

# Sistema operacional de tempo real

## PIC

### Conceitos

O RTOS (Real Time Operating System) permite realizar tarefas diferentes de forma sincronizada. Em um processo de um microcontrolador simples, um processo é executado por vez. Os outros processos precisam aguardar o processo que está sendo executado no momento:

```
void main(void) {  
  
    while(TRUE){  
        sensorCorrente(); // 2ms  
        temopar(); // 10ms  
        umidade(); // 15ms  
        sensorPresenca(); // 100ms  
        sendToUart(); // 1ms  
        updateLcd(); // 10ms  
    }  
}
```

Veja como demora para executar cada método. Dentro dos métodos, pode ser que o microcontrolador necessite esperar a chegada de algum sinal externo, ou seja, mais tempo perdido que poderia estar sendo utilizado para executar outras tarefas.

Uma forma de resolver este problema é criando uma máquina de estado que executa funções de forma parcial. Veja que, na rotina abaixo, `Task_2()` demora 60us para ser chamada em um processo normal:

```
while(1){  
    Task_1(); // 60us  
    Task_2(); // 2us  
}  
// Tempo de loop = 62us
```

Se nos dividirmos `Task_1()` em intervalos e executá-los um de cada vez em uma máquina de estado, a função `Task_2()` será chamada mais vezes, isso em detrimento de um tempo de execução um pouco maior de `Task_1()`.

```
while(TRUE){  
    switch(Task_1_state){  
        case a:  
            Task_1_state_a(); // 20us  
            break;  
        case b:  
            Task_1_state_b(); // 20us  
            break;  
    }
```

```

        case c:
            Task_1_state_c(); // 20us
            break;
    }
    Task_2(); // 2us
}
// Tempo de loop = 22us

```

Nós subdividimos uma tarefa em várias partes. Também é perfeitamente possível criar este tipo de arquitetura com as interrupções. Mas um conceito muito importante na execução destas tarefas é em relação as suas prioridades.

A nível de hardware, é possível manipular níveis de prioridade de interrupções. Mas isso fica limitado a respostas de sinais externos ou a periféricos. A nível de software, como é possível configurar prioridade em duas funções que não causam interrupção na CPU? É aí que entra o RTOS

Com o sistema operacional, você consegue dividir a execução das rotinas em threads diferentes. A principal característica do RTOS é que as tarefas têm **tempos pré-definidos** em suas execuções. É possível saber exatamente o tempo que levará cada processo.

O código abaixo consegue fazer um pisca led sem utilizar uma função de delay em main. Ou seja, outros processos poderiam ser executados normalmente:

```

/*
 * File: Conceito Sistema operacional de tempo real
 * Este software nao esta implementando o RTOS,
 * esta apenas mostrando os efeitos *
 * Realizando contagem de tempo de tasks com
 * o timer
 *
 * Este software simula uma funcao delay, porem
 * durante a piscagem dos leds e possivel executar
 * qualquer outra funcao dentro do while
 */
#include <xc.h>
#include "../Bibliotecas/config.h"
#define _XTAL_FREQ 8000000
#define TRUE 1
#define FALSE 0
#define TMR0_VALUE 0xFE0C // Tempo de incremento do timer em 1ms
#define TASK_TIME 300 // Tempo em milissegundos de cada task

// Variavel incrementada a cada estouro do TIMER0
volatile unsigned long int ticks = 0;
// Variavel de contagem das tarefas
volatile unsigned long int t0, t1, t2;

void MCU_Init(void);
void tick_Init(void);
void tick_ISR(void);

```

```

void task_1(void);
void task_2(void);
void task_3(void);
void __interrupt(high_priority) relogioTempoReal(void){
    if(TMR0IF){
        tick_ISR();
    }
}

void main(void) {

    MCU_Init();
    tick_Init();

    t0 = t1 = t2 = ticks;

    while(TRUE){
        // Pisca leds
        task_1();
        task_2();
        task_3();

        // .... continuacao do codigo
    }
}

void MCU_Init(void){
    ANSELA = 0;
    ANSELB = 0;
    ANSELC = 0;
    ANSELD = 0;

    TRISD = 0;
}
/**
 * Sistema de tick
 */
void tick_ISR(void){
    INTCONbits.TMR0IF = 0;
    ticks++;
    TMR0H = TMR0_VALUE >> 8;
    TMR0L = TMR0_VALUE & 0xff;
}

/**
 * Inicia um processo de geracao de interrupcoes
 * O tempo de cada interrupcao e definido por TMR0_VALUE
 */
void tick_Init(void){
    T0CON = 0b00000001;
    TMR0H = TMR0_VALUE >> 8;
    TMR0L = TMR0_VALUE & 0xff;
}

```

```

    T0CONbits.TMR0ON = 1;

    INTCONbits.TMR0IE = 1;
    INTCONbits.TMR0IF = 0;
    INTCON2bits.TMR0IP = 1;

    INTCONbits.GIE = 1;
    INTCONbits.PEIE = 1;
    RCONbits.IPEN = 1;
}

/**
 * As tasks sao simuladas por acionamento dos LEDS no PORTD
 * Cada task e executada depois de ticks incrementar
 * acima de TASK_TIME
 */
void task_1(void){
    if(ticks - t0 >= TASK_TIME){
        t0 = ticks;
        PORTDbits.RD0 = !PORTDbits.RD0;
    }
}
void task_2(void){
    if(ticks - t1 >= TASK_TIME){
        t1 = ticks;
        PORTDbits.RD1 = !PORTDbits.RD1;
    }
}
void task_3(void){
    if(ticks - t2 >= TASK_TIME){
        t2 = ticks;
        PORTDbits.RD2 = !PORTDbits.RD2;
    }
}
}

```

## Tasks

Uma task permanece em um dos 5 estados:

- Desativado (não ter sido criada ou por ter sido deletada)
- Espera (aguarda por algum evento)
- Estado de pronta (pronta para ser executada, porém não tem ainda o controle)
- Estado de execução (A tarefa está em execução)
- Estado de Pausa (a tarefa está ativada, porém está em Pausa e não tem o controle)

Para uma task ter o controle é necessário:

- Estar no estado de pronta e receber um evento
- Possuir maior prioridade que as outras Tasks no estado de pronta

Existem 3 modos de prioridades;

- Prioridades desativadas: todas as prioridades são ignoradas (modo rápido e compacto).
- Prioridade normal: cada task terá sua prioridade. Quando existir mais de uma task no estado de pronta, a Task de maior prioridade terá o controle. Caso existam Tasks de mesma prioridade no estado de pronta, então será executado o modo Round-Robin
  - Desvantagens: Enquanto existirem tasks de maior prioridade no estado de pronta, nenhuma task de menor prioridade conseguirá ter o controle. Quando duas ou mais tasks de mesma prioridade estiverem no estado de pronta e aguardando por um mesmo evento, somente uma única task terá o controle.
- Prioridades estendidas: É garantido que todas as tasks terão o controle, de acordo com sua prioridade. Uma task que ainda não foi atendida, sua prioridade é aumentada pelo RTO's
  - Desvantagem: Consome mais memória RAM

## Conceito de multitasks

Multitasking, aplicado a sistemas operacionais de tempo real (RTOS - Real-Time Operating Systems), refere-se à capacidade do sistema de gerenciar e executar múltiplas tarefas de forma concorrente e com prioridades definidas. O conceito de multitasking em um RTOS é fundamental para lidar com sistemas que exigem resposta rápida a eventos em tempo real, como sistemas embarcados em automóveis, equipamentos médicos, sistemas de controle industrial, entre outros.

Aqui estão os principais aspectos do conceito de multitasking em um RTOS:

1. **Gerenciamento de Tarefas:** Um RTOS permite que múltiplas tarefas sejam executadas simultaneamente. Cada tarefa é uma unidade independente de execução, com seu próprio contexto e prioridade associada.
2. **Tempo Real:** Um RTOS deve garantir que as tarefas sejam executadas dentro de prazos específicos, conhecidos como restrições de tempo real. Isso significa que o sistema operacional deve ser capaz de atender aos requisitos de tempo das tarefas em tempo real, garantindo que elas sejam concluídas dentro de limites de tempo determinados.
3. **Prioridades:** As tarefas em um RTOS são geralmente atribuídas a prioridades diferentes. Isso permite que o sistema atenda primeiro às tarefas de maior prioridade, garantindo assim que os requisitos de tempo real sejam cumpridos. O RTOS geralmente executa a tarefa de maior prioridade que está pronta para ser executada.
4. **Escalonamento de Tarefas:** O RTOS emprega algoritmos de escalonamento para determinar qual tarefa deve ser executada em um determinado momento, com base em suas prioridades e estado atual. Algoritmos de escalonamento comuns incluem Round-Robin, FIFO (First-In, First-Out), Prioridade Fixa, Prioridade Dinâmica, entre outros.
5. **Context Switching:** O RTOS é responsável por alternar entre as tarefas de forma eficiente, em um processo conhecido como context switching. Durante o context switching, o estado da tarefa atual é salvo e o estado da próxima tarefa a ser

executada é restaurado. Isso permite que o RTOS mantenha o controle sobre a execução das tarefas e garanta que todas recebam tempo de CPU adequado.

6. **Sincronização e Comunicação entre Tarefas:** Em sistemas multitarefa, as tarefas muitas vezes precisam compartilhar recursos, como memória, periféricos, e dados. O RTOS fornece mecanismos para sincronização e comunicação entre tarefas, como semáforos, filas, mutexes, e sinais, garantindo que o acesso aos recursos seja feito de forma segura e coordenada.

Em resumo, o conceito de multitasking aplicado a um RTOS permite que sistemas embarcados e sistemas de tempo real executem múltiplas tarefas concorrentemente, atendendo aos requisitos de tempo real e garantindo um comportamento determinístico e previsível do sistema.

### Trabalhando com OSA

- OSA é um RTOS cooperativo e não-preemptivo (uma task de alta prioridade não interrompe uma de baixa) para dispositivos de baixa capacidade de memória e processamento
- Suporta microcontroladores PIC, AVR, dsPIC, STM8 e PIC32
- Compiladores C18, CCS, PIC, AVR studio, MikroC PRO.
- Possui um software configurador de código fonte que permite a configuração do OSA

O sistema operacional tem a capacidade de executar processos em paralelo, mesmo que o microcontrolador não suporte isso.

As tarefas ainda precisam de sincronismo. Imagine que você separou duas threads, em uma ocorrerá o incremento de uma variável, e em outra esta variável será enviada ao display LCD. Você deve sincronizar os dois processos para que a variável não seja enviada antes de ser incrementada

É muito importante evitar o uso de variáveis globais que podem ser acessadas por threads diferentes. O OSA já implementa formas de você compartilhar parâmetros entre threads diferentes sem utilizar as variáveis globais, este recurso é chamado de **mensagens**.

### Serviços do OSA RTOS

- Semáforos binários: Semáforos são recursos do sistema operacional que nos permite bloquear alguma tarefa. Eles existem como um controle de quando os métodos são chamados e também como forma de desalocar os métodos da memória. Com semáforo binário, é feita uma enumeração onde é possível trabalhar com TRUE ou FALSE para ativar ou desativar alguma tarefa
- Semáforos contadores: é um controle feito através de um contador. Quando este contador zera, não é possível acessar o método
- Alocação de memória
- Manipulação de seções críticas: em trechos de código considerados críticos, é possível desativar interrupções e outras tasks temporariamente para proteger alguns endereços de memória
- Mensagens: conversação entre tasks diferentes de forma segura
- Flags: Utilização dos bits individuais de uma variável como flags de sinalização
- Timers: o sistema operacional permite a criação de timers

Primeiramente você deve utilizar o software OSAcfg\_Tool para criar o arquivo cabeçalho.

O código abaixo faz a inicialização simples do sistema operacional. Ele realiza um pisca led.

```
#include "../../osa.h"

#define T2CON_CONST 0B01001101 //Timer2 ON, Prescaler 1:4 e
                                //Postscaler 1:10

#define PR2_CONST (49)          //1us para FOSC

//Protótipos
void Init_MCU(void);
void TickTimerIE(void);

void Task_LED0(void);
void Task_LED1(void);
void Task_LED2(void);

//Informa para o Linker do compilador mikroC que as funções
//(tasks) serão chamadas indiretamente pelo S0.
#pragma funcall main Task_LED0
#pragma funcall main Task_LED1
#pragma funcall main Task_LED2

void INTERRUPT_HIGH() iv 0x0008 ics ICS_AUTO {
    if (TMR2IF_bit){
        TMR2IF_bit = 0;
        OS_Timer();
    }
}

void main() {
    OSCCON = 0b01100011;

    OS_Init(); //Inicializa RTO's
    Init_MCU();
    TickTimerIE();

    //Cria as tasks. Máx prioridade = 0. Mín prioridade = 7
    OS_Task_Create(0, Task_LED0); //Cria task LE0 (máxima prioridade)
    OS_Task_Create(0, Task_LED1); //Cria task LE1 (máxima prioridade)
    OS_Task_Create(0, Task_LED2); //Cria task LE2 (máxima prioridade)

    //É possível editar a função OS_EI() "Operacional System Enable
    //Interrupt"
    //OS_EI(); // Enable interrupts
    //.....

    OS_Run(); //Executa o scheduler
}

void Init_MCU(void){
    ANSELA = 0;
    ANSELB = 0;
    ANSELC = 0;
    ANSELD = 0;
```

```

    ANSELE = 0;

    TRISD = 0; //PORTD configurado como Saída
    PORTD = 0; //LEDs OFF
}

void TickTimerIE(void){
    //Carrega configuração do TIMER2 (OS_TickTimer)
    T2CON = T2CON_CONST;
    PR2 = PR2_CONST;
    //Configuração geral das Interrupções
    INTCON.GIEH = 1;
    INTCON.GIEL = 1;
    RCON.IPEN = 1;
    //Habilita a interrupção do TIMER2 (OS_TickTimer)
    TMR2IE_bit = 1;
    TMR2IP_bit = 1;
    T2CON.TMR2ON = 1;
}

void Task_LED0(void){
    for(;;) //loop infinito
    {
        LATD.RD0 = ~LATD.RD0;
        OS_Delay(300);
    }
}

void Task_LED1(void){
    for(;;) {
        LATD.RD1 = ~LATD.RD1;
        OS_Delay(300);
    }
}

void Task_LED2(void){
    for(;;) {
        LATD.RD2 = ~LATD.RD2;
        OS_Delay(300);
    }
}

```

## Semáforo binário

Um semáforo binário tem os estados bloqueado ou desbloqueado. É usado para partilhar recursos e para comunicação entre tarefas. Enquanto uma tarefa utiliza um dado recurso pode bloquear um semáforo, outra tarefa antes de utilizar este recurso verifica o estado do semáforo, e se estiver bloqueado, não utiliza o recurso. Pode esperar que, o recurso fique liberto, ou pode executar outras funções.

Um semáforo binário é uma condição criada que poderá ter os estados de verdadeiro ou falso

- OS\_Bsem\_Check → Checa se está livre ou ocupado
- OS\_Bsem\_Check\_I → Checa se está livre ou ocupado em uma interrupção



- OS\_Bsem\_Reset → Bloqueia o semáforo
- OS\_Bsem\_Reset\_I → Bloqueia em interrupção
- OS\_Bsem\_Set → Libera o semáforo
- OS\_Bsem\_Set\_I → Libera em interrupção
- OS\_Bsem\_Signal → Prepara o semáforo
- OS\_Bsem\_Signal\_I → Interrupção
- OS\_Bsem\_Switch → Chaveia o estado (set para reset, ou reset para set)
- OS\_Bsem\_Switch\_I
- OS\_Bsem\_Wait → Aguarda a sinalização do semáforo
- OS\_Bsem\_Wait\_TO → Aguardo com timeout, para aguardar até um tempo

Os semáforos são acessados pelos seus índices

Exemplos de utilização:

```
void Task(void){
    for(;;){
        OS_Bsem_Set(3);
    }
}

void Task(void){
    for(;;){
        OS_Bsem_Reset(3);
    }
}

void Task(void){
    for(;;){
        if(OS_Bsem_Check(3)){
            /* ... */
        }
    }
}

void Task(void){
    for(;;){
        OS_Bsem_Wait(3);
    }
}

void Task(void){
    for(;;){
        OS_Bsem_Wait_TO(5, 20); // 20 ticks (timer)
        if(!OS_IsTimeout()){
            /* ... */
        }
    }
}
```

Para criar o semáforo, você vai gerar as variáveis macro no OSACfg utilizando a seção Binary semaphores. Ela vai criar os enums com os nomes que você definiu. Se os semáforos forem utilizados em vetores de interrupção, você deverá barrar o campo ISR.

**Binary semaphores**

3 BANK: BANK\_OS ☐ ISR

N	ID name	Comment
0	BS_LED0	
1	BS_LED1	
2	BS_LED2	
3		
4		
5		
6		
7		

OSAcfg\_Tool (v1.9) - "C:\Users\user\Desktop\OSA\_RTOS\_PIC\osa\apps\OSAcfg.h"

Path: LEDS (C:\Users\user\Desktop\OSA\_RTOS\_PIC\osa\apps\OSAcfg.h) Browse...

**Name:** LEDS

**RAM statistic**

Platform: PIC18: mikroC PRO

RAM 33

0

0

0

**System**

**Priority level:** Disable

☐ Enable all

☐ All used in interrupts

☐ Critical sections

BANK: ADDR:

OSA variables bank: default (near)

**Tasks:** 3 BANK\_OS

**Data services**

**Csem** ☐ Enable default (c) ☐ ISR

**Msg** ☐ ISR

**Smsg** ☐ ISR

**Queue** ☒ Enable ☒ ISR

**Squeue** ☐ Enable ☐ Queues identical

**Flas** ☐ ISR

**Timers**

☒ Use in-line OS\_Timer() long (System timer type)

**Task timers** ☒ Enable long ☒ Optimize by size

**Dynamic** ☐ Enable long

**Queue of timers** ☐ Enable (system)

**Static timers** 0 long ☒ Optimize by size BANK: BANK\_OS

☒ Allow assignment timers IDs in run-time

N	ID name	Comment
0	stimer1	
1	stimer2	
2		
3		
4		
5		
6		

**Binary semaphores**

3 BANK: BANK\_OS ☐ ISR

N	ID name	Comment
0	BS_LED0	
1	BS_LED1	
2	BS_LED2	
3		
4		
5		
6		
7		

Save as default Restore default ☐ Save old style names (<90200) Save Save as... Exit

Você deve setar os semáforos criados em main:

```
void main(void){
    OS_Bsem_Set(BS_LED0);
}
```

E poderá utiliza-los em suas taks:

```
void Task_LED0(void){
    for(;;){
        OS_Bsem_Wait(BS_LED0);
        LATD.RD0 = 1;
        OS_Delay(300);
    }
}
```

## Trabalhando com botões

Existem mais duas funções do sistema operacional que podem ser utilizada para verificar alguma condição:

- `OS_Cond_Wait(condição)` → Aguarda até uma condição ter sucesso

```
void Task(void) {
    for(;;){
        OS_Cond_Wait(a < 5 && TMR1IF);
    }
}
```

- `OS_Cond_Wait_TO(condição)` → Aguarda uma condição com a possibilidade de timeout

```
void Task(void) {
    for(;;){
        OS_Cond_Wait_TO(RB1 == 1, 50); // Wait 50 ticks for RB1 == 1
        if(!OS_IsTimeout()){
            /* ... */
        }
    }
}
```

## Mensagens

Mensagens podem ser por parâmetro (valor) ou referência (passagem para um ponteiro)

- `Os_Msg_Accept` → Aceita uma mensagem criada por `Send`. No momento que a mensagem é aceita, ela deixa de existir
- `Os_Msg_Accept_I`
- `bool Os_Msg_Check(msg_cb)`
- `Os_Msg_Check_I`
- `Os_Msg_Create(msg_cb)` → Cria uma mensagem
- `Os_Msg_Send(msg_cb, message)`
- `Os_Msg_Send_I`
- `Os_Msg_Send_Now`
- `Os_Msg_Send_TO` → Aguardar um tempo para fazer o envio
- `OS_Msg_Wait` → Aguarda receber uma mensagem
- `OS_Msg_Wait_TO` → Aguarda receber uma mensagem por um período de tempo

Exemplos de utilização:

```
OST_MSG_CB msg_cb; // Variável de tipo especial

void Task(void){
    OS_Msg_Create(msg_cb);
    for(;;){
    }
}
```

O send precisa da mensagem para ser enviada, ela fica disponível em Accept para ser utilizada em outro lugar.

```
OST_MSG_CB ms_cb; // Variável de tipo especial
```

```
void Task_01(void){
    OST_MSG msg;
    static char buf[10];
    for(;;){
        OS_Msg_Send(msg_cb, buf);
    }
}

void Task_02(void){
    OST_MSG msg;
    for(;;){
        OS_Msg_Send(msg_cb, msg);
        if(OS_Msg_Check(msg_cb)){
            OS_Msg_Accept(msg_cb, msg);
            /* ... */
        }
    }
}
```

Método Wait:

```
OST_MSG_CB msg_cb;

void Task(void){
    OST_MSG msg;
    for(;;){
        /* ... */
        OS_Msg_Wait(msg_cb, msg); // Espera pela mensagem e
                                   // armazena em msg
        /* ... */
    }
}
```

Método timeout:

```
OST_MSG_CB msg_cb;

void Task(void){
    OST_MSG msg;
    static char buf[10];
    for(;;){
        /* ... */
        OS_Msg_Send_TO(msg_cb, buf, 100); // Envia a mensagem
        if(OS_IsTimeout()){
            // Mensagem nao foi enviada
            /* ... */
        }
    }
}
```

```

    }
    /* ... */
}
}

```

A variável especial do tipo OST\_MSG\_CB é definida dentro do arquivo de configurações OSACfg. Para criá-la, você irá utilizar esta seção do software no campo Msg. Aqui você colocará o tipo da variável, como vamos trabalhar com um array de char, você pode colocar este campo como **char \***.

The screenshot shows the 'Data services' configuration window. The 'Msg' field is set to 'char \*'. Other options include 'Csem' (Enable checkbox, default (c) dropdown, ISR checkbox), 'Smsg' (ISR checkbox), 'Queue' (Enable checked checkbox, ISR checked checkbox), 'Squeue' (Enable checkbox, Queues identical checkbox, ISR checked checkbox), and 'Flags' (ISR checkbox).

Repare que para que a mensagem funcione, você precisará declarar um tipo especial de variável global: OST\_MSG\_CB.

## Enviando valores inteiros por mensagem

Ao trabalhar com tipos de dados mais simples (como char e int) o mais recomendado é utilizar o serviço de mensagens simples, ou Smsg. O funcionamento é o mesmo se comparado ao das mensagens. As definição da variável global passa a ser OST\_SMSG.

As funções são:

- OS\_Smsg\_Send()
- OS\_Smsg\_Send\_TO()
- OS\_Smsg\_Send\_Now()
- OS\_Smsg\_Check()
- OS\_Smsg\_Accept()
- OS\_Smsg\_Wait()
- OS\_Smsg\_Wait\_TO()

## Criar, deletar, pausar e continuar uma Task

É possível criar uma task dentro de outra task. Você pode criar esta task em algum momento do seu programa e depois deleta-la ou pausa-la. Funções do sistema operacional para realizar estas tarefas:

- OS\_Task\_Continue()
- OS\_Task\_Create()
- OS\_Task\_Define()
- OS\_Task\_Delete()
- OS\_Task\_GetCreated()
- OS\_Task\_GetCur()
- OS\_Task\_GetPriority()

- OS\_Task\_IsEnable()
- OS\_Task\_IsPaused()
- OS\_Task\_Pause()
- OS\_Task\_Replace()
- OS\_Task\_SetPriority()
- OS\_Task\_Stop()