou seja, possuem um prazo limite para que o estímulo seja processado e gere uma saída.

Estes sistemas são chamados: “**Sistemas de Tempo Real**”.

Um Sistema Operacional de Tempo Real é, portanto, o **software que gerencia os recursos** de um sistema computacional, com o objetivo de garantir com que todos os **eventos sejam atendidos** dentro de suas **restrições de tempo**.

# SISTEMAS DE TEMPO REAL

## Introdução

Características:

- São sistemas que monitoram, respondem ou controlam um **ambiente externo**. Esse **ambiente** normalmente está conectado ao sistema através de sensores e atuadores;
- Esse sistema deve satisfazer a **restrições temporais** e outras restrições impostas pelo comportamento de tempo real do **mundo externo**.



# SISTEMAS DE TEMPO REAL

## Introdução

Portanto, um sistema de tempo real precisa garantir que todos os eventos sejam atendidos dentro das suas respectivas restrições de tempo.

É por isso que um sistema de tempo real está relacionado ao **determinismo**, e não a velocidade de execução.

Existem basicamente dois tipos de sistemas de tempo real, classificados de acordo com a tolerância às restrições de tempo:

- Soft Real-Time Systems
- Hard Real-Time Systems

# SISTEMAS DE TEMPO REAL

## Classificação

**Soft real-time:** Caso a operação não seja realizada dentro do prazo limite, ainda existe valor para saída gerada.

Exemplos:

- *Display touch* que demora para responder ao tocar na tela
- Envios de mensagem com atraso
- Sistema de comutação telefônico



**Hard real-time:** Uma restrição de tempo não atingida pode inutilizar o sistema ou provocar consequências catastróficas.

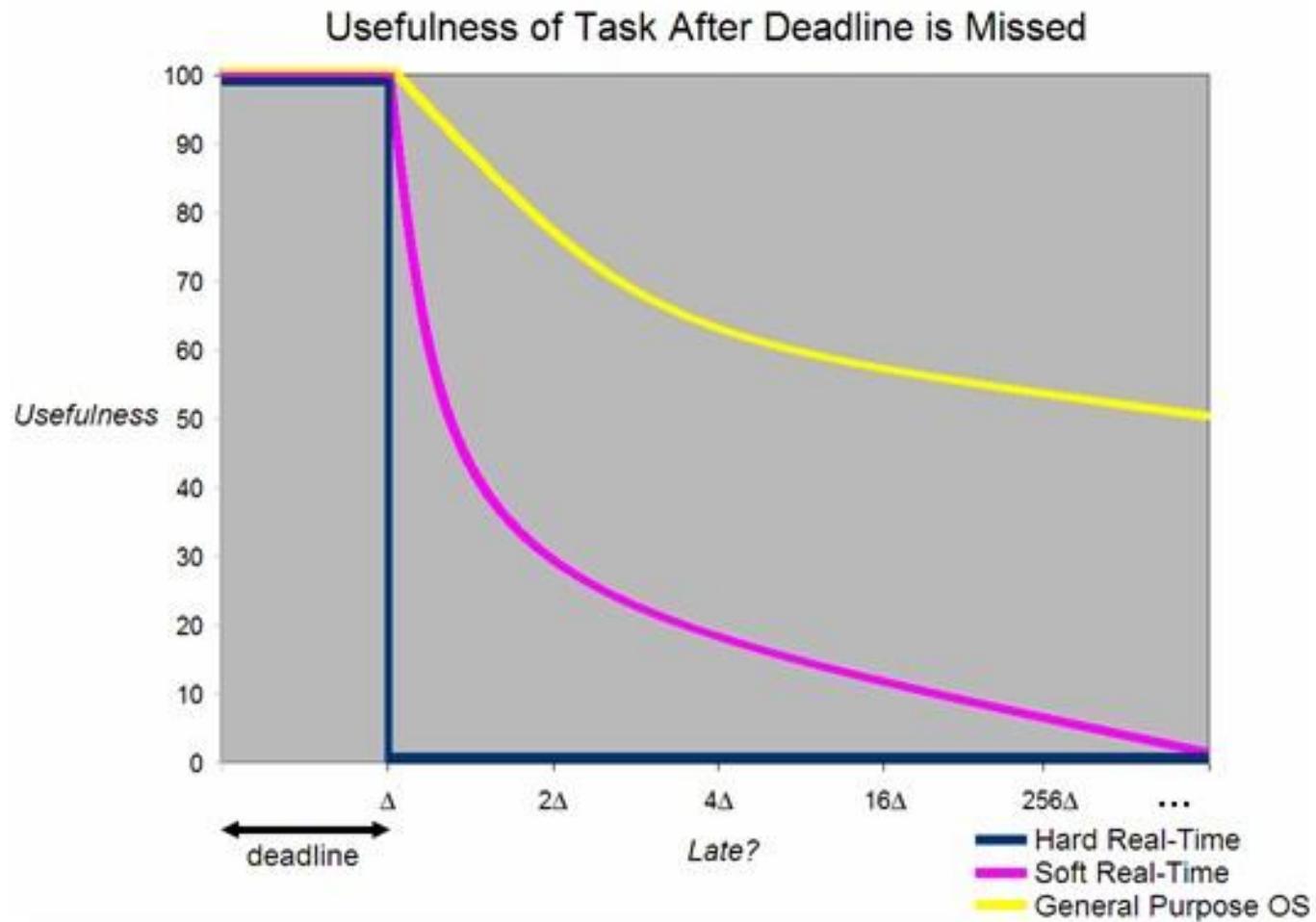
Exemplos:

- Sistema de controle de voo
- Sistema de acionamento de airbag do carro
- Controle de um trem autônomo



# SISTEMAS DE TEMPO REAL

## Classificação



# SISTEMAS DE TEMPO REAL

## Desempenho em tempo real

**“Quanto mais rápido melhor?”**

O equívoco mais comum associado ao desempenho de um sistema operacional de tempo real é dizer que ele aumenta a velocidade de execução do programa.

A aplicação é melhorada proporcionando temporização precisa e previsível das tarefas.

**Previsibilidade**

**Precisão**

**Determinismo**

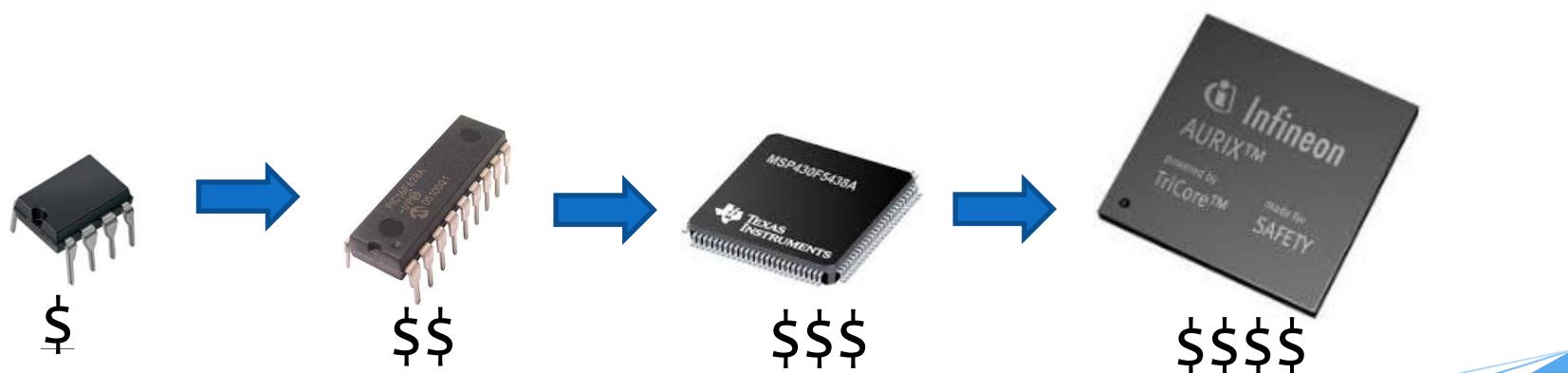
# SISTEMAS DE TEMPO REAL

## Desempenho em tempo real

**“Ok... Mas... Quanto mais rápido melhor?”**  
**Não!**

O mais importante para um sistema de tempo real é o seu **determinismo**.

O sistema deve operar dentro dos tempos especificados pelo projeto, normalmente trabalhar mais rápido não trará vantagens ao sistema, e pode ser um indício de superdimensionamento dos recursos do hardware escolhido. Não jogue dinheiro fora!



# SISTEMAS DE TEMPO REAL

## Desempenho em tempo real

**“Entendi, mas...**

**Se estiver mais rápido não prejudica, certo!?”**

**Sistemas de controle de tempo real, pode prejudicar!**

Por interfaciar com sensores e atuadores, muitos sistemas são sensíveis a frequência de operação;

Sensores e atuadores possuem tempo de respostas mínimos e máximos que devem ser respeitados pelo controlador.

Sistemas de controle em tempo real precisão de cuidado extra para determinar a frequência da malha de controle.

# CONTROLE EM TEMPO REAL

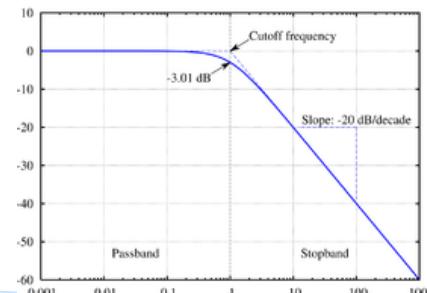
## Conceitos gerais

Com controle em tempo real, é possível monitorar e simular continuamente um sistema físico;

Aplicações de controle em tempo real executam repetidamente uma tarefa definida pelo usuário com um intervalo de tempo específico entre cada execução;

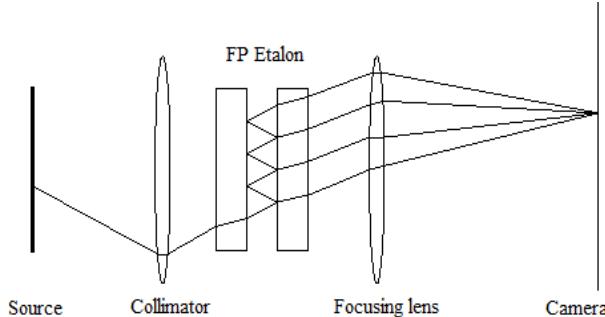
O tempo que leva para que essa malha execute é considerado o tempo de ciclo da malha.

**E por que a frequência de atuação na malha é algo crítico?**





## Exemplo: Interferômetro Fabry Perot



- Deseja-se controlar a distância entre as duas lentes para filtrar um determinado comprimento de onda.
- A frequência que a malha de controle atua no atuador de posição do sistema, terá impacto direto no funcionamento e na estabilidade do sistema.

$\lambda$ : comprimento de onda

$$\lambda = \frac{2ne \cos(i)}{p}$$

$n$ : índice de refração

$e$ : distância entre os vidros

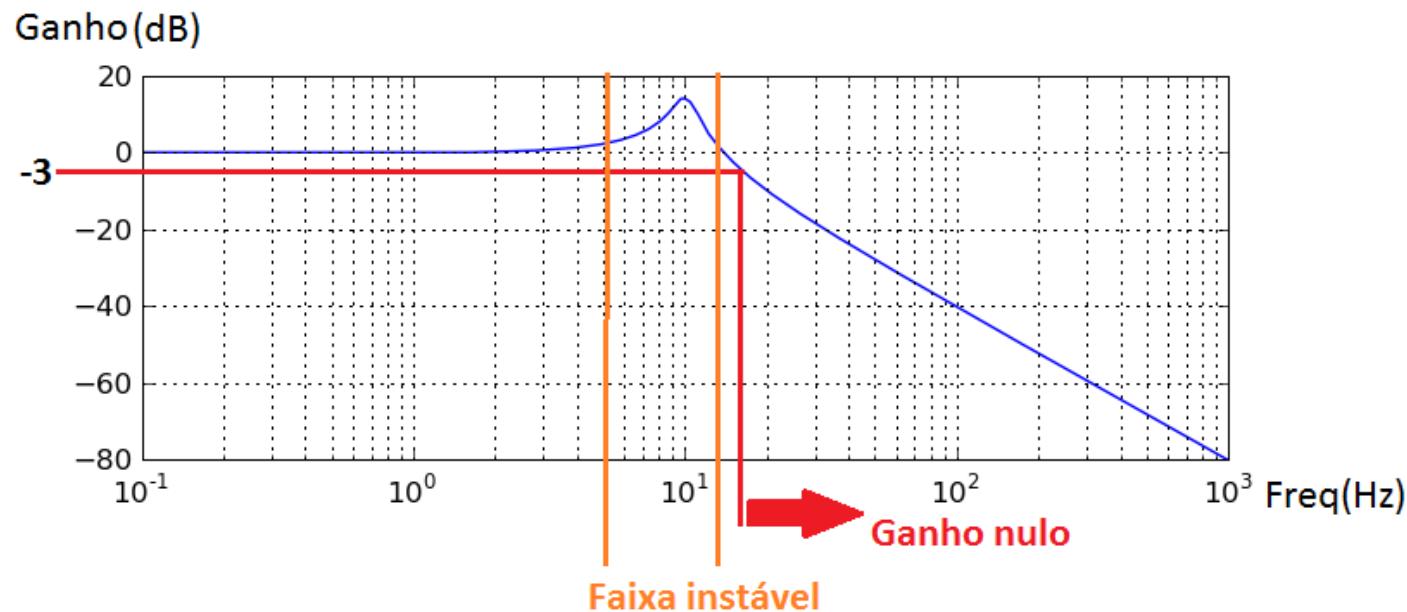
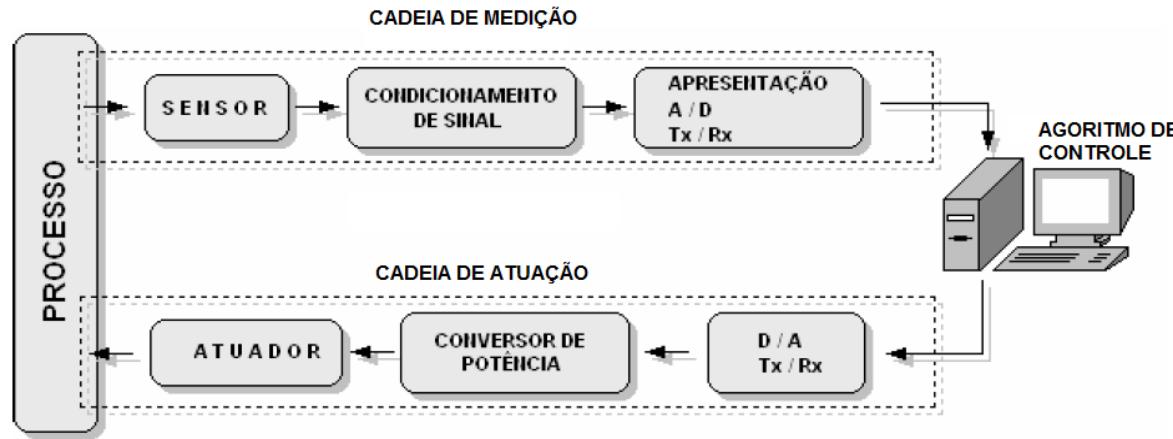
$i$ : ângulo de incidência

$p$ : ordem de interferência

# CONTROLE EM TEMPO REAL



## Exemplo: Interferômetro Fabry Perot



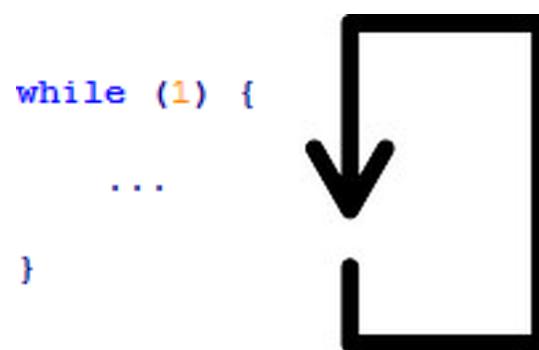
# **Sistema Operacional de Tempo Real**

# Sistemas de Tempo Real

## Super Loop - ForeGround/BackGround

Projetos pequenos e de baixa complexidade são desenvolvidos como sistemas *foreground/background* (também chamados de *super loop*).

- A aplicação consiste em um loop infinito que chama algumas funções para realizar as operações desejadas (*background*).
- E rotinas de tratamento de interrupção tratam eventos assíncronos (*foreground*).



# Sistemas de Tempo Real

## Super Loop - ForeGround/BackGround

```
int main(void)
{
    init_hardware();

    while (1)
    {
        Audio_Play();
        UART_Receive();
        USB_GetPacket();
        ADC_Convert();
        LCD_Manage();
        ...
    }
}
```

**BACKGROUND**

```
void USB_ISR(void)
{
    Limpa_interrupção;
    Lê pacote;
}

void UART_ISR(void)
{
    Limpa_interrupção;
    Trata byte recebido;
}
```

**FOREGROUND**

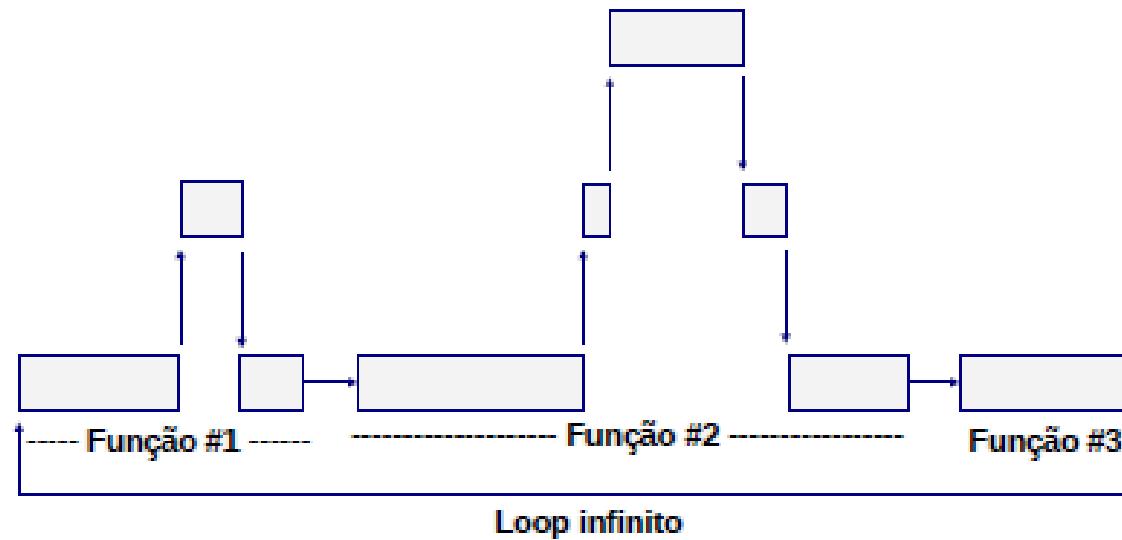
# Sistemas de Tempo Real

## Super Loop - ForeGround/BackGround

Foreground (2)

Foreground (1)

Background



# Sistemas de Tempo Real

## Super Loop - Vantagens

- Facilidade no desenvolvimento e na implementação;
- Não requer treinamento em API's específicas de um sistema operacional;
- Não consome recursos adicionais do microcontrolador;
- Ótima solução para projetos pequenos e com requisitos modestos de restrições de tempo;
- Não requer entendimento de processamento paralelo.

# Sistemas de Tempo Real

## Super Loop - Deficiências

- Difícil garantir que uma operação irá ser executada dentro das restrições de tempo;
- Todo o código em background tem a mesma prioridade;
- As funções de background são executadas sequencialmente;
- Se alguma função demorar mais do que o esperado, todo o sistema será diretamente impactado.

# Sistemas de Tempo Real

## Super Loop - Deficiências

```
while (1) {  
    ADC_Read();  
    UART_Receive();  
    USB_GetPacket();  
    ...  
}
```

```
void ADC_Read (void)  
{  
    configure_ADC();  
    while (conv_rdy == 0)  
    {}  
    ...  
}
```

- Espera o término da conversão do módulo do ADC para continuar a execução do código;
- Isso acarretará em um atraso de todo o sistema de background.

## Super Loop - Deficiências

- Tarefas de alta prioridade precisam ser colocadas em *foreground* (ISR);
- ISRs muito longas podem impactar o tempo de resposta do Sistema;
- É difícil coordenar a execução de rotinas em *background* e em *foreground*.

# Sistemas de Tempo Real

## Super Loop - Deficiências

- Qualquer alteração em determinada parte do código pode impactar o tempo de resposta de todo o sistema;
- Difícil de garantir as restrições de tempo da aplicação;
- Problemas podem aparecer quando o código é mantido por múltiplos desenvolvedores. Como controlar?

É preciso uma solução que gerencie corretamente os requisitos de tempo real do sistema.

Precisamos de um **kernel de tempo real!**

# **Kernel de Tempo Real**

# Kernel de Tempo Real

## Introdução

- Um kernel de tempo real é um software que gerencia o tempo e os recursos da CPU, e é baseado no conceito de tarefas e prioridades;
- Todas as funcionalidades do sistema são divididas em tarefas (*tasks*);
- O *kernel* decide quando uma tarefa deve ser executada baseado na prioridade da tarefa;
- Fica sob responsabilidade do desenvolvedor dividir o sistema em tarefas e definir as prioridades de acordo com as características de tempo real de cada uma delas.

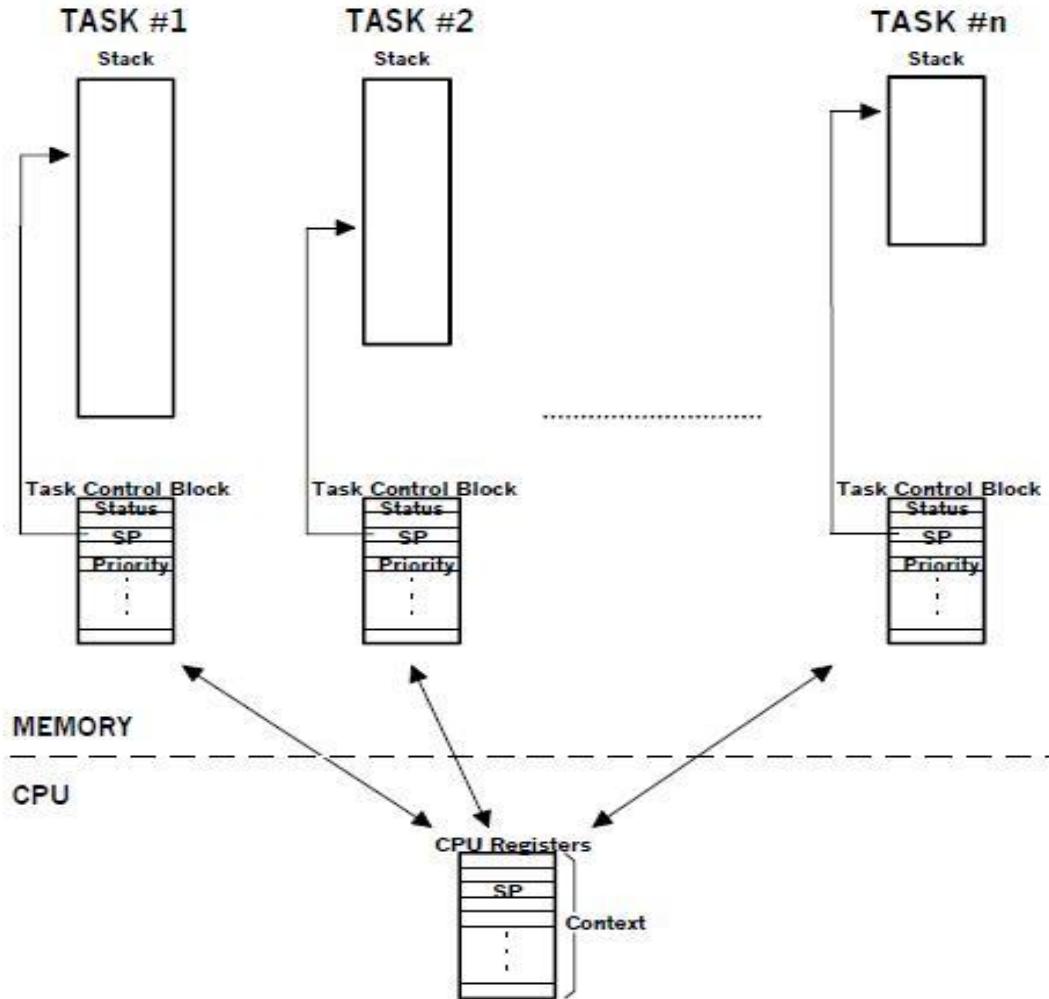
# Kernel de Tempo Real

## Conceito de Tarefa (Task)

- Uma tarefa é um programa simples que pensa que tem toda a CPU para si próprio;
- O procedimento de projeto para um sistema de tempo real envolve dividir o trabalho a ser feito em tarefas as quais são responsáveis por uma porção do problema;
- Para cada tarefa é atribuída uma prioridade, seu próprio conjunto de registradores da CPU e uma área de pilha (*stack*);

# Kernel de Tempo Real

## Multitarefa



# Kernel de Tempo Real

## Multitarefa

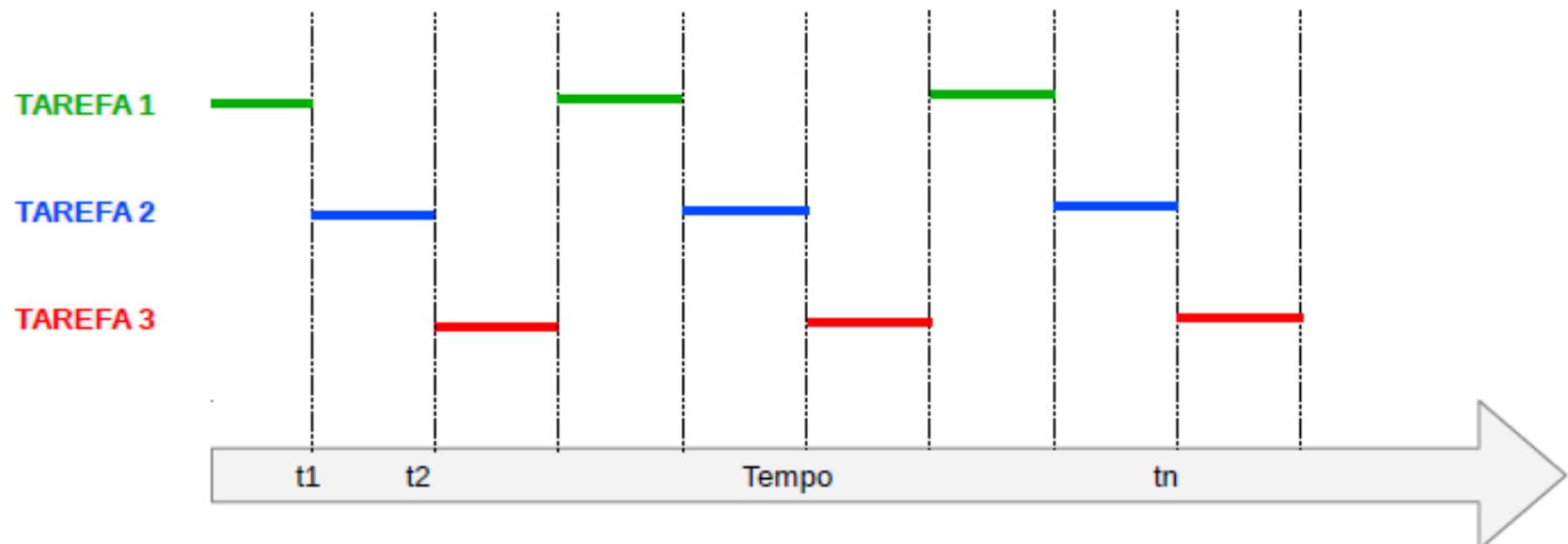
- Em um sistema multitarefa, temos a impressão de que todas as tarefas estão sendo executadas ao mesmo tempo.



# Kernel de Tempo Real

## Multitarefa

- Mas como o processador só pode executar uma tarefa de cada vez (considerando-se uma CPU com apenas um núcleo), é realizado um chaveamento entre as tarefas.



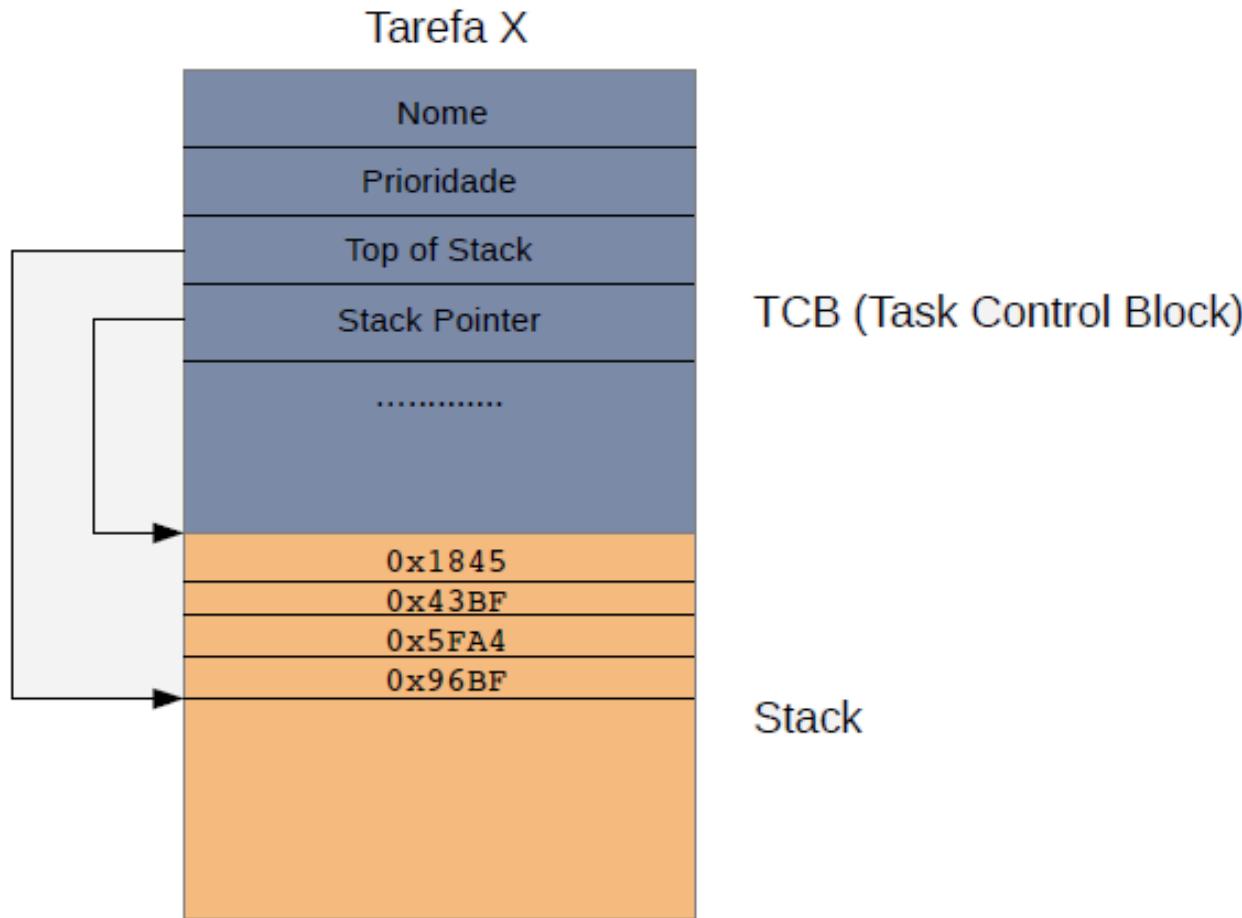
# Kernel de Tempo Real

## Mudança de Contexto

- Enquanto uma tarefa está em execução, ela possui determinado contexto (*stack*, registradores da CPU, etc);
- Ao mudar a tarefa em execução, o *kernel* salva o contexto da tarefa a ser suspensa e recupera o contexto da próxima tarefa a ser executada;
- O controle do contexto de cada uma das tarefas é realizado através de uma estrutura chamada TCB (*Task Control Block*).

# Kernel de Tempo Real

## TCB (Task Control Block)



# Kernel de Tempo Real

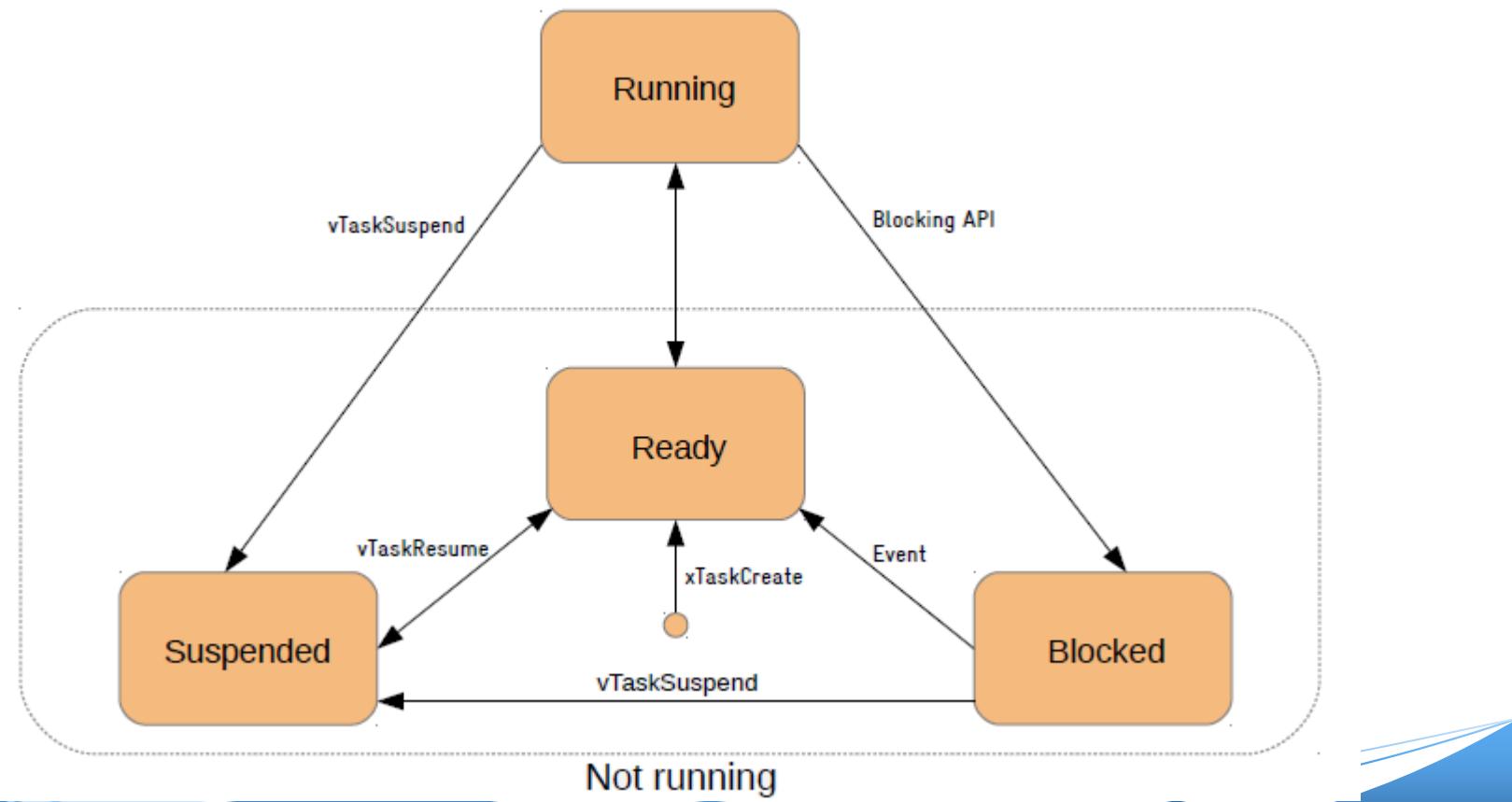
## Trabalhando com tarefas

- Uma aplicação pode conter diversas tarefas;
- Em um microcontrolador com apenas um core, apenas uma tarefa está em execução;
- Portanto, isso significa que uma tarefa pode estar em dois estados: “Running” (executando) ou “Not running” (não executando, parada). Veremos mais para frente que o estado “Not running” possui alguns sub-estados;
- Esta troca de estados é realizada pelo *scheduler* ou escalonador de tarefas.

# Kernel de Tempo Real

## Estado de uma tarefa

- Cada tarefa é um loop infinito que pode estar em de cinco estados: READY, RUNNING, BLOCKED, SUSPENDED.



# **UM BREVE RESUMO SOBRE MICROCONTROLADORES**



- Introdução

- Um Microcontrolador é um sistema computacional completo inserido em um único circuito integrado;
- Possui CPU, memória de dados RAM e ROM para manipulação de dados e armazenamento de instruções, sistema de *clock* para dar sequência às atividades da CPU, portas de I/O, além de outros possíveis periféricos como módulos de temporização, conversores analógico-digital e até mesmo nos mais avançados conversores USB (*Universal Serial Bus*) ou ETHERNET.



- Estrutura básica

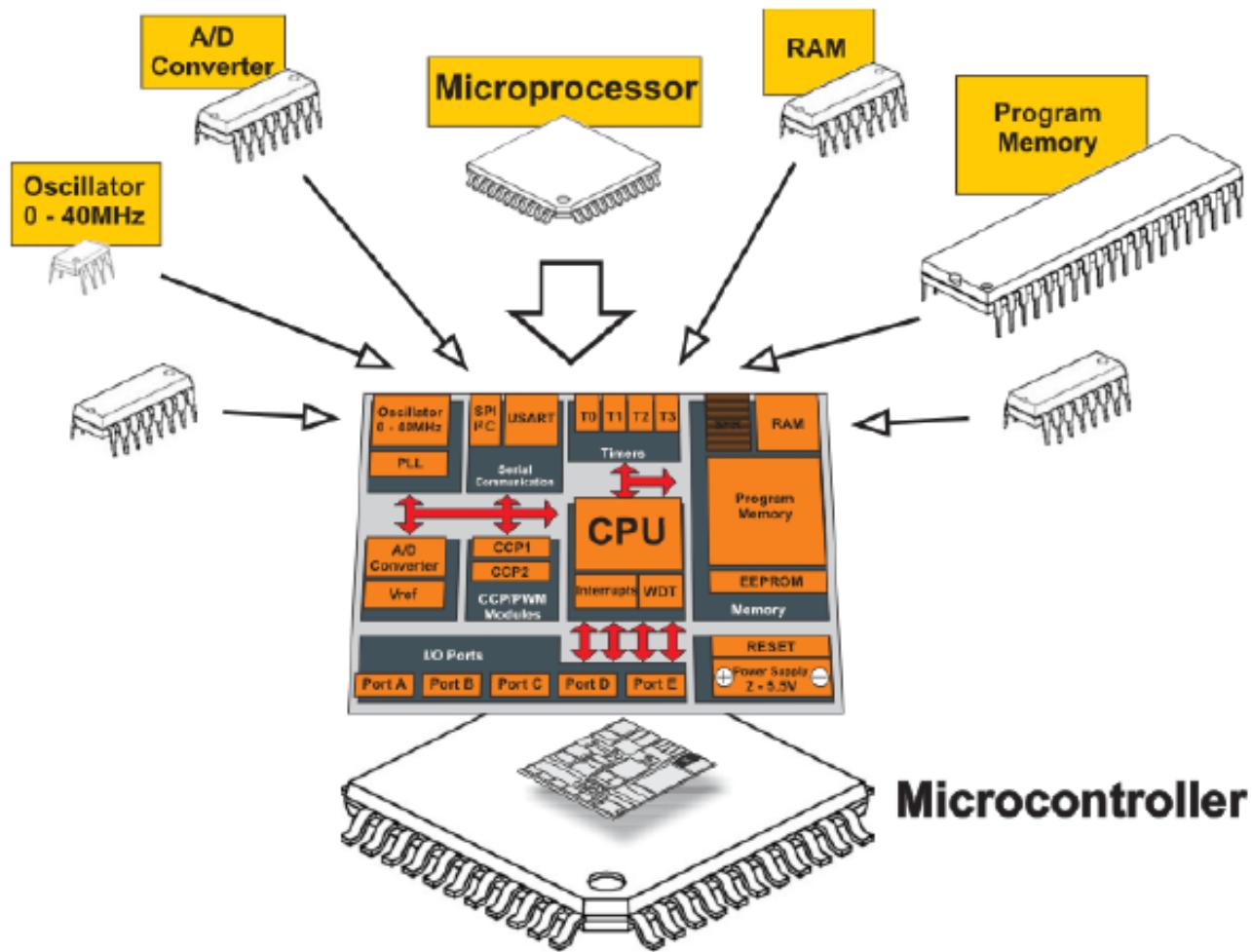
Basicamente, um microcontrolador é constituído de quatro partes:

- Memória de programa;
- Memória de dados;
- Unidade lógica Aritmética (ULA);
- Portas de I/O (Entrada e Saída).

# MICROCONTROLADORES



- Estrutura básica



# MICROCONTROLADORES



- Aplicações



# Advanced RISC Machines - **ARM**





## • Introdução

- Arquitetura ARM (primeiramente Acorn RISC Machine, atualmente Advanced RISC Machine) é uma arquitetura de processador de 32/64 bits e é usada principalmente em sistemas embarcados.
- Headquarters em Cambrige, UK
- ARM não fabrica nenhum tipo de semi-condutor
- A família córtex foi introduzida na versão 7 da arquitetura, e foi lançado recentemente a versão 8-A que traz uma arquitetura de 64 bits.

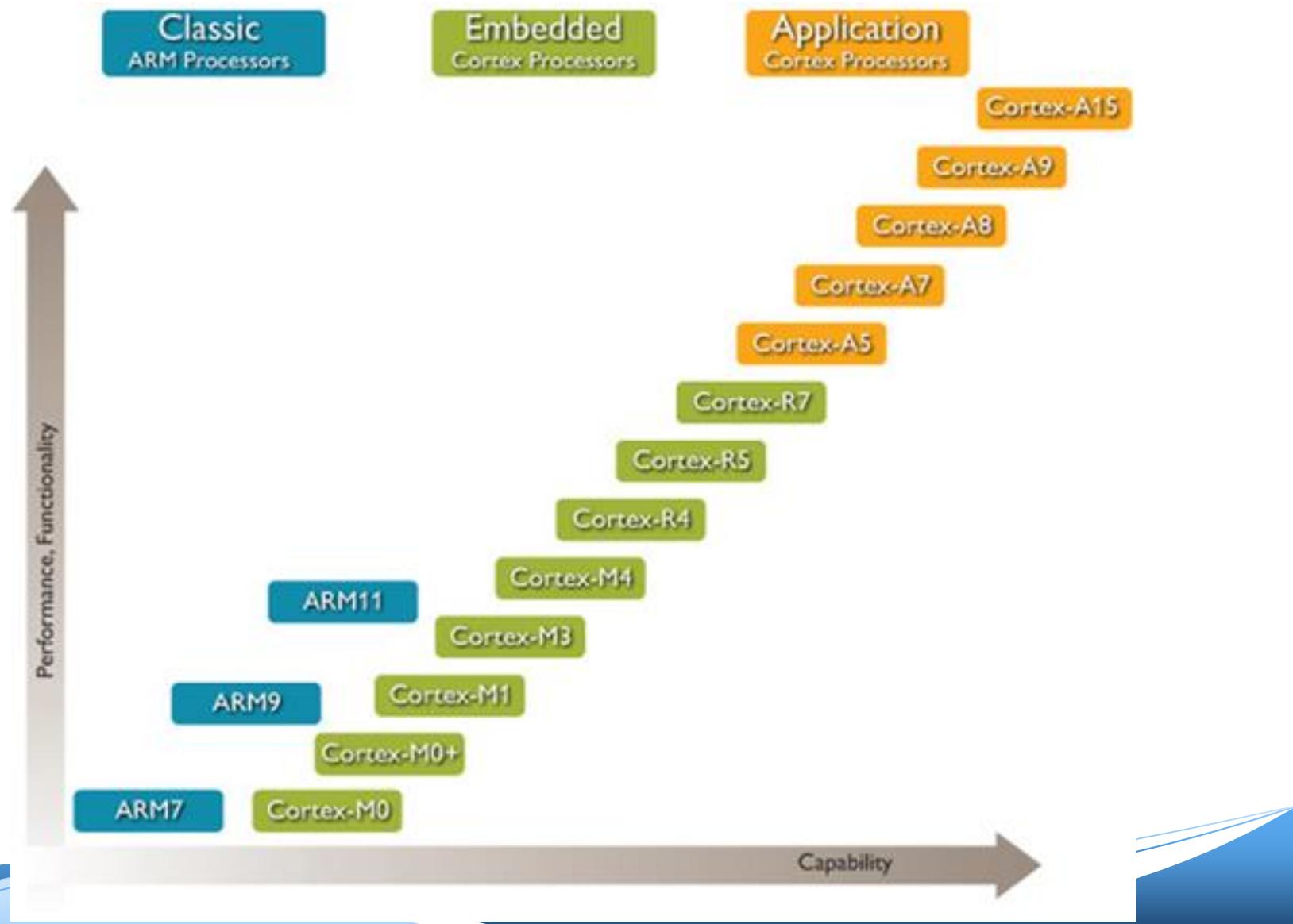


## • Introdução

- Arquitetura ARM (primeiramente Acorn RISC Machine, atualmente Advanced RISC Machine) é uma arquitetura de processador de 32/64 bits e é usada principalmente em sistemas embarcados.
- Headquarters em Cambrige, UK
- ARM não fabrica nenhum tipo de semi-condutor
- A família córtex foi introduzida na versão 7 da arquitetura, e foi lançado recentemente a versão 8-A que traz uma arquitetura de 64 bits.



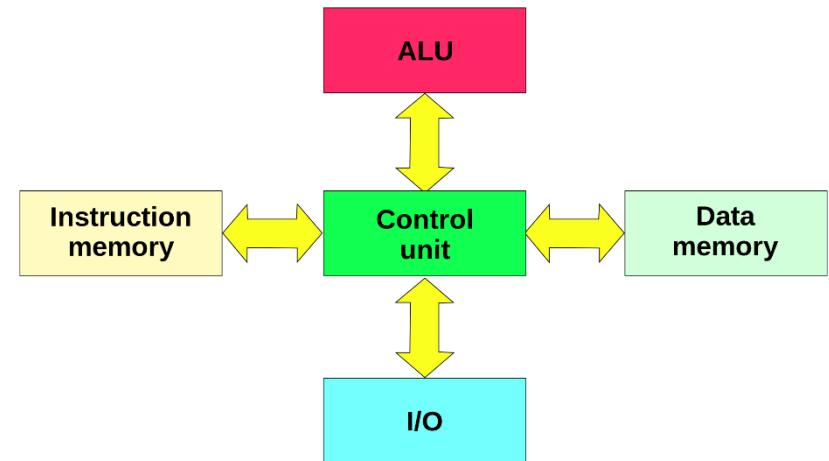
- Funcionalidade x Desempenho





## • Arquitetura

- A ARM adotou a arquitetura Harvard para a sua linha de microcontroladores.
- RISC – set de instruções reduzido. Possui dois tipos de memória, uma para dados e outra para instruções.
- Barramento de dados separados para memória de dados e memória de programa.





- Estrutura Básica

**MEMÓRIA FLASH-** É a memória em que será guardado o programa propriamente dito, no formato de “linguagem de máquina”. Foi desenvolvida na década de 80 pela Toshiba, e é do tipo EEPROM, porém se assemelha com a RAM, permitindo que múltiplos endereços sejam apagados ou escritos em uma só operação.

**MEMÓRIA SRAM-** Static RAM significa que não precisa ser periodicamente atualizada como as RAMs comuns, que sem atualização perdem seus dados. É a memória volátil, rapidamente acessada, e que é apagada toda vez que a alimentação se ausenta. Local onde são armazenadas as variáveis declaradas no programa.

**MEMÓRIA EEPROM-** É a memória não volátil, que mantém os dados escritos independente da alimentação. É acessada quando se deseja armazenar dados por períodos indeterminados, como a senha de um usuário.



- Estrutura Básica

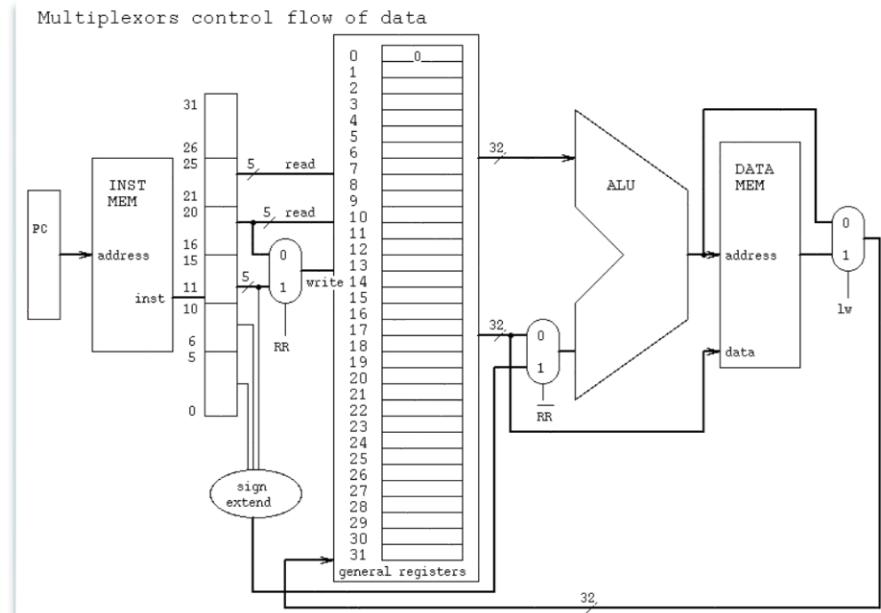
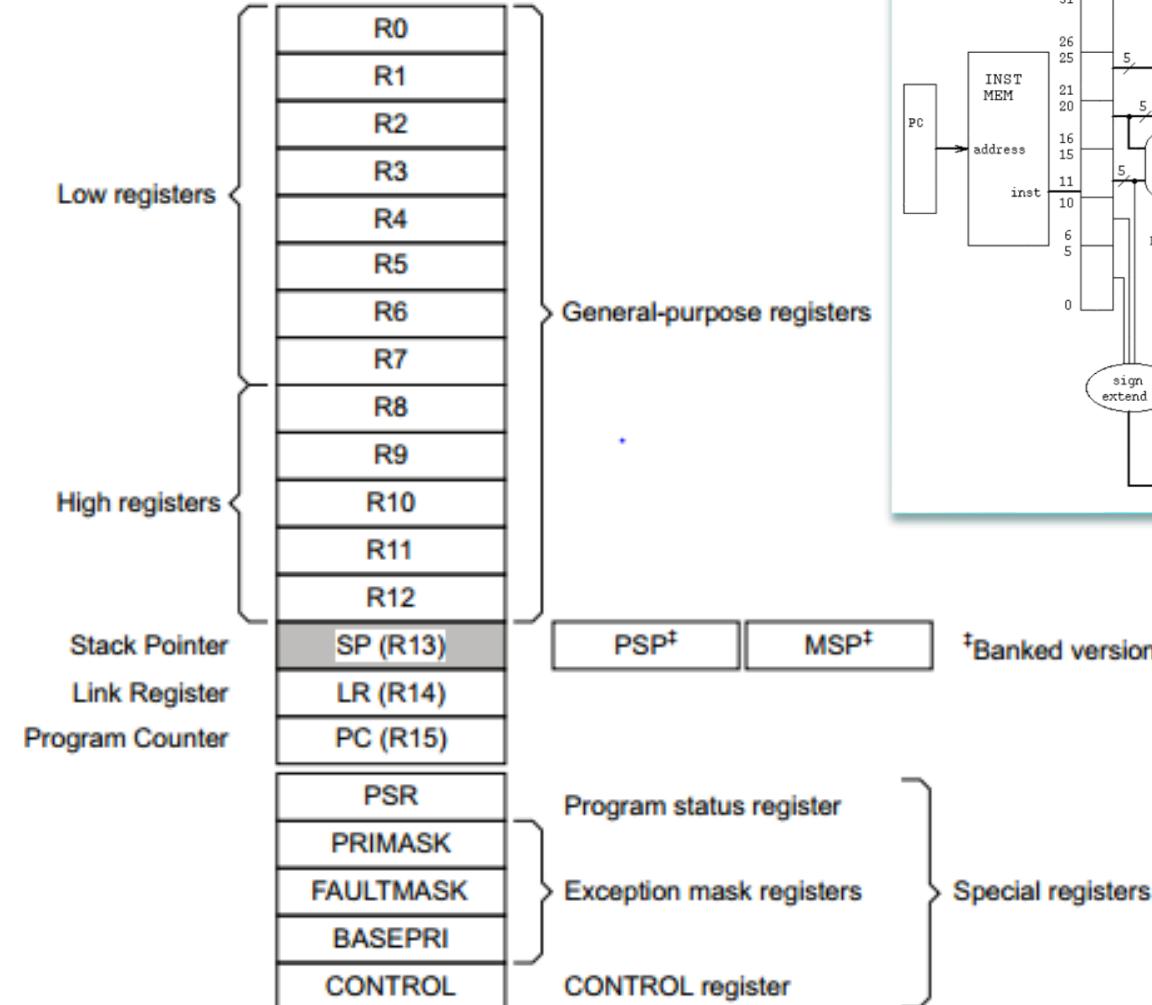
**CPU (Central Processing Unit)** - É um conjunto composto por:

- **PC (Program Counter)**: Seleciona na FLASH o próximo comando a ser executado.
- **WREGs (Work Registers)**: Registradores especiais de trabalho. São acessados constantemente e servem para realizar operações de movimentação e tem acesso direto a ULA.
- **ULA**: Unidade Lógica e Aritmética do uC (microcontrolador), responsável pelas operações lógicas e aritméticas dos WREGs.



# • Registradores

The processor core registers are:





- Estrutura Básica

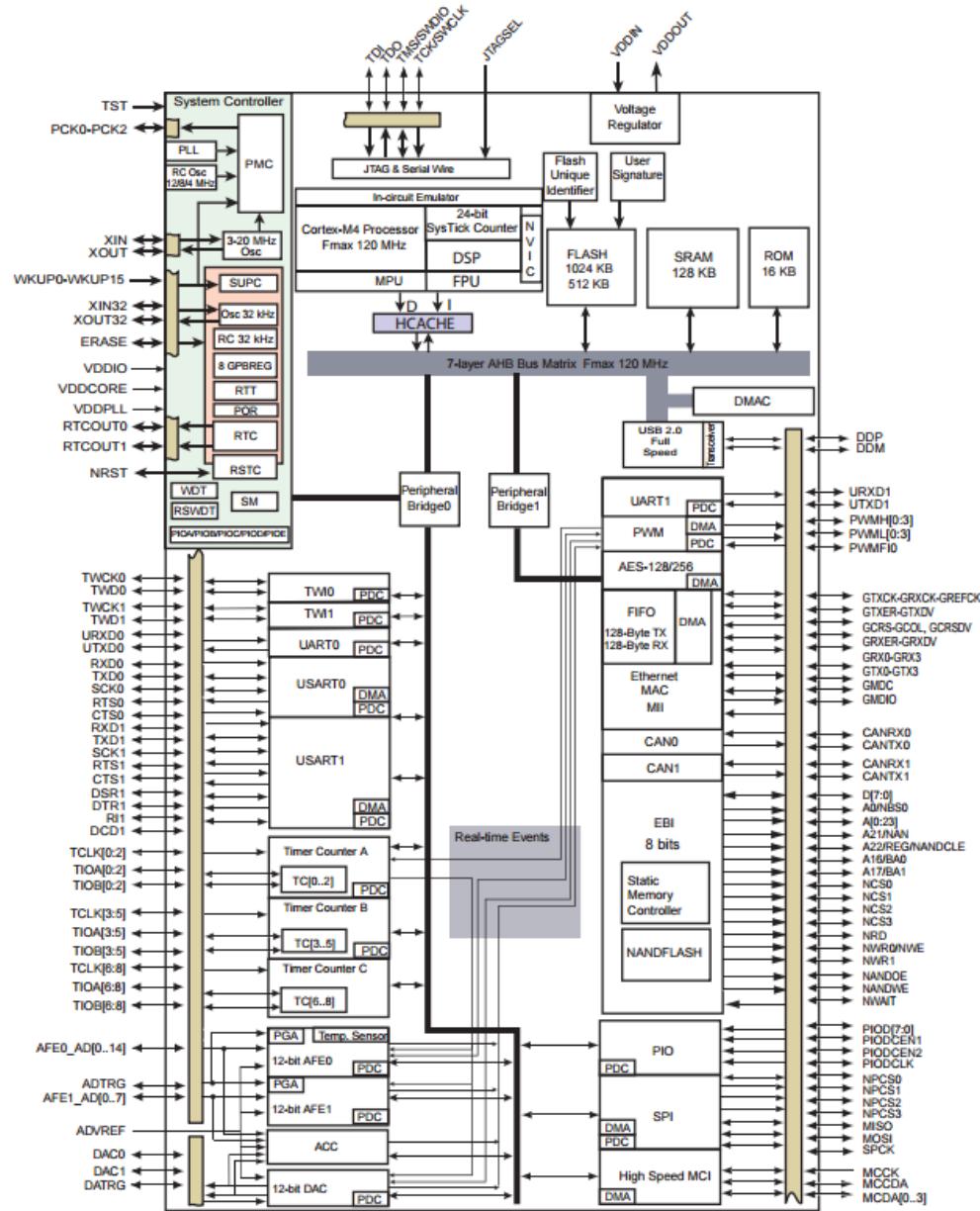
**PERIFÉRICOS:** Circuitos auxiliares que realizam trabalhos específicos. Funcionam de forma paralela ao programa do usuário (na FLASH), ocupando o código apenas para ser configurado ou entregar o resultado de algum serviço. Exemplos de periféricos são:

- **GPIO (General Purpose Input/Output):** Usados para entradas digitais, como um botão ou uma chave, e saídas digitais, como um LED ou um motor DC. O programa usuário pode setar uma saída, colocando um pino externo do PIC em nível lógico ‘0’ (0V) ou ‘1’ (5V).
- **ADC (Analog to Digital Converter):** Usado para recolher sinais analógicos externos e transformá-los em digitais, para então serem processados pelo programa.
- **USART (Universal Synchronous and Asynchronous Receiver-Transmitter):** Periférico responsável por enviar e receber dados através do protocolo RS-232.
- **TIMERS:** Temporizadores de uso geral. Garantem precisão de tempo entre eventos.

# MICROCONTROLADORES – SAM4S



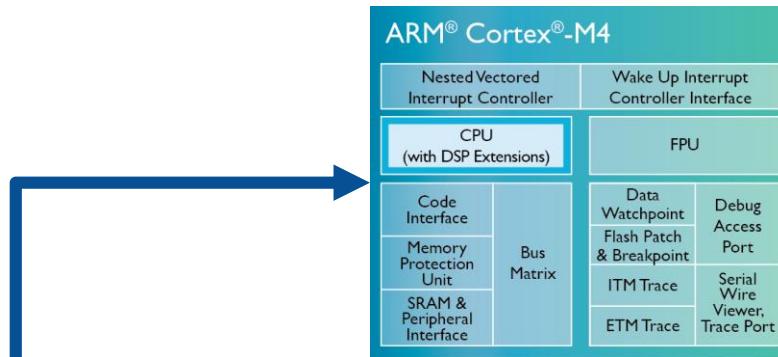
- SAM4S



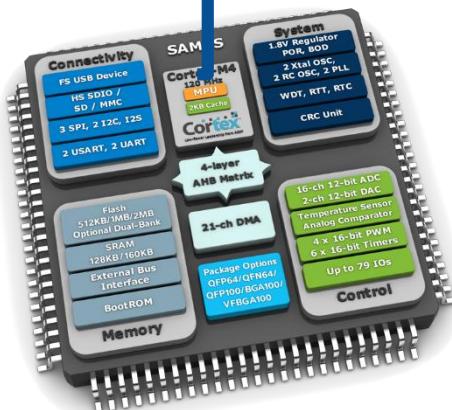
# MICROCONTROLADORES – SAM4



- SAM4E-EK



ARM\_cortex\_m4\_TechnicalReference.pdf



SAM4S-Cortex-M4.pdf



SAM4SD32 Kit.pdf

# **LINGUAGEM C**



- Resumo

A linguagem C foi desenvolvida por Brian Kernighan e Dennis M. Ritchie, na década de 70, no AT&T Bell Labs.

É uma linguagem estruturada, eficiente, rápida e tão poderosa quanto a Linguagem Assembly. A cada ano vem aumentando o número de programadores de Microcontroladores que acabam migrando da Linguagem Assembly para o C.



- Estrutura básica de um programa

Programas em C são baseados em uma ou mais funções que serão executadas, no entanto, a função Main( ) é a primeira a ser executada.

```
void main() // esta é a primeira função que será executada
{ // inicializa a função
    TrisB=0x00;
    PortB=0xFF;
} // finaliza a função
```

Observações:

- ✓ Toda função deve iniciar abrindo chave e finalizar fechando-se a chave.
- ✓ Toda instrução deve ser finalizada com ponto e vírgula (obrigatoriamente).



- ## • Representação Numérica

## Decimal:

Contador=125;

# **Binário:**

Portb=ob0111101;

## Hexadecimal:

Variavel1=0x7D;

DEC.	BINARY								HEX.
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	3
4	0	0	0	0	0	1	0	0	4
5	0	0	0	0	0	1	0	1	5
6	0	0	0	0	0	1	1	0	6
7	0	0	0	0	0	1	1	1	7
8	0	0	0	0	1	0	0	0	8
9	0	0	0	0	1	0	0	1	9
10	0	0	0	0	1	0	1	0	A
11	0	0	0	0	1	0	1	1	B
12	0	0	0	0	1	1	0	0	C
13	0	0	0	0	1	1	0	1	D
14	0	0	0	0	1	1	1	0	E
15	0	0	0	0	1	1	1	1	F
16	0	0	0	1	0	0	0	0	10
17	0	0	0	1	0	0	0	1	11

# Linguagem C



- Tipos de dados

Tipo	Tamanho em Bytes	Faixa Mínima
char	1	-127 a 127
unsigned char	1	0 a 255
signed char	1	-127 a 127
int	4	-2.147.483.648 a 2.147.483.647
unsigned int	4	0 a 4.294.967.295
signed int	4	-2.147.483.648 a 2.147.483.647
short int	2	-32.768 a 32.767
unsigned short int	2	0 a 65.535
signed short int	2	-32.768 a 32.767
long int	4	-2.147.483.648 a 2.147.483.647
signed long int	4	-2.147.483.648 a 2.147.483.647
unsigned long int	4	0 a 4.294.967.295
float	4	Seis dígitos de precisão
double	8	Dez dígitos de precisão
long double	10	Dez dígitos de precisão



- Modificadores - Static

- Indica que uma variável local deve ser armazenada na área de dados e seu conteúdo preservado durante todas as execuções da função.
- A inicialização é feita apenas uma vez

```
#include <stdio.h>
#include <string.h>

int funcao(void)
{
    static int variavel = 0;
    variavel+=2;
    return variavel;
}

main()
{
    char i=9;
    int ultimo_valor = 0;
    while (i--)
    {
        ultimo_valor = funcao();
    }
    printf("%d", ultimo_valor );
}
```

Qual o valor que irá aparecer na tela?

- a) 8
- b) 2
- c) 20
- d) 16
- e) 18
- f) Nenhuma das anteriores



- Modificadores - **Volatile**

- O compilador não aplica nenhum tipo de otimização na variável, pois **volatile** indica que a mesma pode ser alterada a qualquer momento por atividade de background (outras threads, hardware, gerenciador de interrupções etc);
- O compilador entende que a variável não tem um comportamento previsível como uma comum;
- Ela não é lida dos registradores quando a mesma se encontra lá, mesmo que isso seja extremamente mais rápido, ela será lida SEMPRE da memória.



- Modificadores - Volatile

```
unsigned char MinhaVariavel = 255;  
  
void rotina(void)  
{  
  
    while (MinhaVariavel==255)  
        ;  
  
}
```

Otimizado  
Pelo  
Compilador

```
volatile unsigned char MinhaVariavel = 255;  
  
void rotina(void)  
{  
  
    while (MinhaVariavel==255)  
        ;  
  
}
```

```
unsigned char MinhaVariavel = 255;  
  
void rotina(void)  
{  
  
    while (true)  
        ;  
  
}
```

Compilador não aplica  
otimização em trechos de  
código que possuam uma  
variável volatile



- Modificadores - Volatile

```
#include <stdio.h>

int main() {
    int a = 10, b = 100, c = 0, d = 0;

    printf("%d", a + b);
    a = b;
    c = b;
    d = b;

    printf("%d", c + d);

    return 0;
}
```

1

```
#include <stdio.h>

int main() {
    volatile int a = 10, b = 100, c = 0, d = 0;

    printf("%d", a + b);
    a = b;
    c = b;
    d = b;

    printf("%d", c + d);

    return 0;
}
```

2



- Modificadores - Volatile

1

```
.file    "noVolatileVar.c"
.intel_syntax noprefix
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "%d"
.section .text.startup,"ax",@progbits
.p2align 4,,15
.globl main
.type  main, @function
main:
.LFB11:
.cfi_startproc
sub    rsp, 8
.cfi_def_cfa_offset 16
mov    esi, 110
mov    edi, OFFSET FLAT:.LC0
xor    eax, eax
call   printf
mov    esi, 200
mov    edi, OFFSET FLAT:.LC0
xor    eax, eax
call   printf
xor    eax, eax
add    rsp, 8
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE11:
.size  main, .-main
.ident "GCC: (GNU) 4.8.2"
.section .note.GNU-stack,"",@progbits
```

2

```
.file    "VolatileVar.c"
.intel_syntax noprefix
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "%d"
.section .text.startup,"ax",@progbits
.p2align 4,,15
.globl main
.type  main, @function
main:
.LFB11:
.cfi_startproc
sub    rsp, 24
.cfi_def_cfa_offset 32
mov    edi, OFFSET FLAT:.LC0
mov    DWORD PTR [rsp], 10
mov    DWORD PTR [rsp+4], 100
mov    DWORD PTR [rsp+8], 0
mov    DWORD PTR [rsp+12], 0
mov    esi, DWORD PTR [rsp]
mov    eax, DWORD PTR [rsp+4]
add    esi, eax
xor    eax, eax
call   printf
mov    eax, DWORD PTR [rsp+4]
mov    edi, OFFSET FLAT:.LC0
mov    DWORD PTR [rsp], eax
mov    eax, DWORD PTR [rsp+4]
mov    DWORD PTR [rsp+8], eax
mov    eax, DWORD PTR [rsp+4]
mov    DWORD PTR [rsp+12], eax
mov    esi, DWORD PTR [rsp+8]
mov    eax, DWORD PTR [rsp+12]
add    esi, eax
xor    eax, eax
call   printf
xor    eax, eax
add    rsp, 24
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE11:
.size  main, .-main
.ident "GCC: (GNU) 4.8.2"
.section .note.GNU-stack,"",@progbits
```



- Operadores Matemáticos

- Aritméticos

Operador	Descrição	Exemplo
+	Soma dos argumentos	$a + b$
-	Subtração dos argumentos	$a - b$
*	Multiplicação dos argumentos	$a * b$
/	Divisão dos argumentos	$a / b$
%	Resto da divisão	$a \% b$
++	Soma 1 ao argumento ( $a=a+1$ )	$a++$
--	Subtrai 1 ao argumento ( $a=a-1$ )	$a--$

- Relacionais

Operador	Descrição
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que
=	Igual
!=	Diferente



- Operadores Lógicos

- Lógicos

Operador	Descrição
&&	Lógica E (AND)
	Lógica OU (OR)
!	Complemento (NOT)

- Operadores Bit a Bit

Operador	Descrição
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que
=	Igual
!=	Diferente

# Linguagem C



- Decisão IF e IF ELSE

```
if (expressão)
{
    comando1;
    comando2;
    comandoN;
}
```



```
If (conta == 5)
{
    a=a++;
    portc=0xFF;
}
```

```
if (expressão)
{
    comando1;
    comando2;
    comando3;
}
else
{
    comando4;
    comando5;
}
```



```
if (a>22)
{
    Valor1=x;
    y=contador+10;
}
else
{
    Valor2=x;
    Y=contador-5;
}
```



- Decisão SWITCH - CASE

```
switch (variável)
{
    case valor1:
        comando1;
        comando2;
        break;
    case valor2:
        comando3;
        comando4;
        comando5;
        break;
    default: //opcional
        comando6;
}
```



```
int contagem=4;
int valor=5;

switch (contagem)
{
    case 2:
        valor++;
        break;
    case 5:
        valor--;
        break;
    case 10:
        valor=0;
        break;
    default:
        valor=5;
}
```



- Loop FOR

For (inicialização; condição (término); incremento)

{

    comando1;

    comando2;

    comandoN;

}



void main ()

{

    int contador;

    int a = 0;

    for (contador=0; contador<=10; contador++)

        a=a+contador;

}



- Loop WHILE

O loop é repetido enquanto a expressão for verdadeira.

```
void main( )
{
    int a=15;
    while (a>10)
    {
        a--;
        delay_ms(100);
    }
}

while (expressão)
{
    comando1;
    comando2;
}
```

# Perguntas?