

21. RTOS

Este capítulo trata de uma importante ferramenta de programação para desenvolvimento de projetos de mais alto nível, os sistemas operacionais de tempo real - RTOS, compatíveis principalmente com arquiteturas de 32 bits. Entretanto, com o desenvolvimento de novos *firmwares* e com a maior disponibilidade de memória, os microcontroladores de 8 bits podem suportar sistemas operacionais mínimos e ainda apresentar desempenho satisfatório. A utilização de um RTOS se justifica pelas facilidade de programação, permitindo a solução de problemas complexos, difíceis de tratar com a programação convencional. Assim, com algum sacrifício de desempenho, pode-se utilizar um RTOS em um microcontrolador de 8 bits.

21.1 INTRODUÇÃO

Um Sistema Operacional (SO) pode ser definido como uma coleção de programas que atua como uma interface entre os programas do usuário e o hardware. Sua principal função é proporcionar um ambiente para que o usuário de um sistema microprocessado execute programas no hardware do sistema de uma maneira conveniente e eficiente. Em resumo, as tarefas próprias de um sistema operacional são:

- Gerenciar o processador, a memória e os dispositivos de I/O.
- Oferecer uma interface simples para as aplicações e para o usuário.

Com base no número de usuários permitidos e no modo de execução, um sistema operacional pode ser classificado como:

- Monousuário, monotarefa: permite que um usuário possa realizar uma atividade de cada vez. O antigo MS-DOS é um exemplo de um

sistema operacional monousuário, monotarefa.

- Monousuário, multitarefa: o mais utilizado em computadores de uso pessoal. O Windows 7 e o MacOS são exemplos de um SO que permite que um único usuário interaja simultaneamente com vários programas.
- Multiusuário: permite que diversos usuários possam tirar simultaneamente vantagem dos recursos do computador. O sistema operacional deve equilibrar as exigências de todos os usuários e se certificar de que cada um dos programas utilizados contam com recursos separados e suficientes, para que um problema com um usuário não afete os outros usuários. O Unix é um exemplo de sistema operacional multiusuário.
- Sistema Operacional de Tempo Real (RTOS): utilizado em computadores embarcados em robôs, instrumentos científicos e sistemas industriais. Tipicamente, um RTOS tem pouca capacidade de interação com o usuário e nenhuma utilidade para o usuário final, já que o sistema se comportará como uma ‘caixa preta’ quando disponibilizado para uso. Sua principal tarefa é gerenciar os recursos do computador, de tal forma que uma determinada operação seja executada na mesma fração de tempo que as demais em andamento.

A principal diferença entre os sistemas operacionais multitarefas de propósito geral (GPOS) e os sistemas operacionais de tempo real é a necessidade dos RTOS apresentarem um comportamento temporal previsível, ou seja, os serviços do sistema operacional devem ser executados em frações conhecidas e esperadas de tempo.

Os serviços dos GPOS podem produzir atrasos aleatórios no software aplicativo, o que pode causar, em momentos inesperados, a resposta lenta de um aplicativo. O Comportamento temporal previsível não é um objetivo de projeto dos GPOS, eles são desenvolvidos para garantir que todos os

processos tenham a possibilidade de serem executados pela CPU. Não há preocupação com a previsibilidade temporal. O objetivo de um GPOS é garantir um tempo médio de execução.

Os RTOS, por sua vez, oferecem um tempo de execução constante independente da carga para a maioria dos serviços da CPU. Por exemplo, o tempo de transmissão de uma mensagem por uma tarefa é constante, independente de fatores, tais como: o tamanho da mensagem a ser enviada, o número de tarefas e filas de mensagens que estão sendo gerenciados pelo RTOS.

Os RTOS, em resumo, são concebidos para operar sistemas que devem prestar os seus serviços em prazos de tempo restritos e esperados e que possuem capacidade de memória e poder de processamento limitados. O núcleo (*kernel*) de um RTOS fornece uma camada de abstração cujo objetivo é esconder do usuário os detalhes do hardware do processador sob o qual o seu software será executado, conforme ilustrado na fig. 21.1. O software do usuário é dividido em blocos chamados tarefas (*tasks*). As tarefas, por sua vez, possuem restrições temporais e uma relação de comunicação e sincronização com outras tarefas.

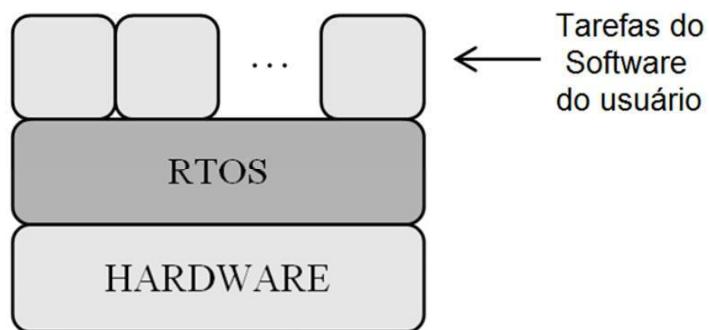


Fig. 21.1 - O núcleo do RTOS (*kernel*) fornece uma camada de abstração entre o software do usuário e o hardware.

A partir da camada de abstração fornecida, o núcleo do RTOS oferece cinco principais categorias de serviços básicos para o software do usuário, conforme ilustrado na fig. 21.2.



Fig. 21.2 – Serviços básicos oferecidos pelo núcleo de um RTOS.

A Gestão de Tarefas, representada no centro da fig. 21.2, é a principal categoria de serviço do núcleo. O conjunto de serviços oferecidos pelo RTOS permite que um aplicativo possa ser implementado, testado e mantido mais facilmente através da divisão do problema a ser resolvido em tarefas menores e de compreensão mais fácil. Essa abordagem modular permite, ainda, que as tarefas individuais sejam utilizadas em outros projetos. Nessa categoria, o principal serviço do RTOS é o escalonador de tarefas, serviço que controla a execução das tarefas da aplicação.

Outra categoria de serviços do núcleo apresentada na fig. 21.2 é a sincronização e comunicação entre tarefas. Esses serviços permitem que tarefas troquem informações sem o risco de que sejam corrompidas.

Em função dos prazos rigorosos de tempo que devem ser cumpridos pelas tarefas em muitos sistemas embarcados, os RTOS geralmente disponibilizam serviços básicos de temporização, tais como serviços de atraso e de limite de tempo.

Outros serviços que podem ser oferecidos por um RTOS são a alocação dinâmica de memória e a supervisão de dispositivos de I/O. Os serviços de supervisão de dispositivos de I/O fornecem suporte para organizar e

acessar os *drivers* do hardware de um sistema embarcado. A alocação dinâmica de memória permite que as tarefas solicitem porções da memória RAM para uso temporário no software aplicativo. Esse serviço não é oferecida pelos núcleos de pequenos RTOS concebidos para microcontroladores com limitação de memória.

Ainda, um RTOS pode expandir seus serviços oferecendo componentes opcionais, tais como: a organização de sistema de arquivos, gerenciamento de rede e interface gráfica de usuário. Buscando minimizar o consumo de memória de programa, os componentes de expansão são incluídos no sistema embarcado somente se os seus serviços forem necessários para a implementação da aplicação.

21.2 GESTÃO DE TAREFAS

Em um sistemas monotarefas, representado pelo diagrama de estados da fig. 21.3, o processador fica ocioso enquanto aguarda por um recurso de hardware mais lento como, por exemplo, uma leitura em disco. Em um sistema multitarefas, esse problema é resolvido permitindo que o processador suspenda a execução da tarefa que aguarda por dados externos e passe a executar outra tarefa. Um sistema operacional capaz de retirar uma tarefa da CPU antes do término da sua execução é chamado de preemptivo. A preempção torna os sistemas mais produtivos, permitindo que várias tarefas possam estar em andamento ao mesmo tempo, porém em estados diversos. Um sistema com essa capacidade é chamado multitarefa, podendo ser representado pelo diagrama de estados da fig. 21.4.



Fig. 21.3 – Diagrama de estados de um sistema monotarefa.

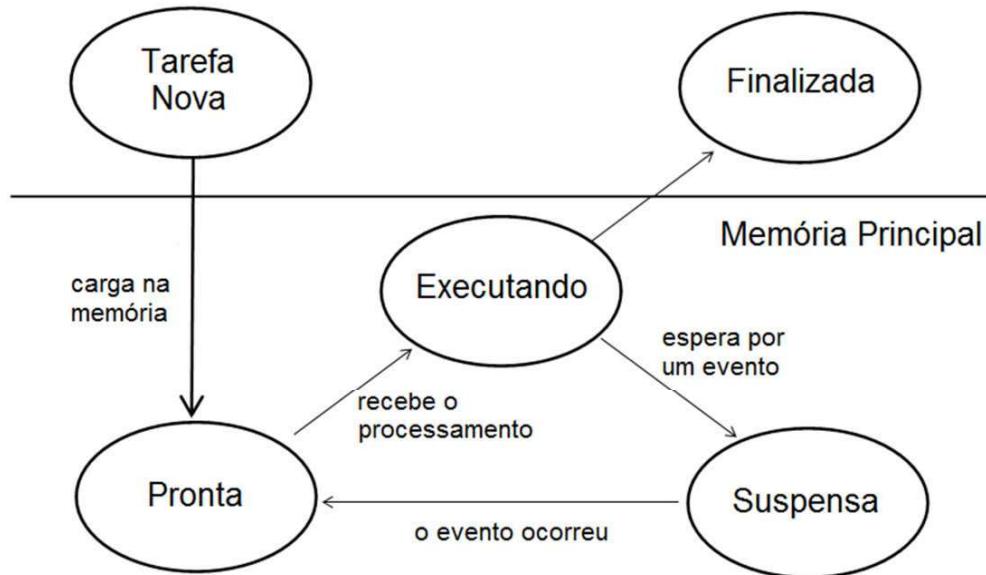


Fig. 21.4 – Diagrama de estados das tarefas em um sistema multitarefa.

Em um sistema multitarefa, as tarefas podem assumir cinco estados:

- Tarefa nova - o sistema operacional está ciente da sua existência, porém não lhe foi alocada uma prioridade e um contexto.
- Pronta - a tarefa está aguardando para ser processada em uma fila cuja ordem de execução é definida por um algoritmo de escalonamento.
- Executando - neste momento, o processador está dedicado à tarefa, executando seu código. Ela ficará em execução até que precise de algum dado externo ou aguardar por um evento ou, ainda, até que uma tarefa de maior prioridade receba o processamento no seu lugar.
- Suspensa - a tarefa não pode ser executada porque precisa de algum dado externo não disponibilizado, esperar por um evento ou que outra tarefa libere o processador.
- Finalizada - o processamento da tarefa foi encerrado e a memória destinada a sua estrutura é liberada.

Quando uma tarefa é gerada, ou pelo sistema operacional, ou por outra tarefa, inicia-se um processo que envolve carregá-la na memória, criar e atualizar certas estruturas de dados do sistema operacional necessárias para a sua execução. Até terminar esse processo, a tarefa está no estado de ‘nova’. Uma vez que o processo se encerra, ela entra no estado ‘pronta’, onde fica aguardando para ser processada. Neste momento, a sua execução passa a ser considerada pelo escalonador (*task scheduler*), que é quem decide a ordem de execução das tarefas prontas.

Uma tarefa passa do estado ‘pronta’ para o estado ‘executando’ pela ação do despachante ou executivo (*dispatcher*) quando o escalonador determina qual tarefa deve ser executada, de acordo com sua política de agendamento. Quando a tarefa encerra o seu processamento, ela entra no estado ‘finalizada’ e, em seguida, é removida da visão do sistema operacional.

Durante a execução da tarefa, podem ocorrer interrupções de software ou hardware. Nesse caso, dependendo da prioridade da interrupção, a tarefa atual pode ser transferida para o estado pronta e esperar pela sua próxima alocação pelo escalonador. Finalmente, uma tarefa pode, durante o seu curso de execução, ser interrompida devido a exigências de sincronização com outras tarefas, ou para aguardar a conclusão de alguns serviços que ela requisitou. Durante tal tempo, ela está no estado ‘suspensa’. Uma vez que o requisito de sincronização é cumprido, ou o serviço solicitado é concluído, ela é retornada para o estado ‘pronta’, passando a esperar o seu próximo agendamento.

Sempre que ocorre uma troca de tarefas, o contexto da tarefa em execução, representado pelos conteúdos do contador de programa, pilha e registradores, é salvo pelo sistema operacional em uma estrutura especial de dados, chamada de bloco de controle da tarefa (*Task Control Block – TCB*), de modo que a tarefa possa recomeçar em seu próximo agendamento. Por sua vez, o contexto da tarefa que entrará em execução tem que ser restaurado a partir do seu TCB, antes de receber o

processador. Dessa forma, sempre é perdido tempo de processamento durante a troca de contexto, no qual nenhuma tarefa está sendo executada. O diagrama temporal que representa uma troca de contexto entre duas tarefas é mostrado na fig. 21.5.

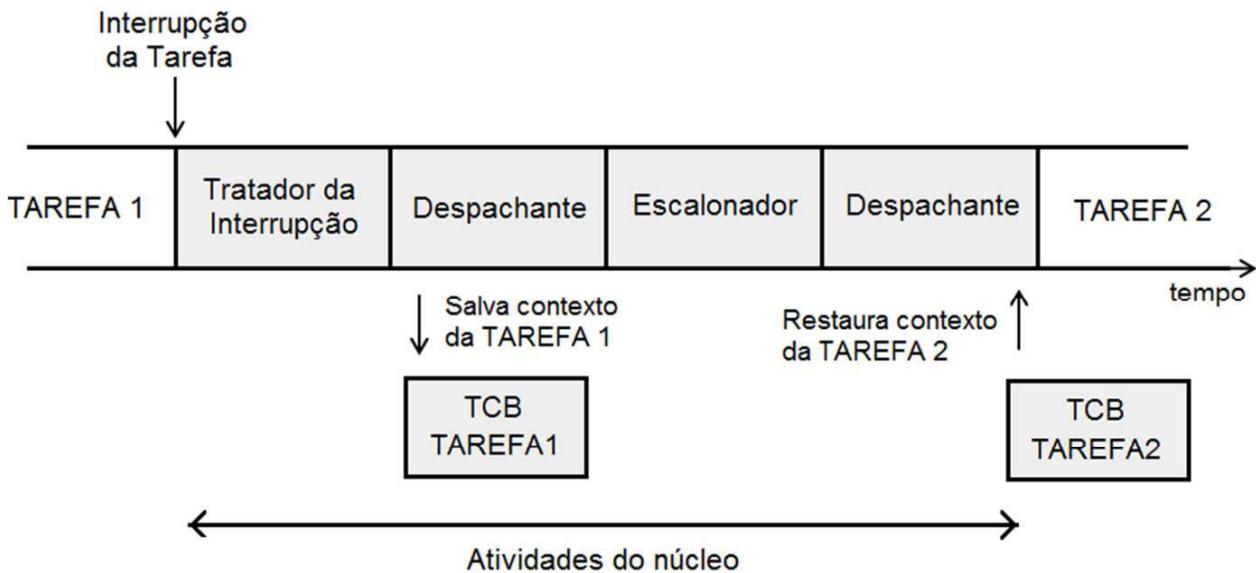


Fig. 21.5 – Diagrama temporal da troca de contexto entre duas tarefas. As atividades em cinza são realizadas pelo núcleo do sistema operacional.

Os escalonadores da maioria dos RTOSs usam um esquema conhecido como escalonamento preemptivo baseado em prioridade. No escalonamento por prioridades, uma prioridade é atribuída para cada tarefa da aplicação, geralmente na forma de um número inteiro, sendo que quanto mais rápida for a resposta desejada para a tarefa, maior será a prioridade que lhe deve ser atribuída. É a natureza preemptiva do escalonador de tarefas que garante a rapidez desejada na resposta. O termo preemptivo significa que o escalonador pode, a qualquer momento, parar a execução de uma tarefa se ele determinar que outra tarefa precisa imediatamente receber o processador. Assim, para o escalonador preemptivo baseado em prioridade, em cada instante de tempo, a tarefa pronta para executar e de maior prioridade é a tarefa que deve estar sendo executada. Ou seja, se uma tarefa de baixa prioridade e uma tarefa da alta prioridade estão prontas para serem executadas, o escalonador permitirá

que a tarefa de alta prioridade seja executada primeiro. A tarefa de baixa prioridade só conseguirá ser executada após a tarefa de alta prioridade ser finalizada.

Considerando-se três tarefas: uma de baixa prioridade, uma de média prioridade e outra de alta prioridade. Se uma tarefa de maior prioridade (alta ou média) tornar-se pronta quando uma tarefa de prioridade baixa estiver em execução, o escalonador preemptivo retirará a tarefa de baixa prioridade de funcionamento após ela completar a instrução em linguagem *assembly* em execução e entregará o processador para a tarefa de maior prioridade. Após a tarefa de mais alta prioridade finalizar o seu trabalho, a tarefa de baixa prioridade ganhará novamente o controle do processador. Obviamente, enquanto a tarefa de média prioridade está em execução, o processo de alta prioridade pode ficar pronto. Nesse caso, a tarefa de média prioridade será interrompida para permitir que a tarefa de alta prioridade seja executada. Quando a tarefa de alta prioridade terminar o seu trabalho, a tarefa de média prioridade reassume o processador. Essa situação poderia ser chamada de aninhamento preemptivo. O exemplo de uma linha do tempo representando o escalonamento preemptivo baseado em prioridades é mostrada na fig. 21.6.

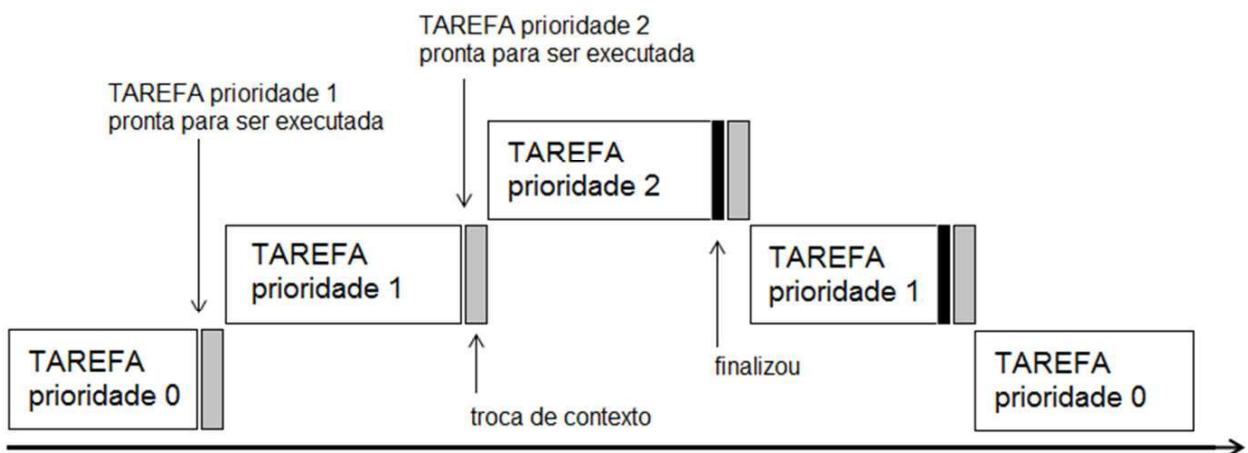


Fig. 21.6 – Exemplo de uma linha do tempo para o escalonamento preemptivo baseado em prioridades (o número 2 representa a maior prioridade).

Um RTOS menos complexo pode fazer a troca de tarefas fornecendo um tempo limite de processamento para cada tarefa, o quantum. Esgotado esse tempo, a tarefa em execução perde o processador e volta para a fila de tarefas prontas. Essa ‘preempção por limite de tempo’ é realizada pela interrupção por estouro de um temporizador do hardware. Em resumo, cada tarefa é executada por um período fixo de tempo até ser finalizada. Então, se a necessidade de uma mudança de tarefa surgisse em qualquer lugar dentro da janela do quantum, a troca de tarefa ocorreria apenas no final do tempo limite. Esse atraso seria inaceitável na maioria dos sistemas embarcados de tempo real. Um exemplo de tarefas com tempo limite para execução é apresentado na fig. 21.7.

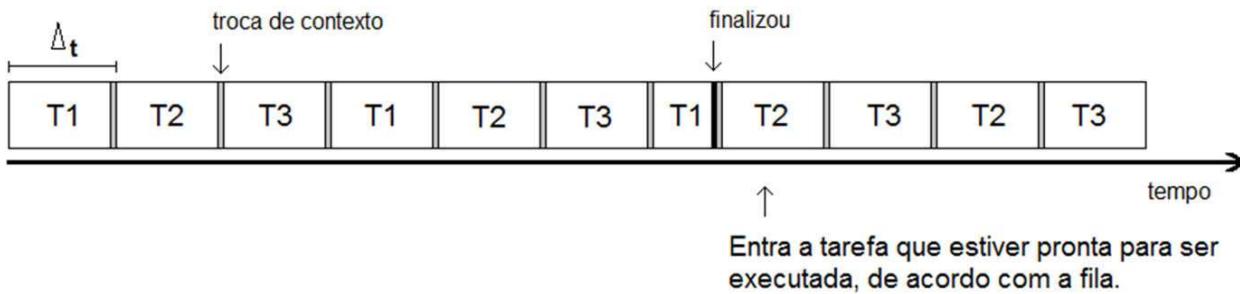


Fig. 21.7 – Diagrama de estados das tarefas em um sistema multitarefa, com tempo limite para execução das tarefas (tarefas T1, T2 e T3 todas com a mesma prioridade).

Como pode ser percebido na fig. 21.7, cada tarefa possui o mesmo tempo de execução. Dessa forma, pode-se calcular quantas trocas de tarefas são feitas por segundo. Como as tarefas são todas executadas rapidamente uma após a outra, o sistema operacional dá a impressão que todas estão sendo executadas em paralelo. Dependendo do sistema operacional, caso uma tarefa tenha sido concluída e ainda não exista uma tarefa pronta para ser executada, é possível que ele execute uma tarefa específica para essa ocasião (*Idle*).

21.3 COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE TAREFAS

Os sistemas operacionais oferecem mecanismos para a comunicação e a sincronização entre tarefas. Tais mecanismos são necessários em um ambiente preemptivo multitarefas, porque na ausência deles, as tarefas podem transmitir informações corrompidas ou interferirem umas nas outras. Por exemplo, uma tarefa pode perder o processador quando está no meio da atualização de uma tabela de dados. Se uma segunda tarefa, que adquiriu o processador, ler a partir dessa tabela, ela lerá uma combinação de algumas áreas de dados recém atualizadas e de outras que ainda não foram atualizadas. Essa atualização parcial dos dados deve torná-los inconsistentes.

Provavelmente, a forma mais popular de comunicação entre tarefas em sistemas embarcados é a transmissão de dados de uma tarefa para outra. Muitos RTOS oferecem um mecanismo de transmissão de mensagens para esse fim, conforme ilustrado na fig. 21.8. Cada mensagem pode conter uma matriz ou um registrador de dados (*buffer*). Se as mensagens podem ser enviadas mais rapidamente do que podem ser tratadas, o RTOS fornecerá filas de mensagens para manter as mensagens até que elas possam ser processadas.

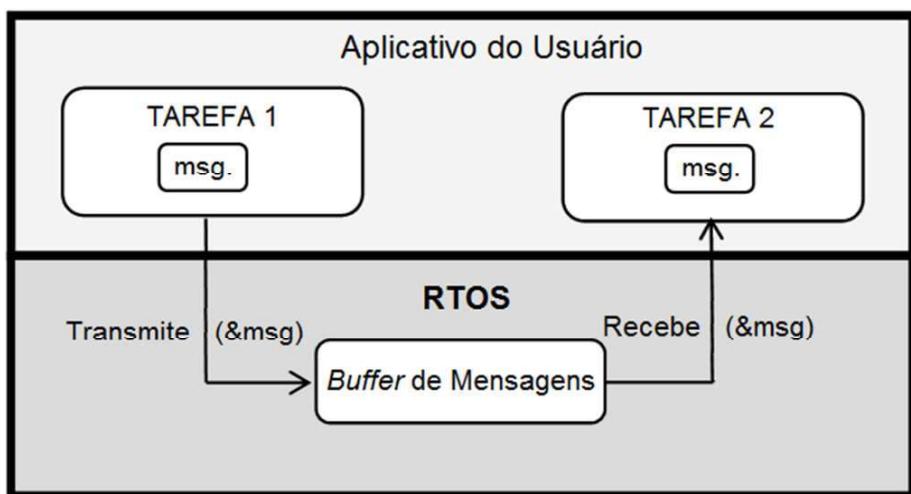


Fig. 21.8 – Esquema da comunicação entre tarefas através de mensagens.

Através do conceito de *mailbox* (caixa de correio) mensagens podem ser trocadas sem conflitos entre tarefas. As caixas podem ser usadas livremente por todas as tarefas e interrupções. Elas são identificadas com um número de *mailbox*. Geralmente, as caixas de correio permitem as seguintes operações:

- Enviar uma mensagem - cada tarefa pode enviar uma mensagem para qualquer caixa de correio. Neste caso, a mensagem a ser enviada é copiada na lista de mensagens. Se a lista de mensagens da caixa de correio já está cheia durante o envio, a tarefa é colocada no estado de espera (entrou na lista de espera de escrita). Ela permanece no estado de espera até outra tarefa buscar uma mensagem na caixa e, então, gerar espaço. Como alternativa, um tempo limite pode ser especificado para o envio, após o qual a espera é abortada. Se a lista de mensagens não está cheia quando o envio ocorre, a mensagem é imediatamente copiada na lista de mensagem e a tarefa não deve esperar.
- Ler uma mensagem - cada tarefa pode ler uma mensagem de uma caixa de correio qualquer. Se a lista de mensagens da caixa está no momento vazia (nenhuma mensagem disponível), a tarefa é colocada no estado de espera (entrou na lista de espera de leitura). Ela permanece no estado de leitura até outra tarefa enviar uma mensagem para a caixa. Como uma alternativa, um tempo limite pode ser especificado para a leitura após o qual a espera será abortada. Se a lista de mensagens não está vazia no momento da leitura, então a tarefa recebe imediatamente a mensagem.

Outra espécie de comunicação entre tarefas em sistemas embarcados é a transmissão do que pode ser chamada de informação de sincronização de uma tarefa para outra. Essa informação de sincronização pode ser considerada um comando, que por sua vez pode ser negativo ou positivo. Um comando negativo ocorre quando há concorrência entre as tarefas, ou seja, as tarefas requerem o uso de algum recurso que não pode ser

utilizado simultaneamente por mais de uma tarefa. Um exemplo seria o bloqueio, por parte de uma tarefa, do uso da USART do sistema para o seu próprio uso, impedindo a sua utilização por outra tarefa que solicitar o acesso a esse periférico. Um comando positivo ocorre quando há cooperação entre tarefas, ou seja, uma determinada tarefa precisa aguardar que outra tarefa conclua alguma operação específica para que ela possa prosseguir a sua execução.

Muitos RTOS oferecem um mecanismo de semáforo para o tratamento da sincronização negativa. Esse mecanismo permite que as tarefas bloqueiem certos recursos do sistema embarcado apenas para o seu uso, recurso que seria liberado após a sua utilização. Um semáforo contém um passe (*token*) que o código adquire para continuar a execução. Se o recurso já está em uso, a tarefa requerente é bloqueada até o passe ser retornado ao semáforo pela sua concorrente. Existem dois tipos de semáforos:

- Binários – permitem somente dois valores: zero (indisponível), um (disponível); resolvem o problema da exclusão mútua.
- Contadores – permitem uma gama maior de valores. Neste caso, o semáforo é um contêiner que mantém um número de passes. Antes de uma tarefa continuar, ela deve adquirir um passe para executar seu procedimento e, então, retornar o passe. Se todos os passes foram adquiridos por outras tarefas, a tarefa requerente esperará até outra tarefa retornar um passe para o semáforo.

Os semáforos geralmente permitem as seguintes operações:

- Espera por passe – quando uma tarefa requisita por meio de uma função do RTOS um recurso controlado por um semáforo, se um passe está disponível a tarefa continuará a sua execução. De outra forma, ela será bloqueada até o passe ser disponibilizado ou, em alguns casos, um tempo limite opcional for excedido.
- Retorna (envia) passe - após completar a sua operação sobre um recurso, a tarefa retornará o passe associado ao semáforo através de uma função do RTOS.

O termo semáforo binário é usado frequentemente como um sinônimo de mutex. Porém, em alguns RTOS, ao contrário dos semáforos, apenas uma tarefa é proprietária de um mutex num determinado momento. Ou seja, um mutex pode ser destravado somente pela tarefa que se apropriou dele.

Para a sincronização positiva, diferentes RTOS oferecem diferentes mecanismos. Para a sincronização entre as tarefas sem a troca de dados, alguns RTOS oferecem os sinais. Nesse caso, cada tarefa ativa conteria seu próprio bit de sinal com o qual as seguintes operações poderiam ser executadas:

- Espera por sinal - cada tarefa pode esperar por seu bit de sinal. Ela espera até seu bit de sinal ser ativado por outra tarefa. Após o sinal ser recebido, a tarefa em espera limpa seu bit de sinal e entra para o estado ‘pronta’ ou ‘executando’.
- Envia um sinal - cada tarefa e cada interrupção podem ativar um bit de sinal de qualquer outra tarefa.
- Limpa um sinal - uma tarefa pode limpar o bit de sinal de qualquer outra tarefa.

21.4 ALOCAÇÃO DINÂMICA DE MEMÓRIA

Muitos sistemas operacionais de computação geral em tempo não real oferecem serviços de alocação de memória a partir da memória livre (*heap*) que se encontra entre a área destinada ao armazenamento permanente e a pilha. Os serviços *malloc* e *free*, utilizados na linguagem C, trabalham a partir dessa memória livre. Chamando *malloc*, uma tarefa solicita ao sistema operacional um pedaço desta memória livre. Quando essa tarefa, ou mesmo outra tarefa, não necessitar mais da memória alocada, ela pode liberá-la chamando *free*, permitindo que o sistema operacional possa destiná-la para a utilização em outra tarefa.

A memória livre está sujeita a um processo de fragmentação que pode causar a degradação do serviço de alocação. Essa fragmentação é causada porque quando a área de memória solicitada é liberada, ela pode ser dividida em porções menores pelas próximas chamadas de *malloc*. Após muitas chamadas de *malloc* e *free*, pequenas porções de memória podem aparecer entre as áreas da memória livre que estão sendo usados pelas tarefas. Essas porções, inúteis para as tarefas, ficam presas entre as áreas de memória em uso, não podendo ser aglutinadas em uma porção maior. Ao longo do tempo, a memória livre estará repleta destas porções, o que pode gerar a recusa por parte do sistema operacional da solicitação de memória de certo tamanho, apesar do sistema operacional ter memória suficiente disponível. Em outras palavras, a memória disponível está dispersa em pequenos fragmentos distribuídos em várias partes separadas da memória livre e o sistema operacional não pode atender a solicitação de uma tarefa. A fragmentação pode ser resolvida por softwares de desfragmentação. Contudo, esses softwares injetam atrasos de duração aleatória nos serviços da memória livre.

Os sistemas operacionais de tempo real não podem conviver com atrasos aleatórios, nem permitir que a fragmentação de sua memória faça com que eles recusem o uso da memória livre por uma tarefa. Eles evitam a fragmentação da memória utilizando blocos de dimensões fixas, os *pools*. Um RTOS permite configurar vários *pools*, cada um consistindo do mesmo número de porções de memória, os *buffers*. Em um dado *pool*, todos os *buffers* são do mesmo tamanho. Os *pools* evitam a fragmentação da memória externa, pois não permitem que um *buffer* retornado ao *pool* seja dividido em *buffers* menores em futuras solicitações. Em vez disso, quando um *buffer* é retornado ao *pool*, ele é colocado em uma ‘lista de *buffers* livres’ de *buffers* do seu tamanho, que ficam disponíveis para futura reutilização no seu tamanho original. Através desse mecanismo de alocação, a memória é alocada e liberada a partir do *pool* em um tempo previsível e constante.

21.5 BRTOS

Para ilustrar o funcionamento de um RTOS será utilizado o *Brazilian Real-Time Operating System* (BRTOS), desenvolvido por brasileiros e gratuito. O BRTOS é um sistema operacional de tempo real de pequeno porte que suporta 32 tarefas e fornece controle sobre semáforos, mutex, caixas de mensagens e filas. Ele executa o escalonamento preemptivo baseado em prioridades e conta com suporte para diversos microcontroladores, entre eles o ATmega328.

Para utilizar o BRTOS em uma aplicação, é necessário configurar alguns parâmetros do sistema no seu arquivo **BRTOSConfig.h**:

- Para evitar o uso desnecessário da memória do sistema, é importante definir o número de tarefas (NUMBER_OF_TASKS) que serão utilizadas na aplicação.
- Para definir a resolução do gerenciador de tempo do RTOS (marca de tempo), é preciso definir a frequência do barramento do microcontrolador em Hz (configCPU_CLOCK_HZ), a resolução do gerenciador de tempo do RTOS propriamente dita (configTICK_RATE_HZ), cujos valores recomendados estão entre 1 ms (1000 Hz) e 10 ms (100 Hz), e o *prescaler* do periférico responsável pela base de tempo (configTIMER_PRE_SCALER e configTIMER_PRE_SCALER_VALUE). Esses valores são utilizados pela função void TickTimerSetup(void) do arquivo **hal.c**.
- Para cada serviço utilizado é preciso habilitá-lo e definir o seu número máximo de instâncias. Por exemplo, se a aplicação utilizar o serviço de semáforos, é preciso definir o valor 1 para BRTOS_SEM_EN e um valor para BRTOS_MAX_SEM.

A seguir, é apresentada a estrutura do arquivo **BRTOSconfig.h** para o microcontrolador Atmega328 utilizada nos dois próximos exemplos, com a frequência da CPU ajustada em 16 MHz e o passo de tempo em 1 ms.

```

// Define if simulation or DEBUG
#define DEBUG 1

/// Define if verbose info is available
#define VERBOSE 0

/// Define if error check is available
#define ERROR_CHECK 0

/// Define if compute cpu load is active
#define COMPUTES_CPU_LOAD 1

// Define if whatchdog active
#define WATCHDOG 1

/// Define Number of Priorities
#define NUMBER_OF_PRIORITIES 16

/// Define if OS Trace is active
#define OSTRACE 0

#if (OSTRACE == 1)
    #include "debug_stack.h"
#endif

// Define the number of Task to be Installed
// must always be equal or higher to NumberOfInstalledTasks
#define NUMBER_OF_TASKS 6

/// Define if TimerHook function is active
#define TIMER_HOOK_EN 0

/// Define if IdleHook function is active
#define IDLE_HOOK_EN 0

// Habilita o serviço de semáforo do sistema
#define BRTOS_SEM_EN 1

// Habilita o serviço de mutex do sistema
#define BRTOS_MUTEX_EN 1

// Habilita o serviço de mailbox do sistema
#define BRTOS_MBOX_EN 0

// Habilita o serviço de filas do sistema
#define BRTOS_QUEUE_EN 1

/// Enable or disable queue 16 bits controls
#define BRTOS_QUEUE_16_EN 0

/// Enable or disable queue 32 bits controls
#define BRTOS_QUEUE_32_EN 0

// Define o número máximo de semáforos (limita a alocação de memória p/ semáforos)
#define BRTOS_MAX_SEM 2

// Define o número máximo de mutex (limita a alocação de memória p/ mutex)
#define BRTOS_MAX_MUTEX 2

// Define o número máximo de Mailbox (limita a alocação de memória p/ mailbox)
#define BRTOS_MAX_MBOX 1

```

```

// Define o número máximo de filas (limita a alocação de memória p/ filas)
#define BRTOS_MAX_QUEUE      3

// TickTimer Defines
#define configCPU_CLOCK_HZ      (INT32U)16000000
#define configTICK_RATE_HZ       (INT32U)1000
#define configTIMER_PRE_SCALER   (INT8U)3
#define configTIMER_PRE_SCALER_VALUE (INT8U)64
#define OSRTCEN                  0

//Stack Defines
// P/ ATMEGA com 2KB de RAM, configurado com 512 p/ STACK Virtual
#define HEAP_SIZE 4*128

// Queue heap defines
// Configurado com 512B p/ filas
#define QUEUE_HEAP_SIZE 1*32

// Stack Size of the Idle Task
#define IDLE_STACK_SIZE          (INT16U)80

```

O BRTOS utiliza o temporizador/contador 0 do ATmega328 para gerar a base de tempo (*time tick*) do sistema operacional. Desta forma, o software de aplicação não pode empregar esse temporizador.

Para ilustrar a utilização do BRTOS, será empregado um exemplo bem simples, um contador de dois dígitos hexadecimal, com as seguintes características:

- um botão para contagem crescente (0 a 0xFF);
- um botão para contagem decrescente (0xFF a 0);
- um botão para a inicialização da contagem (0x00);
- dois *displays* de 7 segmentos multiplexados.

Cada vez que um botão de contagem é pressionado ou mantido pressionado a contagem é efetuada. Quando a contagem chega ao seu valor máximo, volta novamente ao mínimo, e vice-versa.

O BRTOS permite uma abordagem modular do problema, onde cada função do programa será desempenhada por uma tarefa. Na tab. 21.1, é apresentado um resumo comparativo entre a programação convencional e a programação com o auxílio de um RTOS para a implementação do exemplo supracitado.

Tab. 21.1 – Exemplo comparativo entre a programação convencional e a programação empregando um RTOS.

Programação Convencional	RTOS
<pre> int main() { Inicializações_normais(); while(1) { Multiplexa_Display(); Incrementa_Contagem(); Decrementa_Contagem(); Inicializa_Contagem(); } //----- //funções do programa //----- void Multiplexa_Display() {.....}; void Incrementa_Contagem(){.....}; void Decrementa_Contagem(){.....}; void Inicializa_Contagem(){.....}; </pre>	<pre> int main() { Inicializações_normais(); Inicializações_RTOS(); while(1); } //----- //tarefas do RTOS //----- void Multiplexa_Display() {while(1){...}}; void Incrementa_Contagem(){while(1){...}}; void Decrementa_Contagem(){while(1){...}}; void Inicializa_Contagem(){while(1){...}}; </pre>
Pode utilizar funções ou não. Cada função não pode conter um laço infinito.	Cada função é uma tarefa independente do RTOS, com um laço infinito de execução.
As tarefas são sequenciais, executadas no programa principal dentro de um laço infinito. O paralelismo que se consegue é com o uso das interrupções do microcontrolador.	As tarefas são concorrente. O RTOS gerencia sua execução. Existe a sensação de paralelismo na execução e, ainda, as interrupções normais do microcontroladores podem ser utilizadas.
O programa pode travar em alguma função, como por exemplo, quando existe um laço de espera que nunca é finalizado, pois a execução do programa é sequencial.	O programa não trava em funções com laços de espera. Se uma tarefa ficar presa, o RTOS vai passar o controle para outra função com o passar do tempo.
Dependendo do problema, maior complexidade de programação.	Menor complexidade de programação – melhor estruturação.
Uso de menor quantidade de memória.	Emprega mais memória, impacto maior na RAM.
Pode obter o máximo de desempenho, com a utilização constante da CPU para a execução do programa.	Perda de processamento na troca de contexto pela CPU (<i>overhead</i>).

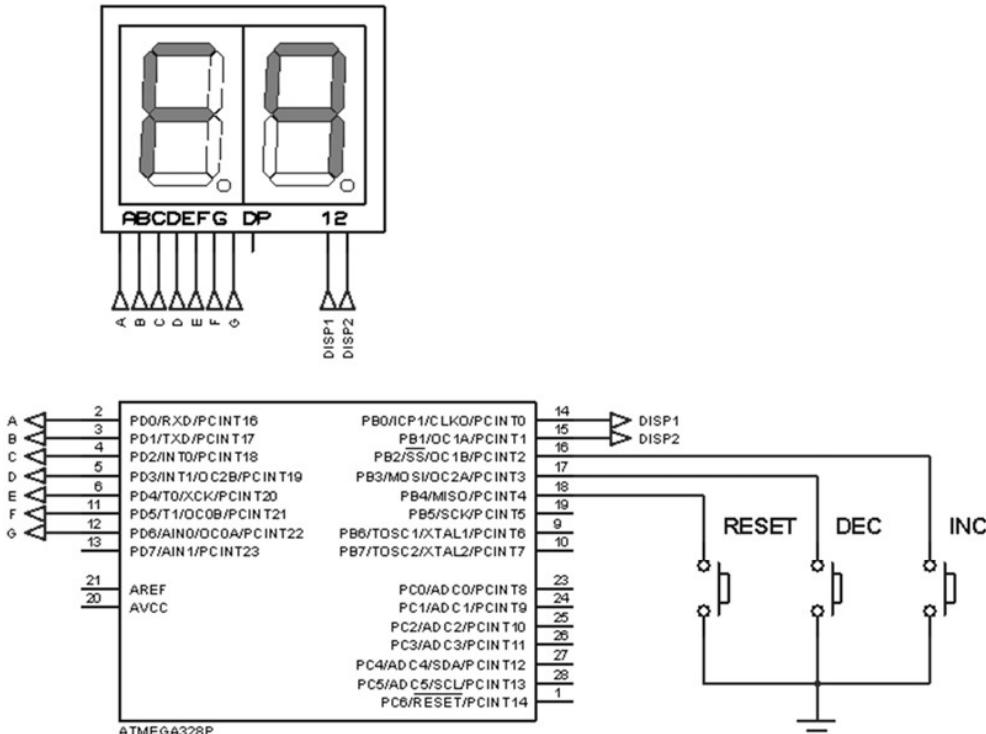


Fig. 21.9 – Diagrama esquemático do contador que embarca o BRTOS no seu firmware (circuito simplificado).

Para a execução do exemplo acima, são necessários os seguintes passos:

1. Baixar a versão mais recente do BRTOS para o ATmega328 (<http://code.google.com/p/brtos/downloads/list>) e descompactar em uma pasta de livre escolha. Neste exemplo, foi utilizado o arquivo **BRTOS 1.6x - ATMEGA328.rar**.
 2. Abrir o AVR Studio e criar o projeto com o nome Exemplo_BRTOS na pasta desejada, com a seleção do ATmega328P.
 3. Copiar na pasta do projeto `<..\\Exemplo_BRTOS\\Exemplo_BRTOS>`, onde fica o arquivo **Exemplo_BRTOS.c** (principal), somente as pastas geradas na descompactação do arquivo do BRTOS: brtos, drivers, hal e includes; não incluir os arquivos **main.c** e demais, que são do projeto exemplo do BRTOS.
 4. No arquivo **Exemplo_BRTOS.c** coloque o código apresentado abaixo (programa principal).

```
#include "BRTOS.h"
#include "tasks.h"
#define FOSC 16000000 // Frequênci da CPU

// Associa nomes as tarefas
const CHAR8 Task1Name[] PROGMEM = "Multiplexa_Display";
const CHAR8 Task2Name[] PROGMEM = "Incrementa_Contagem";
const CHAR8 Task3Name[] PROGMEM = "Decrementa Contagem";
```

```

const CHAR8 Task4Name[] PROGMEM = "Inicializa_Contagem";

PGM_P MainStringTable[] PROGMEM =
{
    Task1Name,
    Task2Name,
    Task3Name,
    Task4Name
};

//-----
void avr_Init() //inicialização dos registradores do microcontrolador utilizado
{
    DDRD = 0xFF;
    PORTD = 0xFF;
    DDRB = 0b00000011;
    PORTB = 0b11111100;
}
//-----
int main(void)
{
    avr_Init(); //inicialização dos registradores do microcontrolador utilizado
    BRTOS_Init(); //inicialização do BRTOS

    /* Instala todas as tarefas no seguinte formato: (Endereço da tarefa, Nome da
       tarefa, Número de Bytes do Stack Virtual, Prioridade da Tarefa)
       Cada tarefa deve possuir uma prioridade. A instalação de uma
       tarefa em um prioridade ocupada gerará um código de exceção*/
    if(InstallTask(&Multiplexa_Display,Task1Name,40,4) != OK)//tarefa de maior prioridade
        while(1){};// Oh Oh - Não deveria entrar aqui !!!
    if(InstallTask(&Incrementa_Contagem,Task2Name,40,2) != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!
    if(InstallTask(&Decrementa_Contagem,Task3Name,40,1) != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!
    if(InstallTask(&Inicializa_Contagem,Task4Name,40,3) != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!
    /*Inicialização do escalonador. A partir deste momento as tarefas instaladas
       começam a ser executadas*/
    if(BRTOSStart() != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!
    while(1){};//laço infinito
}
//-----

```

5. Adicionar ao projeto os arquivo **BRTOS.c** e **HAL.c**, respectivamente, das pastas: <.. \Exemplo_BRTOS\Exemplo_BRTOS\brtos> e <.. \Exemplo_BRTOS\Exemplo_BRTOS\hal> .
6. Adicionar ao projeto um novo arquivo, renomeá-lo como **tasks.c** e inserir o código a seguir (é neste arquivo que estão as tarefas).

```

#include "BRTOS.h"
#include "drivers.h"
#include "tasks.h"

#define DEBOUNCE 10
#define MUX 2

#define botao_INCREMENTO !(PINB&(1<<PINB2)) //leitura do botão de incremento
#define botao_DECREMENTO !(PINB&(1<<PINB3)) //leitura do botão de decremento
#define botao_RESET      !(PINB&(1<<PINB4)) //leitura do botão de reset

const char tabela[16] PROGMEM = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x02, 0x78, 0x00,
                                0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};

volatile unsigned char cont;
//-----
void Multiplexa_Display()
{
    while(1)
    {
        PORTB &= 0xFD;           //desliga display 2
        PORTD = pgm_read_byte(&tabela[cont/16]); //valor no display 1
        PORTB |= 0x01;          //liga display 1
        DelayTask(MUX);        //espera por intervalo: MUX ticks
        PORTB &= 0xFE;          //desliga display 1
        PORTD = pgm_read_byte(&tabela[cont%16]); //valor no display 2
        PORTB |= 0x02;          //liga display 2
        DelayTask(MUX);        //espera por intervalo: MUX ticks
    }
}
//-----
void Incrementa_Contagem()
{
    while(1)
    {
        if(botao_INCREMENTO)
        {
            cont++;
            DelayTask(200);
        }
        else
            DelayTask(1);
    }
}
//-----
void Decrementa_Contagem()
{
    while(1)
    {
        if(botao_DECREMENTO)
        {
            cont--;
            DelayTask(200); //aguarda expirar o tempo: DEBOUNCE ticks
        }
        else
            DelayTask(1);
    }
}
//-----

```

```

void Inicializa_Contagem(void)
{
    while(1)
    {
        if(botao_RESET)
            cont=0;
        else
            DelayTask(1);
    }
}
//-----

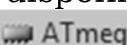
```

7. Modificar o arquivo **tasks.h** na pasta <...\\Exemplo_BRTOS\\Exemplo_BRTOS\\includes>, substituindo o código existente pelo abaixo.

```

/****************************************************************************
 *                                BRTOS
 *          Brazilian Real-Time Operating System
 *          Acronymous of Basic Real-Time Operating System
 *          Open Source RTOS under MIT License
 * OS Tasks
 ****
#include "hardware.h"
void Multiplexa_Display(void);
void Incrementa_Contagem(void);
void Decrementa_Contagem(void);
void Inicializa_Contagem(void);
void Transmite_Uptime(void);
void Transmite_Duty_Cycle(void);
void Transmite_RAM_Ocupada(void);
void Transmite_Task_Stacks(void);
void Transmite_CPU_Load(void);
void Reason_of_Reset(void);

```

8. Acessar o menu <Project> <Properties> <toolchain> <AVR/GNU C Compiler> <Directories>, caso essa opção não esteja disponível é possível clicar diretamente no ícone da janela principal  , ver a fig. 3.12 (capítulo 3). Então, devem ser adicionados os caminhos dos *includes* do BRTOS (*Include Paths*), os quais são:

-/brtos/includes
-/hal
-/drivers
-/includes

9. Escolher o nível de otimização desejado (fig. 3.12, capítulo 3).

10. Finalmente, compilar o projeto.

A seguir, apresenta-se uma aplicação que utiliza o serviço de mutex do BRTOS. Neste caso, duas tarefas disputam o uso da USART do Atmega328. Para ter acesso ao recurso, a tarefa de menor prioridade bloqueia o acesso da USART, reservando-a apenas para o seu uso.

Além de se adicionar ao projeto os arquivos **BRTOS.c** e **HAL.c**, é necessário também adicionar os arquivos **serial.c**, **mutex.c** e **queue.c** (das pastas do BRTOS que devem ter sido copiadas para o projeto). Os passos para a execução do programa foram apresentados no exemplo anterior e os arquivos novos do programa são:

1. Mutex_USART_BRTOS.c (programa principal)

```
#include "BRTOS.h"
#include "tasks.h"
#include "serial.h"

#define FOSC 16000000 // Clock Speed
#define BAUD 1200
#define MYBAUD (FOSC/16/BAUD)-1

const CHAR8 Task1Name[] PROGMEM = "Usa_serial_1";
const CHAR8 Task2Name[] PROGMEM = "Usa_serial_2";

PGM_P MainStringTable[] PROGMEM =
{
    Task1Name,
    Task2Name
};

BRTOS_Mutex *TestMutex;
BRTOS_Queue *Serial;

int main(void)
{
    BRTOS_Init();
    Serial_Init(MYBAUD);

    /*Argumentos da função que cria um Mutex: 1 - O endereço de um ponteiro do tipo
     “bloco de controle de mutex” (BRTOS_Mutex) que receberá o endereço do bloco de
     controle alocado para o mutex criado. 2 - A prioridade que será associada ao mutex
     e que deve ser sempre um nível maior do que a maior prioridade das tarefas que
     irão competir por um determinado recurso.3 - A tarefa que recebe o recurso passa a
     prioridade do mutex, não sendo interrompida pelas tarefas que competem pelo mesmo
     recurso. A tarefa volta a sua prioridade original quando liberar este recurso*/
}

if (OSMutexCreate(&TestMutex,7) != ALLOC_EVENT_OK)
    while(1){}// Oh Oh - Não deveria entrar aqui !!!

if(InstallTask(&Usa_serial_1,Task1Name,100,6) != OK)
    while(1){}// Oh Oh - Não deveria entrar aqui !!!

if(InstallTask(&Usa_serial_2,Task2Name,100,5) != OK)
    while(1){}// Oh Oh - Não deveria entrar aqui !!!
```

```

    // Start Task Scheduler
    if(BRTOSStart() != OK)
        while(1){}// Oh Oh - Não deveria entrar aqui !!!

    while(1);
}
//-----

```

2. tasks.c

```

#include "BRTOS.h"
#include "drivers.h"
#include "tasks.h"

extern BRTOS_Mutex *TestMutex;
extern BRTOS_Queue *Serial;

const CHAR8 SerialTeste1[] PROGMEM = "Esta eh a Tarefa 1";
const CHAR8 SerialTeste2[] PROGMEM = "Agora eh a Tarefa 2";

PGM_P TaskStringTable1[] PROGMEM = { SerialTeste1 };
PGM_P TaskStringTable2[] PROGMEM = { SerialTeste2 };

void Usa_serial_1(void)
{
    while(1)
    {
        strcpy_P(BufferText, (PGM_P)pgm_read_word(&(TaskStringTable1[0])));
        Serial_Envia_Frase((CHAR8*)BufferText);
        Serial_Envia_Caracter(LF);
        Serial_Envia_Caracter(CR);
        DelayTask(1);
    }
}
//-----
void Usa_serial_2(void)
{
    while(1)
    {
        // Adquire Mutex
        OSMutexAcquire(TestMutex);
        strcpy_P(BufferText, (PGM_P)pgm_read_word(&(TaskStringTable2[0])));
        Serial_Envia_Frase((CHAR8*)BufferText);
        Serial_Envia_Caracter(LF);
        Serial_Envia_Caracter(CR);

        // Libera Mutex
        OSMutexRelease(TestMutex);
        DelayTask(1);
    }
}
//-----

```

3. tasks.h

```
*****  
* OS Tasks  
*****  
#include "hardware.h"

void Usa_serial_1(void);
void Usa_serial_2(void);

void Transmite_Uptime(void);
void Transmite_Duty_Cycle(void);
void Transmite_RAM_Ocupada(void);
void Transmite_Task_Stacks(void);
void Transmite_CPU_Load(void);
void Reason_of_Reset(void);
```

Exercícios:

21.1 – Pesquise quais são os RTOSs disponíveis atualmente para os sistemas microcontrolados. Por que não se costuma utilizar um RTOS com microcontroladores de 8 bits?

21.2 – Repita o exercício 11.3 (capítulo 11), desta vez empregando o BRTOS.

21.3 – Por que os sistemas embarcados modernos exigem o uso de um RTOS?
