SISTEMAS OPERACIONAIS DE TEMPO REAL E SUA APLICAÇÃO EM SISTEMAS EMBARCADOS

GUSTAVO WEBER DENARDIN
CARLOS HENRIQUE BARRIQUELLO

Blucher

Gustavo Weber Denardin
Carlos Henrique Barriquello

Sistemas operacionais de tempo real e sua aplicação em sistemas embarcados

Sistemas operacionais de tempo real e sua aplicação em sistemas embarcados

Editora Edgard Blücher Ltda.

Ilustrações
Gustavo Weber Denardin e Carlos Henrique Barriquello

Imagem da capa
iStockphoto

Blucher

Rua Pedroso Alvarenga, 1245, 4° andar
04531-934 – São Paulo – SP – Brasil
Tel.: 55 11 3078-5366
contato@blucher.com.br
www.blucher.com.br

Segundo o Novo Acordo Ortográfico, conforme 5. ed. do *Vocabulário Ortográfico da Língua Portuguesa*, Academia Brasileira de Letras, março de 2009.

Dados Internacionais de Catalogação na Publicação (CIP)
Angélica Ilacqua CRB-8/7057

Denardin, Gustavo Weber

Sistemas operacionais de tempo real e sua aplicação em sistemas embarcados / Gustavo Weber Denardin, Carlos Henrique Barriquello. – São Paulo : Blucher, 2019.

474 p. : il.

Bibliografia
ISBN 978-85-212-1396-3 (impresso)
ISBN 978-85-212-1397-0 (e-book)

1. Processamento eletrônico de dados em tempo real 2. Sistemas embarcados (Computadores) 3. Sistemas operacionais (Computadores) I. Título II. Barriquello, Carlos Henrique

19-0129 CDD 005.43

Índice para catálogo sistemático:
1. Sistemas operacionais (Computadores)

Conteúdo

Capítulo 1

Introdução aos sistemas de tempo real

Sistemas operacionais de tempo real são uma subclasse de sistemas operacionais destinados à concepção de sistemas computacionais, geralmente embarcados, em que o tempo de resposta a um evento é fixo e deve ser respeitado sempre que possível. Esses sistemas são conhecidos na literatura como sistemas de tempo real e são caracterizados por possuírem requisitos específicos de sequência lógica e tempo que, se não cumpridos, resultam em falhas no sistema a que se dedicam. Ressalta-se que o tempo de resposta aos eventos controlados em um sistema de tempo real não deve ser necessariamente o mais rápido possível. A prioridade é o cumprimento dos prazos de todos os eventos controlados pelo sistema.

Existem dois tipos de sistemas em tempo real: *soft* e *hard*. Em um sistema de tempo real *soft*, o sistema pode continuar funcionando mesmo que restrições temporais não sejam respeitadas. Isso não significa que o sistema irá funcionar corretamente, mas que durante os períodos em que os prazos são perdidos ocorrerá inconsistência em seu funcionamento, voltando a operação correta quando os prazos voltarem a ser cumpridos. Um sistema de aquisição de dados pode ser considerado *soft*, pois possui restrições temporais, mas a perda de

prazos não implica falha geral do sistema. Já sistemas de tempo real *hard* devem seguir suas restrições temporais rigidamente, de forma a evitar o colapso total do sistema. Normalmente sistemas *hard* são utilizados em aplicações que estão diretamente relacionadas à vida de pessoas, como o sistema de controle de um avião. A maioria dos sistemas em tempo real existentes utilizam uma combinação de requisitos *soft* e *hard*.

Normalmente os sistemas em tempo real são embarcados. Isso significa que o sistema computacional é completamente encapsulado e dedicado ao dispositivo ou sistema que controla. Diferentemente de computadores de propósito geral, como o computador pessoal, um sistema embarcado realiza um conjunto de tarefas predefinidas, geralmente com requisitos específicos. Já que o sistema é dedicado a tarefas específicas, pode-se otimizar o projeto reduzindo tamanho, recursos computacionais e custo do produto/produção.

Alguns exemplos de sistemas embarcados são:

- Automotivos: controle de injeção eletrônica, controle de tração, controle de sistemas de frenagem antibloqueio (ABS) etc.;
- Domésticos: micro-ondas, lavadouras de louça, lavadouras de roupa etc.;
- Comunicação: telefones celulares, roteadores, equipamentos de GPS etc.;
- Robótica: robôs industriais, humanoides, *drones* etc.;
- Aeroespacial e militar: sistemas de gerenciamento de voo, controle de armas de fogo etc.;
- Controle de processos: processamento de alimentos, controle de plantas químicas e controle de manufaturas em geral.

Este capítulo irá discutir as principais motivações para a utilização de sistemas operacionais de tempo real na concepção de sistemas embarcados, destacando os principais impactos na utilização desse tipo de sistema frente às técnicas tradicionais de projeto. Ainda, descrevem-se os principais conceitos envolvidos em sistemas de tempo real. No texto os termos "processador" e "*central processing unit* (CPU)" serão utilizados de forma análoga para evitar o uso dos mesmos termos em longas sequências de texto.

1.1 Sistemas *foreground/background*

O ensino de projetos de sistemas embarcados usualmente se inicia com sistemas de baixa complexidade. Isso deve-se ao fato de ser o primeiro contato com uma mudança de paradigma de programação, em que em vez de se projetar um *software* sequencial, projeta-se um *software* que responda a eventos. Ademais, provavelmente será o primeiro contato com o projeto de um *software* que deve executar indefinidamente, enquanto o sistema estiver energizado. Existem várias formas de escrever um código que se comporte dessa maneira. No entanto, devido à sua simplicidade, normalmente começa-se utilizando o conceito *foreground/background*, também conhecido por superlaço. Nesse tipo de projeto, um laço infinito executa determinadas tarefas por meio da chamada de funções, sempre que a condição de solicitação de uma dessas tarefas ocorrer. Esse laço infinito é conhecido por *background*. Rotinas de serviço de interrupção (RSI) são utilizadas para o tratamento de eventos síncronos e assíncronos. Quando o sistema encontra-se nessas rotinas, define-se que se está em *foreground*.

As posições de *foreground* e *background* são também conhecidas por "nível de interrupções" e "nível de tarefas", respectivamente. A Figura 1.1 demonstra um diagrama temporal de operação de um sistema *foreground/background*.

Note que apesar do *foreground/background* ser um modelo de programação em que se desempenham tarefas a partir de testes de condições, a execução de tais testes e das funções que respondem a essas condições ainda é sequencial, somente sendo interrompida pelas rotinas de serviço de interrupção. Esse comportamento faz com que o tempo de resposta a uma condição dependa da posição atual do código no momento em que a condição se torna verdadeira. Portanto, para garantir os requisitos temporais de projeto, as operações críticas devem ser executadas pelas RSI. Como a maioria dos processadores permite alterar o nível de prioridade de suas interrupções, pode-se projetar a resposta temporal do sistema utilizando esse recurso. No entanto, como o código no âmbito de *background* somente será executado quando não houverem interrupções sendo tratadas, deve-se evitar que as rotinas de serviço de interrupção sejam longas a ponto de prejudicarem a execução do código em *background*.

Normalmente em sistemas utilizando superlaço as informações disponibilizadas pelas RSI são processadas no *background*, salvo nos casos em que as restrições temporais impeçam tal abordagem. Por exemplo, ao implementar um terminal de

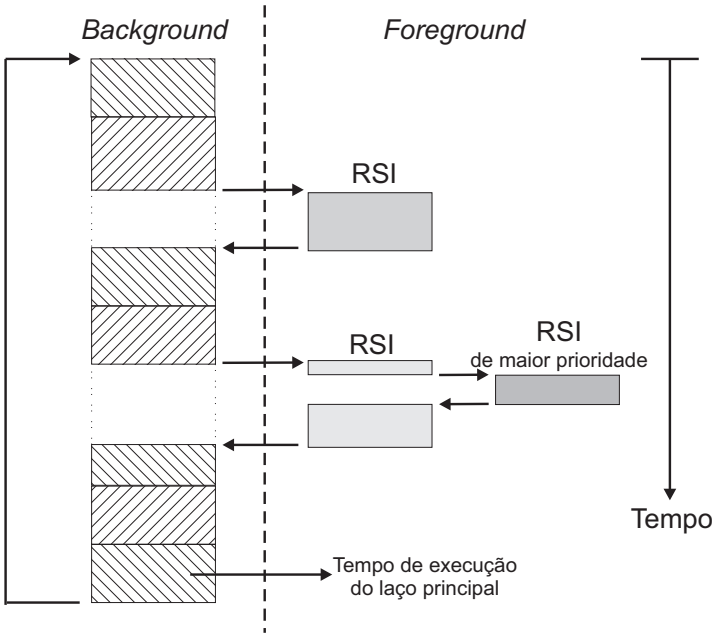


Figura 1.1 Exemplo de um sistema *foreground/background*.

Fonte Adaptado de Labrosse (1998).

comandos, o caractere digitado é recebido pela RSI, mas a ação relacionada a esse caractere é tomada em uma função executada no laço principal. O tempo para que essas informações sejam processadas é conhecido como "tempo de resposta de tarefa". O pior caso de tempo de resposta de tarefa em sistemas utilizando superlaço depende do tempo de execução do laço de *background*. É fácil perceber esse comportamento, pois o pior caso ocorre quando o evento a ser processado acontece logo após o seu teste no laço, sendo necessário um ciclo completo do laço para voltar a esse teste.

Nesse modelo de programação o tempo de execução de um ciclo do laço não é constante devido às tomadas de decisão inerentes ao código. Ainda, alterações no código afetam também o tempo de execução do laço. Assim, apesar do pior tempo de resposta de tarefa ser previsível, sucessivas execuções do laço podem apresentar grande variação de tempo de execução. Quando as tarefas do laço possuem grandes diferenças em tempo de execução, como a atualização de uma tela gráfica e a atualização de um terminal serial, a variação do tempo de execução do laço pode ir de poucos microssegundos a centenas de milissegundos.

Um outro aspecto que deve ser considerado quando projetando sistemas *foreground/background* é garantir que o código da RSI seja completamente executado antes que ocorra nova interrupção de um mesmo evento. No caso de ocorrer um agendamento de uma nova interrupção, antes que sua rotina de serviço da interrupção anterior seja completamente executada, poderá haver perda de dados e, consequentemente, as tarefas de *background* deixarão de ser alimentadas com esses dados. Ainda, no caso de ocorrer uma rajada de eventos, o processador poderá ficar excessivamente ocupado em modo *foreground*, prejudicando o tempo de resposta das tarefas em *background*.

Com o intuito de melhorar o tempo de resposta e otimizar o consumo de energia em sistemas utilizando superlaço, a abordagem mais utilizada é acionar os modos de baixo consumo de energia do processador no final do processamento do laço. A maioria dos processadores e microcontroladores permite configurar a saída do modo de baixo consumo de energia a partir da ocorrência de interrupções. Assim, executa-se todo o processamento por rotinas de serviço de interrupção e por funções em *background* acionadas por essas RSI.

A maioria das aplicações baseadas em microcontroladores ainda são projetadas utilizando-se o conceito *foreground/background*, principalmente quando se consideram cenários com baixa complexidade ou sem necessidade de rígidos requisitos em tempo de resposta. Isso deve-se aos desenvolvedores estarem acostumados com uma realidade em que as restrições de processamento e memória são muito grandes. Por esse mesmo motivo uma parcela considerável dos desenvolvedores de sistemas embarcados ainda tem pouco ou nenhum conhecimento em sistemas operacionais.

No entanto, nos últimos anos esse cenário tem se modificado. Atualmente sistemas embarcados são baseados em processadores e microcontroladores de alto desempenho com barramentos de 16, 32 e até 64 *bits*. Dispositivos de alto desempenho e baixo custo são oferecidos por uma grande variedade de fabricantes, geralmente com quantidades generosas de memória interna e uma grande gama de periféricos. Ademais, cada vez mais existem sistemas operacionais disponíveis, sejam estes gratuitos ou com licenças comerciais. Uma parcela de sistemas embarcados inclusive já permite o uso de sistemas operacionais de maior porte, como o Linux.

Apesar de a tecnologia atual viabilizar cada vez mais o uso de sistemas operacionais em sistemas embarcados, não significa que devemos fazer uso desse

recurso em todo e qualquer projeto. Uma das perguntas que se pretende responder durante este capítulo é por que devemos utilizar um sistema operacional de tempo real no projeto de sistemas embarcados.

1.2 Sistemas operacionais

Um sistema operacional (SO) é um *software* ou conjunto de *softwares* cuja função é servir de interface entre um sistema microprocessado e o usuário (podendo este ser o usuário final ou o projetista de um sistema embarcado). Há basicamente duas formas distintas de se conceituar um sistema operacional:

- visão *top-down*: pela perspectiva do usuário ou projetista de aplicativos, provendo abstração do *hardware*, ou seja, fazendo o papel de intermediário entre o aplicativo e o *hardware* do sistema microprocessado;
- visão *bottom-up*: opera como gerenciador de recursos, ou seja, controla quais tarefas podem ser executadas, quando podem ser executadas e que recursos (*display*, comunicação etc.) podem ser utilizados.

De forma simplista pode-se dizer que um sistema operacional possui as funções de gerenciamento de tarefas, gerenciamento de memória e gerenciamento de recursos. Outros recursos importantes que um sistema operacional pode prover são sistemas de arquivos (FAT, NTFS, EXT3, ReiserFS etc.) e implementação de protocolos de comunicação, como TCP/IP e *universal serial bus* (USB).

De maneira geral, a utilização de sistemas operacionais encontra-se bastante difundida. Encontramos sistemas operacionais em computadores pessoais, telefones celulares, videogames, centrais de entretenimento etc. No entanto, existem distinções no projeto de SO que os tornam mais susceptíveis ao uso em determinadas aplicações. Por exemplo, os sistemas operacionais destinados ao uso em computadores pessoais têm características diferenciadas dos sistemas utilizados em *smartphones* e sistemas embarcados com restrições de tempo real.

O objetivo do texto a seguir é discutir a utilização de sistemas operacionais em sistemas embarcados e que impactos sua utilização irá trazer, tanto para o desenvolvedor do sistema quanto para o usuário final. Conceitos clássicos sobre sistemas operacionais e exemplos de implementação serão apresentados. Ainda, pretende-se apontar alguns dos motivos pelos quais há uma tendência cada

vez maior de utilização de sistemas operacionais, a partir de uma análise das vantagens e desvantagens de seu uso em sistemas embarcados.

Um bom ponto de partida para analisar o porquê de se utilizar um sistema operacional é definir o que é um *software* embarcado de qualidade. Apesar desse assunto poder se estender demasiadamente, existem boas práticas que deveriam ser aplicadas a todo e qualquer sistema embarcado. Assim, pode-se dizer que sistemas embarcados devem:

- realizar as tarefas a que se destinam corretamente;
- realizar essas tarefas no tempo esperado;
- ter comportamento previsível;
- ter seu código desenvolvido de forma a facilitar a manutenção;
- permitir a análise estática de sintaxe e fluxo de dados do código, bem como a análise de comportamento do código em tempo de execução;
- ter desempenho em tempo de execução previsível;
- ter requisitos de memória previsíveis.

Essa lista de requisitos pode ser ampliada, dependendo das características de uma dada aplicação. Felizmente, os tópicos listados anteriormente são um bom ponto de partida para analisar o impacto de se utilizar um sistema operacional de tempo real na concepção de um sistema embarcado.

1.3 Motivações para sistemas operacionais de tempo real

À medida que os sistemas embarcados crescem em complexidade, a implementação de um sistema *foreground/background* torna-se inviável, ou pelo menos de difícil concepção e manutenção. Ainda, certos sistemas embarcados necessitam de conexão com interfaces Ethernet, *wireless*, pilha TCP/IP, USB, RS-232, *displays*, memórias, entre outros. Portanto, o *software* a ser desenvolvido deve interagir com cada um desses dispositivos. Entretanto, como muitos desses dispositivos são comuns a vários projetos de sistemas embarcados, é importante reusar o máximo possível do *software* desenvolvido para evitar a replicação, a cada novo projeto, do esforço já dispendido por algum outro engenheiro de

software. Somando-se a isso os problemas relacionados à interação entre esses recursos, dificilmente pode-se desenvolver soluções que permitam a reutilização de código com o emprego da técnica do superlaço.

Ao se projetar um sistema embarcado, deve-se implementar um agendamento de tarefas. Uma forma bastante simples de se implementar esses agendamentos consiste na utilização de um laço infinito contendo diversos testes de *flags*. Um *flag* pode ser visto como um semáforo, que indica se devemos avançar e executar um determinado código ou se devemos esperar para executá-lo. A Listagem 1.1 apresenta um exemplo de agendamento de processos utilizando *flags* em um laço infinito.

```
for (;;){
   // Somente atualiza o display se display_flag = 1
   if (display_flag == 1){
      display_flag = 0;
      atualiza_display();
   }

   /* Somente processa se houver tecla pressionada,
   ou seja, se teclado_flag = 1 */
   if (teclado_flag == 1){
      teclado_flag = 0;
      trata_teclado();
   }

   /* Somente atualiza o relógio se a interrupção de tempo
   configurar relogio_flag = 1 */
   if (relogio_flag == 1){
      relogio_flag = 0;
      atualiza_relogio();
   }
}
```

Listagem 1.1 Agendamento de processos por *flags*.

No método apresentado, a indicação de um *flag* (que nada mais é do que a troca de estado de um *bit*) pode ser gerado por uma interrupção. No caso de um relógio digital, por exemplo, uma interrupção associada a um temporizador assinala com o *flag* a cada 1 segundo que o relógio deve ser atualizado.

Embora esse método seja satisfatório para sistemas mais simples, como um forno de micro-ondas, algumas desvantagens podem ser citadas. No caso do forno, cada tarefa pode ser executada até que seja concluída. O código de processamento do teclado ocupa o tempo que for necessário para tratar as entradas de usuários pelo teclado. Novamente, nesse caso as tarefas são muito simples, e o tempo de processamento mais longo para a tarefa mais complexa é ainda muito curto para causar um problema.

Para melhor analisar o problema do desenvolvimento de sistemas embarcados deve-se partir do princípio mais fundamental nesse tipo de sistema. Em sistemas eletrônicos analógicos (tempo contínuo), todas as operações podem, caso necessário, ser executadas ao mesmo tempo, ou seja, concorrentemente. Ainda, as operações são em sua maior parte executadas instantaneamente. Obviamente isso não é possível em sistemas baseados em processadores, pois processadores só podem realizar uma operação por vez, já que são máquinas sequenciais. Complementarmente, essas operações levam tempo, ou seja, as operações não são executadas instantaneamente. Esses princípios são fundamentais para entender que as necessidades temporais do sistema em desenvolvimento são de grande importância para o projeto de um sistema embarcado.

Em um sistema simples, como um controlador de temperatura, pode-se facilmente determinar um conjunto de tarefas que descrevem sua operação. Por exemplo, o sistema deverá adquirir a temperatura do local/material que se pretende controlar a partir de um sensor de temperatura, ajustar/linearizar o sinal, compará-lo com a temperatura desejável, gerar um sinal de controle para o atuador do sistema e esperar um determinado tempo antes de voltar a executar esse conjunto de tarefas. Verifica-se que esse sistema apresenta algumas características básicas, como:

- executa em um laço contínuo;
- completa sua operação em cada passagem do laço;
- leva um determinado tempo para completar uma operação;
- repete sua operação periodicamente;
- não realiza outras tarefas enquanto espera pela próxima execução;
- necessita de uma mecanismo de controle de tempo para gerenciar sua periodicidade, sendo que a maioria das implementações utiliza um temporizador em *hardware* para isso.

Nesse exemplo, o período de execução do laço e o tempo máximo para completar uma operação são requisitos do sistema. Já o tempo de execução de uma operação e o tempo restante (tempo ocioso) dependem de como o código foi implementado e do *hardware* utilizado. Se considerarmos que o período entre ativações desse sistema de controle de temperatura é de 50 ms e que o tempo de execução de uma operação é de 5 ms, facilmente conclui-se que a ocupação do

processador é de 10%. Três observações podem ser realizadas a partir desse exemplo: (1) em sistemas embarcados as tarefas devem ser realizadas completamente a cada vez que são ativadas; (2) em um sistema prático deve haver algum tempo livre; (3) possuir um período que determina o espaçamento entre ativações não significa que necessariamente se tem o tempo total entre ativações para completar um operação, ou seja, o atraso excessivo de processamento das informações pode causar problemas com o comportamento geral do sistema.

O exemplo anterior de sistema permite construir uma solução ótima para o *software* com o mínimo possível de complexidade a partir de uma estrutura de laço infinito. No entanto, percebe-se que tal sistema consiste somente em uma operação bem definida. Quando é preciso aumentar a escala do sistema embarcado para múltiplas operações, outras preocupações surgem no projeto. No entanto, o uso de laço infinito no projeto não deve ser descartado inicialmente, mesmo em projetos de sistemas com múltiplas operações.

Suponha que se deseja conceber o *software* para o controle de um quadricóptero, em que é necessário realizar controles de arfagem, rolagem e guinada. Cada um desses controles individualmente pode ser visto como o mesmo problema do controle de temperatura do exemplo anterior. No entanto, estabilizar o quadricóptero no ar depende da realização das três operações. Assume-se que os requisitos de tempo para esses controles são:

- o controle de arfagem deve ser realizado 25 vezes por segundo (período de 40 ms);
- o controle de rolagem deve ser realizado 50 vezes por segundo (período de 20 ms); e
- o controle de guinada deve ser realizado 100 vezes por segundo (período de 10 ms).

O *software* de controle desse sistema pode ser desenvolvido a partir da extensão da abordagem utilizada para o controle de temperatura. No entanto, além de realizar a execução periódica das três operações, deve-se realizá-la com diferentes taxas. O exemplo apresentado na Listagem 1.2 é uma possível abordagem para realizar os três controles em diferentes taxas. Note que agora é necessário que haja uma coordenação entre as operações, realizada pela análise da variável *flag*.

A implementação do controle de temperatura e do controle de quadricóptero concebida dessa maneira não necessita nem mesmo de interrupções de um

```
int flag = 0;

while(1){
  swith(flag){
    case 0:
      ExecutaControleGuinada();
      ExecutaControleRolagem();
      ExecutaControleArfagem();
      flag = 1;
      break;

    case 1:
      ExecutaControleGuinada();
      flag = 2;
      break;

    case 2:
      ExecutaControleGuinada();
      ExecutaControleRolagem();
      flag = 3;
      break;

    case 3:
      ExecutaControleGuinada();
      flag = 0;
      break;
  }
  // Espera de 10 ms
  Espera_ms(10);
}
```

Listagem 1.2 Exemplo de solução para a execução concorrente de três controladores de um quadricóptero com diferentes taxas de ativação.

hardware temporizador, pois pode ser realizada por um simples contador ou até mesmo por contagens de ciclos de *clock* do processador, para controlar a periodicidade da execução das operações. Note que existem níveis de aplicações críticas em que não se permite o uso de interrupções. Por exemplo, o controle de temperatura em um reator nuclear.

Todavia, ao considerar maior complexidade no sistema em desenvolvimento, a abordagem utilizada até o momento começa a apresentar suas deficiências. Considerando-se uma série de tarefas secundárias ao controle do quadricóptero, como um sistema de rádio frequência para o controle do equipamento, uma câmera para capturar imagens aéreas e um sistema de armazenamento em memória não volátil de informações do voo, surgem novos desafios no projeto desse sistema embarcado. Assim, baseando-se nos requisitos do sistema, pode-se iniciar o projeto do *software* embarcado pelo desenvolvimento individual de cada uma das operações que devem ser realizadas. No entanto, é improvável que a união dos trechos individualizados de código funcione corretamente utilizando o método proposto anteriormente.

O problema em unir as diferentes operações do sistema embarcado proposto é o paralelismo de tarefas assíncronas, ou seja, a existência de concorrência entre tarefas que são independentes e que não possuem períodos de ativação múltiplos. A análise desse problema fica mais clara quando se percebe que:

- existem operações que devem ser executadas em intervalos regulares;
- existem operações que são ativadas aleatoriamente (tarefas aperiódicas);
- existem operações que devem ser executadas simultaneamente;
- existem operações de longa duração na execução;
- e que os requisitos de tempo de cada operação são muito diferentes.

Uma análise mais profunda dos requisitos temporais permite identificar as principais dificuldades existentes na técnica de projeto anterior para o caso proposto. Os laços de controle de arfagem, rolagem e guinada possuem período de 25, 50 e 100 ms, respectivamente. Imagine que o tempo de execução de cada controlador seja de 2 ms. As tarefas secundárias podem ser caracterizadas a título de exemplo como:

- controle por radiofrequência: conexão com um rádio por comunicação *serial peripheral interface* (SPI) e um pino de I/O para informar a chegada de um pacote. Os pacotes de controle do rádio têm 20 *bytes* e a taxa de transmissão da SPI está configurada para 500 kb/s. Note que um novo comando pode chegar a cada 312,5 μs;
- leitura dos sensores: antes de os laços de controle serem executados, a informação já deve ter sido lida. Assim, os dados dos sensores como giroscópio e acelerômetro são adquiridos por uma conexão I2C (*inter-integrated circuit*) a 400 MHz. Os dados dos sensores devem ainda ser filtrados e pós-processados para então serem aproveitados pelos algoritmos de controle;
- conexão com a câmera: conexão por meio de interface DCMI (*digital camera memory interface*). A conexão de dados é paralela e existem dois pinos de controle para sincronia dos quadros de imagem. Uma vez que a captura de imagem é disparada, quadros de imagem devem ser adquiridos pela interface a uma taxa de 54 MHz, em que a cada dois ciclos um valor de 32 *bits* é adquirido. O tempo total para aquisição de uma imagem pode atingir até 1,5 ms. Após a aquisição da imagem os dados são salvos em um cartão

SD. A taxa de escrita no cartão é de 500 kB/s, o que significa um tempo total de 600 ms para salvar uma imagem;

- conexão com a memória não volátil: conexão com memória *flash* por SPI a uma taxa de 1 Mb/s. Os dados de giroscópio, acelerômetro e outros sensores devem ser salvos a cada 500 µs para análises posteriores de padrões de voo; o tempo de escrita dos dados na *flash* pode ser de até 100 µs.

Se existem operações com taxas de execução na faixa de microssegundos e o método de *polling* for utilizado como apresentado anteriormente, a taxa de verificação de cada operação deve ser muito alta. Por exemplo, pelo menos a cada 312,5 µs para retirar os dados do rádio antes que um novo pacote de dados seja recebido. Ou, ainda, pelo menos a cada 500 µs para atender aos requisitos de armazenamento da memória não volátil. A conclusão é que se torna praticamente impossível projetar o *software* desse sistema embarcado de uma maneira simples. Nesses casos o uso de interrupções é imprescindível para o projeto do sistema.

Interrupções são a forma de o *hardware* informar que um determinado periférico necessita atenção. Eventos típicos do mundo real que ocorrem em um sistema microprocessado e geram interrupções são: a transição de estado de um pino de entrada/saída conectado a um teclado, dados sendo recebidos por uma porta serial, um temporizador expirando sua contagem, entre outros. Quando uma interrupção ocorre, um código específico, projetado pelo desenvolvedor, deve atender a solicitação do periférico que a gerou. Esse trecho de código, conforme visto anteriormente, chama-se rotina de serviço de interrupção. Muitos sistemas embarcados são projetados tendo na coordenação de tratamento de eventos as rotinas de serviço de interrupções.

Se for aplicado o método de gerenciamento por eventos/interrupções no exemplo do sistema embarcado do quadricóptero, verifica-se o seguinte comportamento:

- o controle de arfagem é ativado por uma interrupção de um temporizador a cada 40 ms;
- o controle de rolagem é ativado por uma interrupção de um temporizador a cada 20 ms;
- o controle de guinada é ativado por uma interrupção de um temporizador a cada 10 ms;

- a leitura dos pacotes do rádio é realizada a partir de uma interrupção gerada pela transição de um pino de entrada/saída. A transição de estado no pino é proveniente do próprio rádio ao receber um pacote de dados por radio frequência;
- a leitura dos sensores é realizada a partir de uma interrupção de um temporizador, programada para ocorrer a cada 500 µs, conforme determinado pela operação de salvamento de dados de voo na memória não volátil do sistema;
- a aquisição das imagens pela câmera não é disparada por uma interrupção, mas sim por uma requisição do rádio.

O que se obtém ao utilizar essa abordagem é um conjunto de operações, realizadas por determinados trechos de código que são ativados por interrupções e que cooperam entre si. Pode-se perceber facilmente que se somente um processador está disponível no sistema, essas operações não podem ocorrer concorrentemente. Assim, no instante em que um desses trechos de código for disparado por um evento, os recursos computacionais disponíveis serão direcionadas para completar essa operação.

Portanto, em um sistema baseado em eventos ocorre a aparente execução concorrente de um conjunto de tarefas individuais. Apesar de as interrupções facilitarem o projeto de um sistema embarcado, na maioria dos casos o comportamento do sistema em tempo de execução não pode ser previsto ou analisado. Uma das maiores dificuldades em se utilizar um sistema embarcado dirigido por interrupções é que os desenvolvedores devem ser altamente qualificados, tanto em *hardware* quanto em *software*. Isso ocorre principalmente porque a forma com que as tarefas partilham o uso do processador em tempo é muito restritiva. Dessa maneira, quando a abordagem por interrupções é empregada por desenvolvedores inexperientes, o resultado em termos de desempenho e comportamento pode deixar a desejar.

Outro potencial problema com tarefas independentes e com diferentes requisitos de tempo em um sistema baseado em prioridades é a definição de prioridades. Caso os códigos sejam todos executados nas rotinas de serviço de interrupção, existe a possibilidade de configurar parcialmente os níveis de prioridade. No entanto, com a abordagem *foreground/background*, todas as operações possuem a mesma prioridade. A impossibilidade de se projetar um sistema embarcado com

prioridades pode fazer com que um sistema seja não realizável, uma vez que tarefas com grandes restrições temporais podem ser atrasadas pelo processamento de uma tarefa com menor importância quanto ao cumprimento de prazos.

Com o objetivo de suprir as deficiências de sistemas *foreground/background* para aplicações de maior complexidade e simplificar o projeto de sistemas por parte do desenvolvedor, surge o conceito de sistemas operacionais de tempo real (do inglês *real-time operating systems* – RTOS). Esses sistemas minimizam a complexidade de sistemas processados para o desenvolvedor, permitindo que ele se concentre nos principais objetivos do sistema em desenvolvimento. Ainda, se os *drivers* já estiverem disponíveis no sistema, não há nem mesmo necessidade de se conhecer detalhes de periféricos como temporizadores e conversores analógico/digital e até mesmo interrupções.

Um grande número de sistemas embarcados possuem restrições temporais bem definidas, pois são criados especificamente para um propósito. Geralmente esses sistemas interagem com eventos externos, que podem ser únicos ou múltiplos, ou, ainda, síncronos ou assíncronos. Quanto maior a quantidade de eventos a serem tratados, maior a complexidade do sistema e mais difícil se torna o projeto baseado em superlaço. Um sistema embarcado deve receber esses eventos, tratá-los e tomar decisões baseadas nesses eventos, provendo respostas em período hábil e limitado, independentemente de quantos e quais eventos ocorrerem. Portanto, o sistema deve ser o mais determinístico possível. Essa característica da maioria dos sistemas embarcados incentivou a criação de sistemas operacionais que possibilitassem o cumprimento de tais prazos da melhor forma possível.

Assim, um RTOS deve gerenciar os recursos limitados de um sistema embarcado e tentar cumprir da forma mais eficiente possível suas restrições temporais. No entanto, somente utilizar um RTOS não necessariamente faz com que o sistema opere de forma determinística. O projetista deve utilizar os recursos do RTOS e técnicas de projeto de sistemas de tempo real de forma a obter o comportamento desejado para o sistema projetado.

Um RTOS pode ser implementado de duas formas básicas: somente o núcleo (mais conhecido como *kernel*) ou o sistema operacional completo. O núcleo geralmente implementa os serviços básicos do sistema, como gerenciamento de tempo e memória, semáforos, filas, entre outros. Um sistema operacional completo deve possuir *drivers* para diversos periféricos, sistema de arquivos, pilhas de comunicação, entre outros recursos comumente utilizados. Apesar de um sistema mais

completo trazer facilidades ao usuário, nada impede que desenvolvedores criem seus próprios *drivers* e pilhas de protocolos utilizando as facilidades providas pelo sistema operacional.

Uma característica comum dos RTOS é a de que o *hardware* do sistema deve gerar uma interrupção periódica, conhecida por marca de tempo (ou, ainda, *timer tick*). Essa marca de tempo é utilizada para o gerenciamento de tempo, agendamento de tarefas e outras funções básicas do sistema. No entanto, existem sistemas operacionais que não requerem interrupções de tempo periódicas, operando completamente a partir de eventos assíncronos.

Um sistema operacional de tempo real suporta tipicamente as seguintes funções:

- multitarefas, que normalmente implica:
  - ativar e desativar tarefas;
  - configurar prioridade de tarefas;
  - agendamento de tarefas;
- comunicação e sincronização entre tarefas;
- gerenciamento de memória;
- gerenciamento de tempo.

A seguir, os principais conceitos relativos a sistemas operacionais de tempo real serão abordados. A partir do melhor entendimento desses conceitos, será possível analisar como os recursos disponibilizados por essa modalidade de sistema operacional pode ajudar no projeto de sistemas embarcados e no cumprimento de suas restrições temporais.

1.4 Definições relativas a sistemas operacionais

1.4.1 Tarefas, corrotinas, processos e *threads*

O conceito de tarefas, corrotinas, processos e *threads* é muitas vezes erroneamente interpretado. Portanto, vamos separar nesta seção a análise tradicional desses conceitos e o que geralmente se aplica a sistemas operacionais de tempo real.

Em sistemas operacionais tradicionais, os termos mais utilizados em referência a estruturas de concorrência são "processos" e "*threads*", apesar de nada impedir o uso de corrotinas. Nesses sistemas, um processo é um agrupamento de recursos relacionados, por exemplo: memória, arquivos abertos, *threads* etc. Já as *threads* são o fluxo de execução do processo e são escalonadas pelo sistema operacional. Consequentemente, são as *threads* que possuem contexto e pilha (do inglês *stack*). As *threads* compartilham o espaço de memória alocado pelo processo, arquivos abertos etc. Nesse tipo de sistema operacional mais tradicional, um processo pode conter uma ou mais *threads*.

No entanto, quando falamos em sistemas operacionais de tempo real, principalmente em sistemas embarcados, o conceito de processo possui uma complexidade incompatível com a maioria das aplicações. Portanto, geralmente os RTOS utilizam muitas vezes a nomenclatura de tarefas e *threads*. Nesse contexto, uma tarefa não é equivalente a um processo, muito menos uma *thread* tem o mesmo conceito anteriormente definido. Analisando de forma mais simplista, não existem processos quando nos referimos a um RTOS, e as tarefas assumem o papel anteriormente definido para as *threads*.

Assim, uma tarefa tem como atribuição executar uma ação por meio de uma sequência de instruções. Tarefas podem realizar funções de sistema, como gerenciar o *driver* de um periférico, ou funções definidas pelo usuário/desenvolvedor, implementando uma funcionalidade específica do sistema embarcado concebido. A principal característica de uma tarefa é ser uma entidade que o sistema operacional pode executar concorrentemente. Dessa forma, podemos dizer que uma tarefa é um programa simples que pensa possuir o processador somente para si e que, ao ser instalado no sistema, recebe a sua própria área de pilha. Esse conceito é praticamente o mesmo aplicado a *threads* em sistemas operacionais tradicionais.

A Figura 1.2 demonstra um sistema com múltiplas tarefas compartilhando um mesmo processador. Note que cada tarefa possui sua pilha, sendo utilizada para alocar variáveis locais, armazenar informações de retorno das funções e interrupções executadas durante sua ativação, e armazenar o contexto da tarefa quando interrompida. O contexto da tarefa são os dados contidos nos registradores de propósito geral do processador, bem como de seus registradores especiais, por exemplo, o contador de programa e o registrador de estado do processador. Verifica-se que quando uma tarefa assume o processador, as informações contidas

nos registradores em sua última ativação (ou seja, seu contexto) são carregadas novamente nos registradores do processador. A posição da pilha da qual se busca essa informação é definida pela última posição conhecida do ponteiro de pilha (*stack pointer*) da tarefa sendo restaurada, que é armazenada em uma estrutura de dados especial do sistema operacional conhecida por bloco de controle de tarefa. Portanto, não só é necessário possuir uma pilha por tarefa como também é necessário armazenar a posição atual do ponteiro de pilha de cada tarefa.

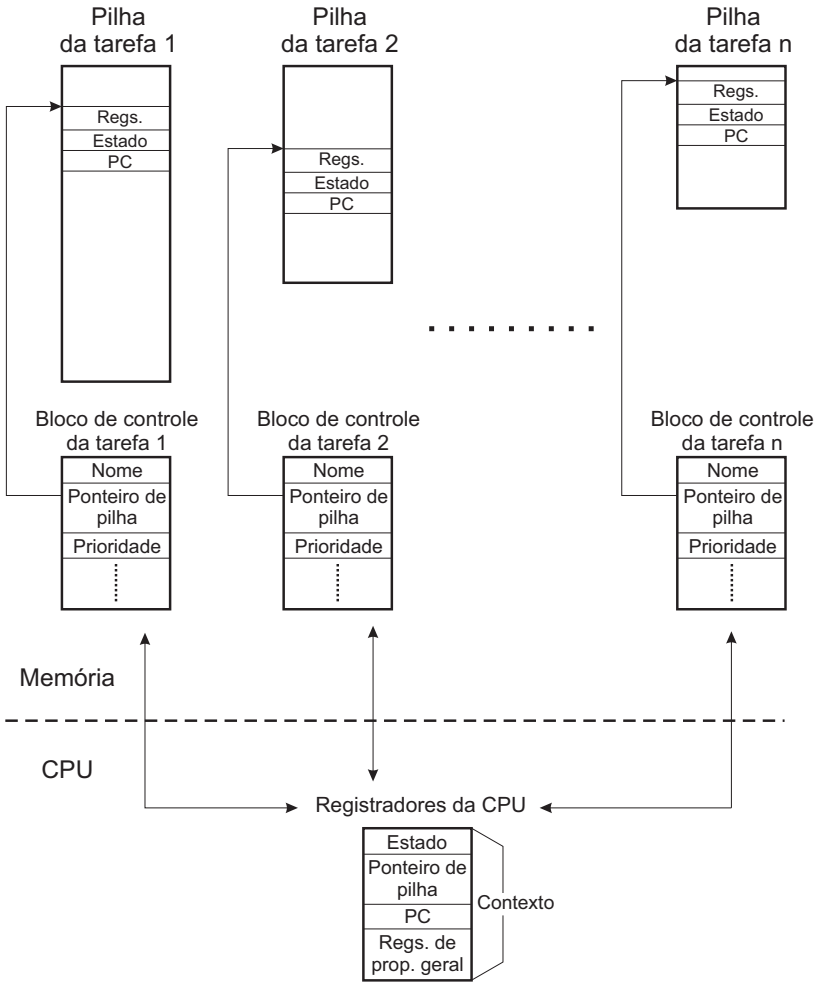


Figura 1.2 Alocação de múltiplas tarefas.

Fonte Adaptado de Labrosse (1998).

Nesses sistemas mais simples não existe obrigatoriamente a necessidade de se criar regiões de memória protegida, como no caso de processos. Assim, apesar

de tarefas possuírem sua própria pilha, compartilham a memória global do sistema sem nenhuma restrição. Isso não impede que certos RTOS implementem a proteção de regiões da memória, mas normalmente esse recurso está associado a um suporte oferecido pelo *hardware* utilizado.

As tarefas tipicamente se apresentam como laços infinitos que podem estar em cinco condições ou estados: execução, pronto para execução, espera, interrompido e inativo. Quando a tarefa está no estado de execução, recebe os recursos do processador do sistema e realiza as ações a que se destina. Já no estado "pronta para execução", a tarefa está pronta para ser executada, embora ainda não tenha recebido os recursos do processador. No estado de espera a tarefa para de executar suas ações e passa a esperar por um determinado evento. Quando uma rotina de serviço de interrupção é executada, necessariamente precisa interromper a tarefa em execução. O estado inativo ocorre quando uma tarefa encontra-se na memória, mas não foi disponibilizada para o agendador de tarefas. A Figura 1.3 apresenta os estados das tarefas em um sistema multitarefas, bem como as possíveis alterações entre estados.

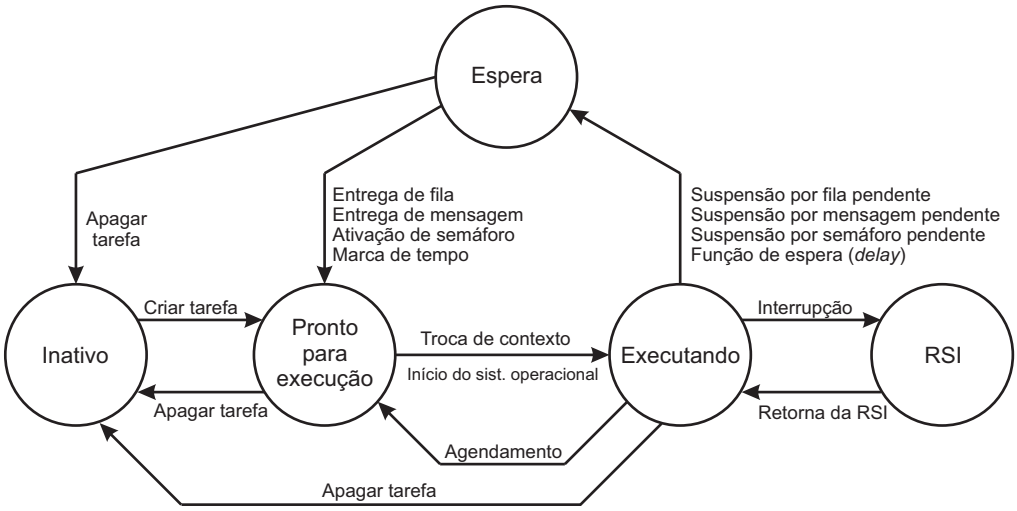


Figura 1.3 Estados das tarefas.

Fonte Adaptado de Labrosse (1998).

As corrotinas são similares às tarefas, também possuem prioridade, estados e uma função que é executada no momento em que a corrotina assume o controle do processador. No entanto, existem diferenças importantes entre tarefas e corrotinas. Uma das mais significativas está relacionada à pilha da tarefa, que não existe

nas corrotinas, ou seja, uma corrotina não possui contexto próprio. Assim, uma corrotina pode armazenar um valor em um registrador antes de ser interrompida e, quando retornar à sua execução, o valor dessa variável pode ou não ser o mesmo, pois outra corrotina pode ter alterado o valor do registrador durante sua execução.

Enquanto o conceito de tarefa é tipicamente associado a um escalonamento preemptivo, ou seja, a cessões involuntárias do controle do processador, o escalonamento de corrotinas é essencialmente cooperativo, pois uma corrotina deve suspender explicitamente sua execução para permitir que outra corrotina possa executar. Assim, o escalonador somente deve decidir qual corrotina deve assumir o processador no momento em que a corrotina em execução interrompe-se. A política de escolha pode ser baseada em prioridades, em ordenações (FIFO, por exemplo), entre outras.

Uma implementação convencional de um ambiente de gerência para corrotinas pode implicar um comportamento indesejado para certas aplicações. Se, por exemplo, uma corrotina é bloqueada ao realizar uma operação de entrada ou saída (como escrever em um *display* ou ler dados de um conversor A/D), nenhuma outra corrotina receberá o controle do processador. Assim, todo o sistema é bloqueado, o que pode comprometer consideravelmente seu desempenho. Note que, para esse bloqueio ocorrer, a corrotina deve acessar um código de espera que não ceda o processador ou acessar um objeto de sincronismo do sistema vinculado às tarefas.

Apesar disso, uma solução para esse problema geralmente é implementada em bibliotecas de entrada e saída na maioria dos sistemas com suporte a corrotinas. Nessas bibliotecas é possível associar um tempo máximo de espera para o término de uma operação. Assim, ao realizar uma operação de entrada ou saída com esse recurso a corrotina é suspensa, permitindo que o gerenciador de corrotinas entregue o processador a outra corrotina. Quando a corrotina interrompida é reativada, a função de entrada ou saída é retomada, finalizando sua operação ou suspendendo novamente a corrotina se a operação não se completou no tempo especificado.

Uma biblioteca de corrotinas em linguagem C foi desenvolvida, por Simon Tatham, utilizando a diretiva *switch* de forma não trivial (TATHAM, 2000). A implementação se baseia no conceito de continuação local, que representa um ponto de "retorna e continua" em uma função. Assim, a função deve apresentar

múltiplos pontos de entrada e saída. Por exemplo, considerando a função ***func()*** da Listagem 1.3, gostaríamos de poder executá-la de forma que, a cada execução, o valor de retorno fosse dependente da execução anterior. Isto é, que a função ***func()*** retornasse, na primeira execução, o valor 0, na segunda, 1, na terceira, 2, e assim sucessivamente. Para conseguir esse efeito, é necessário modificar a função ***func()*** para que ela possa armazenar seu estado de execução, bem como os dados a ele associados, conforme mostrado na Listagem 1.4. Note que a função passou a conter uma variável "estado" que mantém seu estado de execução e que é utilizada para selecionar o ponto de continuação com as diretivas *switch* e *goto*. Além disso, as variáveis foram declaradas com o qualificador *static* para que sejam preservadas entre execuções consecutivas da função.

```
int func(void){
  int i = 0;
  for(i = 0; i < 10; i++){
    return i;
  }
}
```

Listagem 1.3 Função que deve apresentar uma continuação local.

```
int func(void){
  static int i, estado = 0;
  switch(estado){
    case 0: goto LABEL0;
    case 1: goto LABEL1;
  }
  LABEL0: /* primeira execução da função inicia neste ponto */
  for(i = 0; i < 10; i++){
    estado = 1;
    return i;
    LABEL1:;/* próximas execuções continuam a partir deste ponto */
  }
}
```

Listagem 1.4 Função com continuação local.

Embora essa solução possa ser utilizada para implementar os pontos de continuação local das corrotinas, o código tem um custo alto de manutenção, pois deve-se acrescentar um novo rótulo e uma nova entrada na seleção no bloco de *switch* para cada ponto de continuação local a ser incluído na corrotina. Por isso, Tatham se baseou em uma forma não trivial de se usar o *switch-case* para reescrever o código da corrotina de forma mais enxuta, a qual foi proposta primeiramente por Tom Duff e usada em seu famoso *dispositivo de Duff* (DUFF, 1988). Dessa forma, consegue-se utilizar o posicionamento da diretiva *case* do

switch e realizar conjuntamente a seleção e o salto até o ponto de continuação local, evitando, portanto, o uso do *goto*, conforme mostrado na Listagem 1.5.

```
int func(void){
  static int i, estado = 0;

  switch(estado){
    case 0: /* primeira execução da função inicia neste ponto */
    for(i = 0; i < 10; i++){
      estado = 1;
      return i;
      case 1: ; /* próximas execuções continuam a partir deste ponto */
    }
  }
}
```

Listagem 1.5 Função com continuação local usando diretamente *switch*.

Note que uma diretiva *case* foi colocada dentro do laço *for*, e isso, embora não seja usual, é válido na linguagem C. Além disso, a variável "estado", que é utilizada para identificar o ponto de continuação local, precisa assumir um valor único para cada ponto, porém a escolha do valor é arbitrária. Por isso, é possível utilizar a macro __LINE__ do pré-processador da linguagem C para gerar automaticamente o valor de "estado" e da respectiva diretiva *case*, e ainda "esconder" todos esses detalhes em macros de pré-processador, conforme apresentado na Listagem 1.6.

```
#define crBegin   static int estado=0; switch(estado) { case 0:

#define crReturn(x) do { estado=__LINE__; return x; \
                      case __LINE__:; } while (0)
#define crFinish    }

int func(void){
  static int i;
  crBegin;
  for (i = 0; i < 10; i++)
    crReturn(i);
  crFinish;
}
```

Listagem 1.6 Função com continuação local usando *switch* por meio de macros de pré-processador.

Essa implementação de corrotinas pelo emprego do *switch* serviu como base para a criação da biblioteca de corrotinas do sistema operacional de tempo real FreeRTOS [1] e da biblioteca Protothreads,[2] a qual é utilizada no sistema Contiki [3]

[1] O *kernel* FreeRTOS foi criado por David Barry em 2003. Ver freertos.org.

[2] Ver www.dunkels.com/adam/pt.

[3] Contiki é um sistema operacional *open source* para a internet das coisas. Ver www.contiki-os.org.

e na implementação da pilha de TCP/IP uIP.[4] As corrotinas implementadas pela biblioteca Protothreads são também denominadas *protothreads*.

As corrotinas implementadas com a diretiva *switch* ocupam pouca memória de dados para manter o contexto de execução, que consiste basicamente no número da linha associado ao ponto de continuação local e, em geral, pode ocupar apenas dois *bytes* de RAM por corrotina. Note que as variáveis locais não são mantidas, sendo necessário utilizar variáveis estáticas ou globais caso haja necessidade de preservar as variáveis entre execuções consecutivas. Além disso, a implementação é altamente portável, uma vez que não requer codificação de instruções diretamente em *assembly*. Por outro lado, tal implementação tem a limitação de não permitir o uso do *switch* no código da corrotina. Caso se pretenda utilizar a diretiva *switch*, uma implementação alternativa para o compilador GCC está disponível na biblioteca Protothreads e faz uso de uma extensão do compilador que permite utilizar rótulos como ponteiros. Na Listagem 1.7 tem-se algumas das macros disponíveis na biblioteca Protothreads. Já a Listagem 1.8 apresenta um exemplo de uso dessa biblioteca, e a Listagem 1.9 demonstra como ficará o código resultante após a expansão das macros.

```
1 struct pt { unsigned short lc; };
  #define PT_THREAD(name_args)  char name_args
3 #define PT_BEGIN(pt)          switch(pt->lc) { case 0:
  #define PT_WAIT_UNTIL(pt, c)  pt->lc = __LINE__; case __LINE__: \
5                               if(!(c)) return 0
  #define PT_END(pt)            } pt->lc = 0; return 2
7 #define PT_INIT(pt)           pt->lc = 0
```

Listagem 1.7 Macros de pré-processador da biblioteca Protothreads com uso do *switch*.

```
1 static PT_THREAD(exemplo(struct pt *pt)){
    PT_BEGIN(pt);
3
    while(1) {
5     PT_WAIT_UNTIL(pt, contador == 1000);
      printf("Contagem máxima alcançada.\n");
7     contador = 0;
    }
9   PT_END(pt);
  }
```

Listagem 1.8 Exemplo de *protothread* com uso do *switch* por meio de macros da biblioteca Protothreads.

Apesar de corrotinas terem limitações, usualmente consomem menos recursos do sistema, o que as torna interessantes para sistemas com limitações

[4] Adam Dunkels. The uIP Embedded TCP/IP Stack The uIP 1.0 Reference Manual. 2006

```
static char exemple(struct pt *pt){
  switch(pt->lc) { case 0:

    while(1) {
      pt->lc = 8; case 8:
      if(!(contador == 1000)) return 0;
      printf("Contagem máxima alcançada.\n");
      contador = 0;
    }
  } pt->lc = 0; return 2;
}
```

Listagem 1.9 Exemplo de *protothread* com uso do *switch* com as macros da biblioteca Protothreads expandidas.

extremas de memória e capacidade de processamento. Por outro lado, atualmente a maioria dos processadores e microcontroladores dispõe de recursos suficientes para executar um sistema multitarefas completo, deixando as corrotinas em segundo plano para a maioria das implementações.

1.4.2 Prioridades

O gerenciamento de um sistema multitarefas pode se tornar bastante complexo se várias tarefas precisarem ser executadas ao mesmo tempo em um determinado momento. Para ajudar nessa gerência de tempo e recursos, regras precisam ser definidas. Uma das possíveis maneiras de se implementar regras é a partir de prioridades de execução. Dessa forma é possível definir a ordem de execução das tarefas que concorrem à execução em um mesmo momento.

Em um sistema operacional de tempo real podem existir dois cenários: sistemas que permitem múltiplas tarefas com uma mesma prioridade ou sistemas que somente suportam uma tarefa por prioridade. Um estudo aprofundado sobre a utilização de prioridades será apresentado na seção sobre escalonadores de sistemas operacionais, no Capítulo 3, em que se aborda o principal assunto vinculado a prioridades: como distribuí-las entre as tarefas do sistema.

1.4.3 Sistemas multitarefas

Um sistema multitarefas executa a função de multiplexador de um processador, compartilhando a utilização deste entre várias tarefas. Pode-se dizer que um sistema multitarefas opera como um sistema *foreground/background* com múltiplos *backgrounds*. Nesse caso, em vez de existir um grande laço com diversos testes, existe um laço em cada tarefa com uma condição de execução. Essa condi-

ção pode ser um evento assíncrono, tempo ou até mesmo a ordem de inclusão da tarefa no sistema.

Os sistemas multitarefas tendem a maximizar a utilização do processador, além de possibilitar a construção modular de um sistema baseado nas ações que chamamos de tarefas. Um dos aspectos mais importantes da multitarefa é permitir que o desenvolvedor gerencie de uma melhor forma a complexidade inerente às aplicações de tempo real. Sistemas embarcados são normalmente mais fáceis de projetar e manter quando se utiliza um sistema multitarefas, principalmente por permitir um "isolamento virtual" entre as tarefas executadas pelo sistema. Esse isolamento, quando bem projetado, permite um melhor reaproveitamento de código, pois torna os diversos módulos que compõem o sistema independentes. Assim, a adição ou remoção de funcionalidades não influencia a execução de outras tarefas realizadas pelo sistema. Apesar disso, muitos desenvolvedores têm dificuldade na depuração de um sistema multitarefa, por existirem múltiplas sequências de código sendo executadas concorrentemente.

Um sistema com a habilidade de gerir múltiplas tarefas não é considerado um sistema operacional. Para ser considerado um sistema operacional com capacidades mínimas, um sistema deve gerenciar memória, recursos, tempo, entre outros, e, ainda, permitir formas de comunicação e sincronização entre tarefas. Os principais aspectos relativos a essas funcionalidades de um sistema operacional serão apresentados no decorrer dos capítulos seguintes.

1.4.4 Recursos

O gerenciamento de recursos é uma das funções de um sistema operacional. Pode-se dizer que os principais recursos a serem gerenciados são o processador, a memória e os periféricos do sistema. Um recurso é, portanto, qualquer entidade utilizada por uma tarefa. Apesar desse conceito mais abrangente, em que o processador é um recurso a ser gerenciado, a forma de implementação do gerenciamento do processador difere dos outros recursos. Portando, iremos nos referir a recursos no texto de forma genérica como periféricos e memória. Exemplos de recursos são dispositivos de entrada/saída, como teclados, *displays*, conversores A/D etc., ou, ainda, variáveis, estruturas, vetores, matrizes, entre outros.

Os recursos podem ser utilizados por mais de uma tarefa, sendo chamados de recursos compartilhados. Cada tarefa deve obter direito exclusivo de acesso a um

recurso compartilhado para evitar corrupção de dados ou mau funcionamento. Imagine que duas tarefas compartilhem um *display LCD* alfanumérico. Se uma tarefa interromper o processo de escrita no *display* de uma outra tarefa, pode haver duas situações de falha: o protocolo de comunicação com o *display* poderá ser afetado ou a mensagem escrita na tela poderá conter erros, misturando as informações das duas tarefas. O acesso exclusivo de um recurso é conhecido em sistemas operacionais por exclusão mútua, e técnicas para garantir essa condição serão discutidas com mais detalhes no Capítulo 4.

1.4.5 Núcleo (*kernel*)

O núcleo é o módulo central de um sistema operacional e é responsável pelo gerenciamento do processador, da memória, do tempo e dos demais recursos de um sistema computacional. Pode ainda ser visto como o elo entre esses recursos computacionais e as tarefas por ele executadas. Entre as principais funções de um núcleo estão a criação e eliminação de tarefas, a sincronização e comunicação entre essas tarefas e a decisão de qual tarefa deve executar em um determinado momento. O processo de troca da tarefa que está em execução pelo processador também é realizado pelo núcleo.

Em sistemas mais simples, como os baseados em corrotinas, a troca de tarefas ocorre simplesmente pelo retorno da função que implementa a tarefa executada no momento e, logo após, pela chamada da função que contém a próxima tarefa a ser executada. No entanto, na maioria dos sistemas multitarefas a troca da tarefa em execução é realizada por um procedimento conhecido como troca ou chaveamento de contexto. Quando um núcleo multitarefa decide executar uma tarefa diferente, este salva o contexto da tarefa em execução na área de armazenamento de contexto, ou seja, a pilha da tarefa sendo executada no momento. Uma vez que essa operação é realizada, altera-se a pilha em uso para a da nova tarefa e restaura-se o contexto dessa tarefa a partir de sua pilha para o processador. No momento em que o contexto é restaurado, um dos registradores alterados é o contador de programa (*program counter*), fazendo com que o código da nova tarefa passe a ser executado.

Para melhor exemplificar o que acontece durante uma troca de contexto será utilizada uma sequência de figuras da pilha de duas tarefas em determinados momentos desse processo, bem como seus respectivos ponteiros de pilha. O processador utilizado no exemplo é um ARM Cortex-M4. Ainda, para melhor

acompanhar o exemplo, as listagens 1.10 e 1.11 descrevem o código contido em cada tarefa, escrito para o sistema Brazilian Real-time Operating System (BRTOS). Note que ambas as tarefas declaram pelo menos uma variável local e, da mesma forma, ambas desistem do processador por um determinado tempo utilizando uma função de atraso do sistema.

```
1  void task_1(void){
     int milisec = 0;
3    int sec = 0;

5    while(1){
       OSDelayTask(1000);
7      milisec++;
       if (milisec >= 1000){
9        milisec = 0;
         sec++;
11     }
     }
13 }
```

Listagem 1.10 Código-fonte de uma tarefa exemplo escrita para o sistema BRTOS.

```
1  void task_2(void){
     int i = 0;
3
     while(1){
5      i++;
       OSDelayTask(10000);
7    }
   }
```

Listagem 1.11 Código-fonte de uma segunda tarefa exemplo escrita para o sistema BRTOS.

Iniciando a análise da troca de contexto entre essas tarefas, a Figura 1.4 apresenta a pilha da primeira tarefa após sua instalação. Note na figura que a posição do ponteiro de pilha de uma tarefa que acabou de ser instalada aponta para o topo do contexto a ser restaurado.

No caso do processador ARM, o conteúdo do contexto são seus registradores de propósito geral, contador de programa e registrador de condições. Pode-se notar no contexto apresentado, a partir do endereço 0x2001550C, a seguinte ordem de registradores: R4, R5, R6, R7, R8, R9, R10, R11, *backup* de LR, R0, R1, R2, R3, R12, LR, PC, PSR (registrador de condições). Algumas observações são importantes neste ponto:

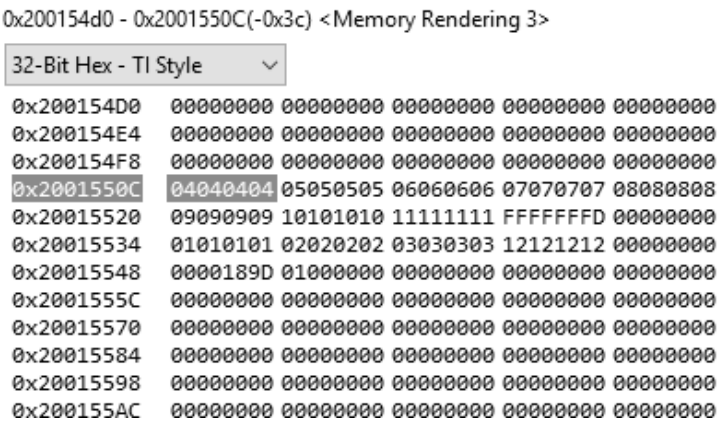


Figura 1.4 Pilha de uma tarefa recém-instalada.

- a ordem de instalação dos registradores nesse exemplo segue o porte[5] oficial do BRTOS para processadores ARM Cortex-Mx;
- o registrador PC contém o endereço do início da tarefa a ser executada, 0x189D no exemplo apresentado;
- o *backup* do registrador LR é necessário para uma tomada de decisão no momento da restauração do contexto;
- os registradores R0, R1, R2, R3, R12, LR, PC e PSR estão no final do contexto porque são salvos nessa ordem pelo *hardware* nos processadores ARM Cortex-Mx.

Quando a tarefa do exemplo assume o processador, o conteúdo da pilha é copiado para seus registradores e o ponteiro de pilha passa a apontar para o início de sua pilha (0x20015550 no exemplo), como apresentado na Figura 1.5. Como o registrador contador de programa recebe o endereço da tarefa que estava armazenado na pilha, essa tarefa passa a ser executada pelo processador.

Logo no início da tarefa as suas variáveis locais são armazenadas na pilha, assim como alguns registradores são alocados para serem utilizados como ponteiros e/ou variáveis auxiliares no processo de leitura e escrita da memória. A Figura 1.6 demonstra o código C/*assembly* desse processo (Figura 1.6 (a)), bem como a pilha e seu ponteiro após a alocação das variáveis (Figura 1.6 (b)). Note

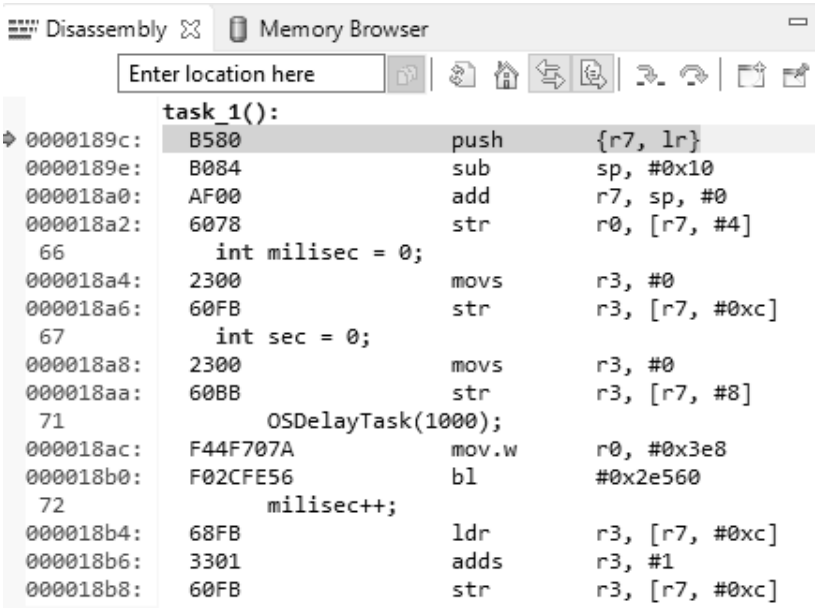
[5] Qualquer RTOS preemptivo possui duas partes, uma independente e outra dependente do processador utilizado. O porte é o código dependente do processador, que deve ser implementado sempre que se quiser prover suporte de um determinado RTOS a uma nova plataforma.

Figura 1.5 Pilha de uma tarefa sendo executada pela primeira vez.

nesse exemplo que foram ocupados 24 *bytes* para gerenciar as variáveis locais, deslocando o ponteiro da pilha para a posição 0x20015538. Os 24 *bytes* foram ocupados no armazenamento dos registradores R0, R7, LR, além das variáveis *milisec* e *sec* do exemplo. Perceba ainda que para acessar as variáveis *milisec* e *sec*, o código utiliza a posição do ponteiro de pilha somado com 0x0C e 0x08, respectivamente.

Ao executar a função ***OSDelayTask()***, a pilha da tarefa é utilizada em mais 32 *bytes* para armazenar as variáveis locais da função, sendo deslocada para a posição 0x20015518, como demonstrado na Figura 1.7.

Como a função ***OSDelayTask()*** é uma desistência temporária do processador, ocorrerá uma troca de contexto. Os registradores do processador nesse momento são então armazenados na pilha da tarefa em execução, para possibilitar a entrega do processador a outra tarefa, deslocando o ponteiro da pilha da tarefa para a posição 0x200154D4. O deslocamento é proporcional aos 68 *bytes* do contexto completo do processador, conforme descrito anteriormente. Nota-se claramente no exemplo que os registradores R4, R5, R6, R8, R9, R10 e R11 não foram utilizados pela tarefa (mantiveram o valor padrão na instalação da tarefa), como pode ser verificado na Figura 1.8. Essa posição do ponteiro de pilha será armazenada no bloco de controle da tarefa, de forma a permitir que na próxima ativação dessa tarefa seu contexto possa ser restaurado. Verifique ainda que o endereço de retorno para essa tarefa está armazenado no contexto como o conteúdo a ser restaurado para o registrador contador de programa, no endereço 0x2E636, demonstrando que a tarefa irá retornar em uma posição diferente de seu início.



(a) Código em linguagem *assembly*.

(b) Mapa de memória.

Figura 1.6 Alocação das variáveis locais da tarefa executando pela primeira vez.

Como o ponteiro de pilha da tarefa que está deixando de ser executada está salvo, o sistema operacional pode buscar pela próxima tarefa a ser executada. Assim que essa tarefa é escolhida, seu bloco de controle de tarefas é acessado para copiar a última posição conhecida de seu ponteiro de pilha. Dessa forma, o ponteiro de pilha do processador recebe o ponteiro de pilha da próxima tarefa, que aponta para a posição 0x2001568C no exemplo apresentado na Figura 1.9.

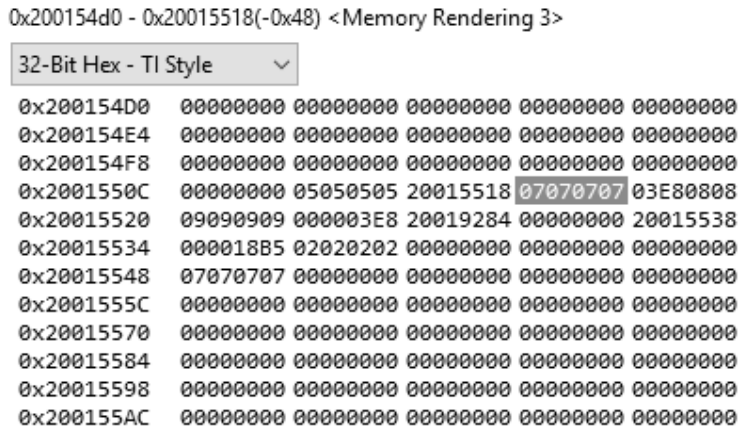


Figura 1.7 Pilha da tarefa após entrar na função ***OSDelayTask()***.

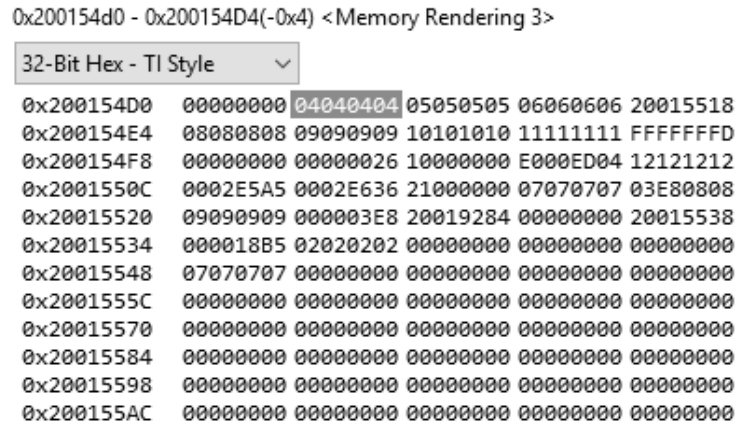


Figura 1.8 Pilha da tarefa com contexto salvo, ou seja, preparada para ceder o processador a outra tarefa.

Note que a penúltima posição do contexto (endereço 0x200156C8) contém o valor a ser copiado para o contador de programas. Esse valor é diferente do contido na tarefa anterior. O endereço inicial da tarefa anterior era 0x189D, enquanto o endereço contido nessa nova pilha é 0x18D1. Portanto, assim que esse contexto for restaurado, o contador de programa será modificado para esse valor e o processador passará a executar a nova tarefa.

Ainda nesse exemplo, quando a nova tarefa for despachada para o processador, o ponteiro da pilha irá apontar para a posição de memória 0x200156D0. Como existem variáveis locais a serem alocadas e registradores a serem salvos, o ponteiro será deslocado para a posição 0x200156B8, como pode ser notado na

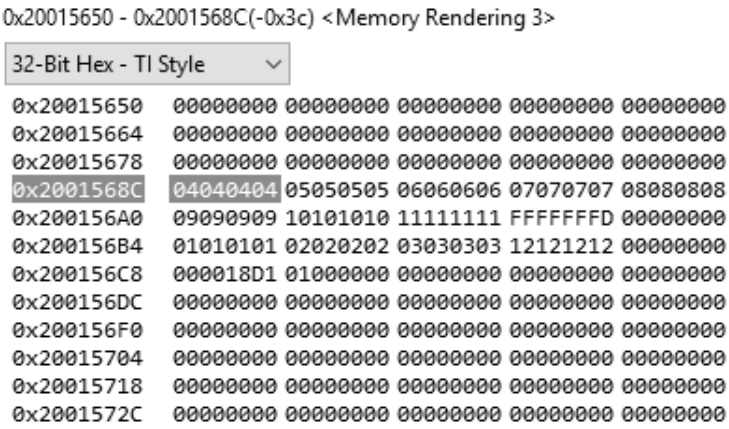


Figura 1.9 Pilha da segunda tarefa com ponteiro de pilha na posição para despachar a tarefa para o processador.

Figura 1.10. Note que a variável i declarada no início da Listagem 1.11 foi alocada na posição de memória 0x200156C4 e já foi incrementada uma primeira vez, pois a figura apresenta um extrato da pilha dessa tarefa logo após o incremento da variável i. Esse processo de troca de contexto acontece constantemente enquanto um sistema multitarefa executa, pois esse procedimento é o que permite que várias tarefas diferentes possam executar concorrentemente.

Figura 1.10 Pilha da segunda tarefa logo após a alocação das variáveis locais.

Como se pode ver nesse exemplo, o uso de um núcleo de tempo real permite que a aplicação seja dividida em múltiplas tarefas. Normalmente essa divisão irá simplificar o projeto de sistemas embarcados, pois possibilita isolar parcialmente as diversas tarefas que irão compor o sistema como um todo, evitando assim

interferências no comportamento de uma tarefa pelas outras. No entanto, além de exigir um projeto adequado na distribuição de tarefas, o uso do núcleo adiciona sobrecarga (*overhead*) ao sistema devido a vários motivos. Entre eles estão:

- o espaço de código extra ao sistema para implementar os serviços providos pelo núcleo;
- a memória RAM para as estruturas de dados;
- cada tarefa possui a própria pilha. Devido a isso, existe uma tendência de maior ocupação de memória RAM conforme cresce o número de tarefas instaladas;
- quanto maior for a quantidade de registradores do processador, maior será o *overhead*, pois o tempo requerido para realizar o chaveamento de contexto depende de quantos registradores devem ser salvos e restaurados pelo processador. Devido a isso, o desempenho de um sistema operacional não deve ser simplesmente julgado por quantos chaveamentos de contexto o núcleo é capaz de realizar por segundo (a menos que um determinado processador seja escolhido para realizar tal análise);
- o núcleo consome tempo de processamento. A utilização do núcleo é extremamente dependente da implementação e do número de tarefas e serviços do sistema utilizados na concepção do sistema embarcado.

Devido a essa sobrecarga, microcontroladores de baixo custo foram por muito tempo inadequados para executar um núcleo de tempo real, sendo um dos principais fatores para a pouca quantidade de memória RAM disponível. Até pouco tempo atrás era comum encontrar microcontroladores com memória RAM entre 128 e 512 *bytes*. No entanto, atualmente os microcontroladores possuem memória suficiente para executar um número razoável de tarefas em um sistema operacional, pois é comum encontrar modelos variando de poucos quilobaites até centenas de quilobaites de memória RAM.

O núcleo de um sistema operacional de tempo real pode ser dividido em três componentes básicos. São eles:

- escalonador (*scheduler*): implementa um ou mais algoritmos destinados a determinar quando e qual tarefa deve ser executada;

- objetos: são estruturas de dados especiais do núcleo, destinadas a possibilitar a sincronização, troca de dados, exclusão mútua etc., entre tarefas sendo executadas concorrentemente em um sistema operacional de tempo real. Exemplos de objetos são semáforos, filas, *mutexes* etc.;
- serviços: são as funções que definem as possíveis operações a serem realizadas com um objeto do sistema operacional, além de permitirem o gerenciamento de tempo e outros recursos do sistema em utilização. Pode-se dizer que os serviços de um RTOS são a sua *application programming interface* (API).

A partir do uso de um núcleo multitarefa e seus objetos e serviços, pode-se atingir um melhor aproveitamento do processador, pois tais ferramentas possibilitam uma melhor organização na execução das tarefas a que o sistema se destina. Um código bem estruturado evita desperdício de processamento e possibilita um melhor gerenciamento dos recursos existentes. É importante ressaltar que utilizar de forma adequada os objetos e serviços do sistema operacional é fundamental para se obter os possíveis benefícios de um sistema multitarefa, o que demanda experiência por parte do desenvolvedor. Maiores detalhes sobre esses serviços serão apresentados no Capítulo 4, que aborda os objetos providos por sistemas operacionais.

1.4.6 Reentrância e funções seguras

Reentrância refere-se à capacidade de uma função ser executada concorrentemente de forma segura. Assim, mais de um trecho de código concorrente pode utilizar uma função reentrante sem a possibilidade de danificar os dados por ela processados. Esse tipo de função pode ser interrompida em qualquer momento e resumida mais tarde sem que ocorram perdas de dados. Para que isso seja possível as funções reentrantes devem:

- utilizar somente variáveis locais (implementadas em registradores da CPU ou alocadas na pilha);
- invocar somente funções que também sejam reentrantes.

Um exemplo de função reentrante é apresentado na Listagem 1.12. Essa função é parte da biblioteca padrão da linguagem C e tem como objetivo copiar

dados de uma posição de memória para outra. Como os argumentos da função ***strcpy()*** são alocados na pilha da tarefa que invocou a função, ***strcpy()*** pode ser chamada por diversas tarefas sem que os dados de cada tarefa sejam danificados. Note, no entanto, que se o destino da cópia de memória em execuções concorrentes da função for o mesmo, ocorrerá corrupção de dados. Essa falha não se deverá à implementação da função, mas sim a um erro de projeto do sistema em desenvolvimento.

```
void strcpy(char *dest, char *src){
   while(*src){
      *dest++ = *src++;
   }
}
```

Listagem 1.12 Exemplo de função reentrante.

O termo reentrância precede o conceito de tarefas/*threads*, sendo comumente utilizado no contexto de sistemas baseados no superlaço. Em um sistema *foreground/background*, uma função reentrante pode ser utilizada tanto no *background* quanto no *foreground*, ou seja, no laço principal e nas rotinas de tratamento de interrupção. Com o surgimento dos sistemas multitarefa surgiu uma nova categoria de função, a das funções seguras (*thread safe*). Apesar de serem muito semelhantes, pequenas diferenças separam as duas categorias de funções.

Como previamente discutido, uma função reentrante pode ser executada, interrompida e voltar à execução. Ademais, pode ser executada concorrentemente por múltiplas tarefas, desde que em cada chamada os dados/parâmetros/entradas providos sejam únicos. Já uma função segura pode ser executada simultaneamente por múltiplas tarefas, mesmo que cada chamada faça referência ou forneça os mesmos dados ou entradas, pois o acesso a estes será realizado sequencialmente.

Um exemplo de função não reentrante é apresentado na Listagem 1.13. A função ***swap()*** tem como objetivo trocar o conteúdo de seus dois argumentos. Iremos assumir que estamos utilizando um sistema operacional que permita a interrupção da execução de uma tarefa em detrimento de outra tarefa de maior prioridade. Vamos assumir, ainda, que algumas interrupções estão ativas e que a variável **teste** está declarada no escopo global.

A Figura 1.11 demonstra o que pode acontecer se uma tarefa com baixa prioridade for interrompida durante a execução da função ***swap()***. Note que quando a tarefa de baixa prioridade é interrompida, a variável **teste**, que tem escopo

```
1 int teste;
  void swap(int *x, int *y){
3    teste = *x;
     *x = *y;
5    *y = teste;
  }
```

Listagem 1.13 Exemplo de função não reentrante.

global, possui o valor 5. Nesse exemplo, a rotina de tratamento da interrupção retira uma tarefa de maior prioridade do estado de espera. Nesse momento, o sistema operacional verifica que essa tarefa é a de maior prioridade pronta para executar. Durante a execução da tarefa de maior prioridade, a variável **teste** tem seu conteúdo alterado, passando para zero. Quando a tarefa de maior prioridade conclui seus procedimentos, a tarefa de menor prioridade recebe novamente o processador. Entretanto, a variável **teste**, que deveria possuir o valor de 5, agora contém o valor 0. Nesse momento podemos notar que os dados da função de menor prioridade foram corrompidos.

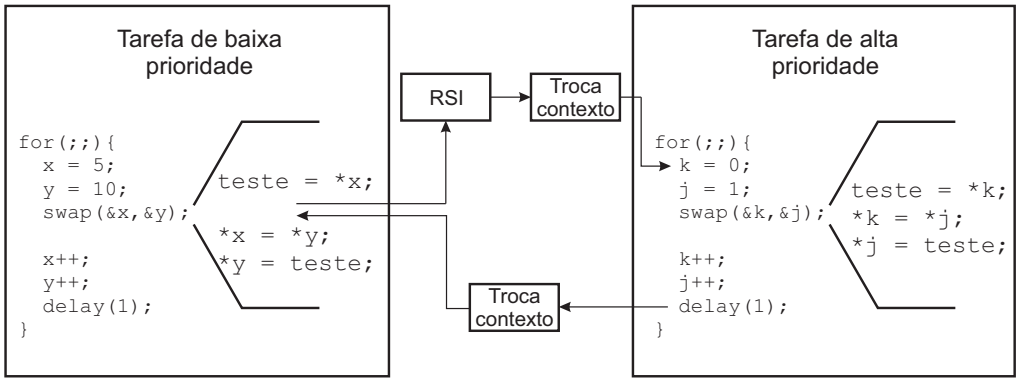


Figura 1.11 Estados das tarefas utilizando funções não reentrantes.

Embora a função ***swap()*** não seja reentrante, as modificações necessárias para torná-la reentrante são relativamente simples. Basta utilizar uma das seguintes técnicas: declarar **teste** como variável local da função ***swap()*** ou desabilitar as interrupções no início e habilitá-las ao final da função. Em outros casos essas modificações podem tornar-se mais complicadas. Sempre que utilizamos sistemas multitarefa devemos ser bastante cautelosos ao utilizar uma função não reentrante, pois as inconsistências causadas por esse tipo de função podem não ser aparentes em um primeiro momento.

Um exemplo clássico de função que não era reentrante e passou a ser suportada por sistemas multitarefa é a função ***strtok()***, contida na biblioteca padrão da linguagem C. Uma sequência de chamadas dessa função quebra uma *string* em várias *substrings*, em que a *string* é uma sequência de caracteres contíguos separados por qualquer caractere que faça parte do delimitador. Na primeira chamada a função deve receber um ponteiro de caracteres como argumento, sendo que o primeiro caractere é utilizado como local inicial para procurar as *substrings*. Em chamadas subsequentes, a função espera um ponteiro nulo e usa a posição logo após o fim da última *substring* como o novo local inicial. A Listagem 1.14 apresenta um exemplo de uso dessa função. Já a Listagem 1.15 demonstra o resultado de sua execução, em que podemos visualizar a saída de cada chamada da função ***printf()***.

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

char str[] = "1,22,333,4444,55555";

// Retorna a primeira substring
char *token = strtok(str, ",");

/* Imprime cada uma das substring delimitadas por uma vírgula */
while (token != NULL){
  printf("%s\n", token);
  token = strtok(NULL, ",");
}
```

Listagem 1.14 Exemplo de uso da função *strtok()*.

```
Substring:1
Substring:22
Substring:333
Substring:4444
Substring:55555
```

Listagem 1.15 Resultado das execuções sequenciais da função *strtok()*.

Note que se essa função for executada por uma tarefa em um sistema multitarefa e houver uma interrupção que cause uma troca de contexto, uma segunda tarefa que a executasse poderia alterar a posição da *string* sendo utilizada e, consequentemente, ocasionar uma falha na próxima chamada da função quando a tarefa interrompida voltasse a executar. Isso deve-se à forma de implementação da função, que pode ser visualizada na Listagem 1.16. Verifique que a posição do final da última *substring* processada é armazenada em uma variável local estática (*static char *lasts*), que nesse caso irá operar como uma variável global. Se

o conteúdo dessa variável for alterado em uma chamada concorrente da função, os dados anteriores serão perdidos.

```
char * strtok(char *s, const char *delim){
  static char *lasts;
  register int ch;

  if (s == 0)
  s = lasts;

  do {
    if ((ch = *s++) == '\0')
    return 0;
  } while (strchr(delim, ch));

  --s;
  lasts = s + strcspn(s, delim);
  if (*lasts != 0)
  *lasts++ = 0;
  return s;
}
```

Listagem 1.16 Código-fonte da função *strtok()*.

Para possibilitar que essa função se torne reentrante um terceiro parâmetro foi adicionado, como pode ser observado na Listagem 1.17, que apresenta a versão reentrante da função ***strtok()***. Note que em vez de utilizar uma variável local estática, essa implementação utiliza um terceiro parâmetro que é fornecido pela tarefa que irá executar a função, mantendo localmente a posição do final da última *substring* processada. A Listagem 1.18 apresenta um exemplo de uso da função ***strtok_r()***. Note que o próprio uso da função é simplificado, pois não é necessário diferenciar a primeira chamada da função das chamadas subsequentes com um ponteiro nulo.

```
char *strtok_r (char *s, const char *delim, char **save_ptr){
  char *token;

  if (s == NULL)
    s = *save_ptr;

  /* Desloca string a partir do delimitador. */
  s += strspn (s, delim);

  if (*s == '\0'){
    *save_ptr = s;
    return NULL;
  }

  /* Encontra o final da substring pelo delimitador. */
  token = s;
  s = strpbrk (token, delim);

  if (s == NULL)
    /* Essa substring termina a string.  */
    *save_ptr = __rawmemchr (token, '\0');
```

```
22    else{
        /* Delimita a substring com caracter \0 e aponta para
24      o início da próxima substring em *SAVE_PTR. */
        *s = '\0';
26      *save_ptr = s + 1;
      }
28    return token;
```

Listagem 1.17 Código-fonte da função *strtok_r()*.

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

4 char str[] = "1,22,333,4444,55555";
  char *substr;
6 char *rest = str;

8 while((substr = strtok_r(rest, ",", &rest))){
    printf("Substring:%s\n", substr);
10 }
```

Listagem 1.18 Exemplo de uso da função *strtok_r()*.

No entanto, em um sistema multitarefa podem existir situações em que não há maneiras de tornar a função reentrante. Nesses casos a única solução é garantir a execução sequencial da função por múltiplas tarefas concorrentes, tornando a função segura, a partir de um mecanismo de exclusão mútua. Um caso clássico em sistemas embarcados são as funções ***malloc()*** e ***free()*** para alocação dinâmica de memória. A função ***malloc()*** fragmenta um grande bloco de memória no tamanho solicitado pelo usuário e retorna um ponteiro para o início da memória solicitada. Se essa função for invocada por duas tarefas ao mesmo tempo, pode retornar o mesmo segmento de memória para ambas as tarefas. Para isso a segunda chamada da função ***malloc()*** deve ocorrer depois da seleção do segmento de memória pela primeira instância da função, mas antes desse segmento de memória ser marcado como ocupado. Nesse caso é impossível tornar a função segura por reentrância com variáveis locais. Portanto, uma possível solução é utilizar o bloqueio das interrupções entre a seleção e a marcação do segmento de memória como ocupado.

1.4.7 Variáveis globais voláteis e estáticas

Ao utilizar um sistema operacional, alguns cuidados adicionais devem ser tomados no uso de variáveis globais. Esses cuidados estão relacionados a como essas variáveis devem ser declaradas e, principalmente, vinculados ao uso da variável em questão. Para fazer essa análise precisamos entender o conceito de

qualificador de variável. Ao declarar uma variável, seu comportamento será determinado pelo qualificador utilizado. Os principais qualificadores em linguagem C são: *static*, *volatile*, *const*, *extern* e *register*. A declaração de uma variável deve seguir o formato apresentado na Listagem 1.19.

```
[<qualificador>] <tipo> <nome>;

Ex.: static int x;
```

Listagem 1.19 Formato para declaração de variáveis com qualificador.

O qualificador *volatile* é um dos mais importantes quando declaramos variáveis globais em um sistema multitarefa. Esse qualificador também é muito importante quando utiliza-se variáveis globais em interrupções, pois as rotinas de tratamento de interrupções também concorrem com o código principal em um sistema embarcado, seja ele concebido com sistema operacional ou não. A principal ação desse qualificador é informar ao compilador que, sempre que essa variável for utilizada, deve-se obrigatoriamente ler seu valor da memória RAM e armazená-lo em um registrador. Para entender a importância desse comportamento, analise o código apresentado na Listagem 1.20.

```
volatile int i = 0;
void main(void){
  // Espere por 100 ms
  while(i < 100);
  ...
}

// Interrupção de tempo a cada 1 ms
void TimerInt(void){
  i++;
}
```

Listagem 1.20 Exemplo de uso do qualificador *volatile* para variáveis em sistemas multitarefa ou com interrupções.

Note que nesse código há um simples teste para verificar quando a variável i extrapola o valor 100, que no caso significa 100 ms. Se o qualificador *volatile* não for utilizado, provavelmente o código será compilado de forma que a variável i esteja sendo testada em um registrador, ou seja, o valor atual da variável é carregado em um registrador e testado continuamente. Assim, mesmo que a variável esteja sendo incrementada na interrupção e seu valor na memória RAM esteja correto, o valor do registrador não será alterado e, portanto, o código nunca passará do laço *while* apresentado. Ainda, se otimizações do compilador forem utilizadas, pode ocorrer de o código ser descartado, pois o teste não fará sentido.

Ao utilizar o qualificar *volatile*, o compilador é informado para sempre fazer a leitura do valor contido em i da memória RAM para um registrador e somente depois realizar o teste para verificar se o valor ainda é inferior ao número 100.

Agora imagine um sistema multitarefa. Esse tipo de situação pode ser a mais comum possível, pois se duas tarefas estiverem utilizando uma mesma variável global, pode ocorrer da alteração dessa variável em uma tarefa não ser percebida na outra tarefa. Tal situação acontecerá se o valor contido na variável for carregado em um registrador em uma dada tarefa e esse registrador estiver sendo utilizado em um teste como o apresentado no laço *while* do exemplo. Portanto, sempre que uma variável global estiver sendo compartilhada entre duas ou mais tarefas, ou, ainda, entre uma tarefa e uma interrupção, o qualificador *volatile* é obrigatório para a declaração dessa variável. O principal exemplo do uso desse qualificador é na declaração de registradores. Note que como o valor armazenado em um registrador pode ser alterado pelo *hardware*, a utilização desse qualificador em sua declaração é mandatória.

Já o qualificador *static* é muito importante quando temos interesse no encapsulamento de uma variável. Imagine que se pretende criar o código de um *driver* e que nesse *driver* existem certas variáveis globais, principalmente por terem seu uso vinculado a uma interrupção. Essas variáveis são globais para poderem ser acessadas tanto pela interrupção quanto pelo código do *driver*. No entanto, como essas variáveis são globais, nada impede que um código de usuário altere o valor de uma dessas variáveis, corrompendo a estrutura de dados do *driver*.

Seguindo uma boa prática de organização de código, esse *driver* será escrito em um arquivo .c e .h à parte. Ao utilizar as variáveis globais como estáticas, o seu acesso fora do arquivo do *driver* será impedido pelo compilador. Caso seja necessário ler seu conteúdo, uma função pode ser disponibilizada pelo *driver* para acessá-lo, como apresentado na Listagem 1.21. Note que nesse exemplo a única forma de acessar a variável i é por meio da função ***GetCounter()***.

```
1  /* main.c  */
   #include "timer.h"
3
   void main(void){
5
     // Espere por 100 ms
7    while(GetCounter() < 100);

9    ...

11 }
```

```
/* timer.c  */
static int i = 0;

// Função para acessar o valor do contador
int GetCounter(void){
  return i;
}

// Interrupção de tempo a cada 1 ms
void TimerInt(void){
  i++;
}

/* timer.h  */
// Declaração do protótipo das funções pública do driver
int GetCounter(void);
```

Listagem 1.21 Uso do qualificador *static* para delimitar o acesso a uma variável global.

O qualificador *static* pode ainda ser utilizado em variáveis locais, quando se deseja manter o valor da variável entre chamadas da função. A Listagem 1.22 apresenta o uso desse qualificador. Note que a função ***contar()*** retorna o número de vezes que a função foi executada. A variável local estática é inicializada, mas o valor atribuído a essa variável só é utilizado na primeira vez que a função for chamada. Nas próximas execuções o valor anterior será mantido. Podemos interpretar uma variável local estática como uma variável global encapsulada em uma função.

```
int contar(void){
  static int cnt = 0;
  cnt++;
  return cnt;
}
```

Listagem 1.22 Exemplo de uso de uma variável local estática.

Os outros qualificadores também têm sua importância. O qualificador *const* é bastante utilizado em sistemas embarcados para forçar o armazenamento de uma dada variável na memória *flash* quando esta for constante. Como sistemas embarcados geralmente possuem memórias de programa (geralmente *flash*) muito maiores do que a memória de dados, o armazenamento de variáveis constantes, por exemplo tabelas, na memória de programa é bastante utilizada.

Outro uso comum do qualificador *const* é evitar que o parâmetro de uma função seja alterado pela própria função. Por exemplo, para proteger um ponteiro, basta declará-lo como constante. Isso impedirá que o conteúdo apontado pelo ponteiro possa ser modificado. Assim, o ponteiro poderá somente ser utilizado para ler o conteúdo apontado, bem como o endereço apontado poderá

ser incrementado ou decrementado. Um exemplo clássico da utilização desse qualificador se dá na função ***memcpy()*** da biblioteca padrão da linguagem C. Note na Listagem 1.23 que o ponteiro da fonte dos dados a serem copiados é declarado como constante, para evitar que os dados copiados não sejam alterados pela função. O ponteiro para o destino dos dados não recebe o qualificador, pois os dados apontados por esse ponteiro serão alterados na função.

```
1 void *memcpy(void *dst, const void *src, size_t len){
    size_t i;
3   char *d = (char *)dst;
    const char *s = (const char *)src;
5
    for (i=0; i<len; i++) {
7   d[i] = s[i];
  }
```

Listagem 1.23 Função *memcpy()* da biblioteca padrão da linguagem C utilizando o qualificador *const*.

O qualificador *extern* informa ao compilador que a variável é global, mas está declarada em outro arquivo .c que não o atualmente em uso. Já o qualificador *register* força o armazenamento de uma dada variável em um registrador de propósito geral do processador. Note que o qualificador *register* só é aceito em variáveis locais, pois alocar uma variável global em um registrador iria impossibilitar o uso desse registrador em outros trechos de código.

Como se pode perceber nos vários exemplos apresentados, a utilização de qualificadores é de extrema importância quando escrevemos códigos para sistemas embarcados, principalmente por esse tipo de código estar mais próximo ao *hardware* utilizado.

1.5 Resumo

Tradicionalmente, o projeto de sistemas embarcados foi baseado na técnica de superlaço. Nessa técnica, o processador executa um fluxo de código principal, denominado superlaço (ou *background*). Este pode ser momentaneamente interrompido pela ativação, periódica ou aperiódica, de rotinas de tratamento de interrupções (*foreground*); as quais, por sua vez, sinalizam, utilizando *flags*, a ocorrência de eventos que demandam processamento posterior a ser realizado no superlaço.

Recentemente, com a evolução dos microcontroladores, ocorreu o aumento da quantidade de memória e capacidade processamento, bem como do número

e da diversidade de periféricos disponíveis a um custo cada vez menor. Isso, por sua vez, tem ocasionado um aumento significativo das funcionalidades dos sistemas embarcados e da complexidade das tarefas realizadas por estes. E, como consequência do aumento de complexidade aliado ao aumento da disponibilidade de recursos (processamento, memória, periféricos), a utilização de um sistema operacional de tempo real (RTOS) no projeto de sistemas embarcados tem aumentado significativamente.

Dentre as vantagens de utilização de um RTOS, em comparação com a técnica de superlaço, pode-se citar uma maior modularização do projeto característica de sistemas multitarefas, aliada a uma melhor reutilização de *software*, bem como as facilidades providas para cumprimento das restrições temporais geralmente impostas aos sistemas embarcados de tempo real. Por outro lado, o projeto satisfatório do sistema embarcado não é atingido tão somente pelo uso do RTOS, requerendo do(s) projetista(s) um conhecimento mais profundo para, com o uso adequado do RTOS, atingir os requisitos especificados para o sistema embarcado. Portanto, o domínio de conhecimentos relacionados aos RTOS, como o escalonamento de tarefas, a definição das prioridades, o gerenciamento do tempo, a utilização correta de funções reentrantes e dos objetos fornecidos para o compartilhamento de recursos e comunicação entre tarefas, é de fundamental importância para o(s) projetista(s) de sistemas embarcados modernos.

1.6 Problemas

Para realizar os exercícios, baixe e instale o ambiente de desenvolvimento MingW, disponível em www.mingw.org, e a interface para linguagem C Eclipse CDT, disponível em eclipse.org/cdt. Então, faça o *download* do projeto do BRTOS pelo *link* github.com/brtos/simulador/archive/master.zip. Descompacte o arquivo *zip* e importe o projeto no Eclipse.

Problema 1.1. Crie duas tarefas que utilizam a função ***OSDelayTask()*** e as execute no simulador do BRTOS.

Problema 1.2. Faça uma função não reentrante de tal forma que seja possível detectar quando houve corrupção dos dados. Então, modifique as tarefas que você criou no exercício anterior para que elas utilizem a função não reentrante recém-criada e determine o instante em que ocorre a corrupção dos dados.

Problema 1.3. Apresente duas formas de tornar a função não reentrante criada no exercício anterior em uma função reentrante e analise os impactos de cada forma com relação ao tempo de execução e consumo de memória.

Problema 1.4. Descreva o que é a pilha de um processador e qual sua utilidade para a concepção de um sistema que possui múltiplos fluxos de execução. Note que em um sistema tradicional, baseado em superlaço, existem múltiplos fluxos de execução concorrentes devido às rotinas de tratamento de interrupções.

Problema 1.5. Explique com suas palavras a importância do qualificador *volatile* para a declaração de variáveis globais compartilhadas entre rotinas de tratamento de interrupções e o código principal de usuário.

Este livro tem como objetivo servir de referência técnica e didática na área de sistemas operacionais de tempo real (RTOS) e na sua utilização nos projetos de sistemas embarcados.

É voltado ao ensino de sistemas embarcados nos cursos de Engenharia Elétrica, Eletrônica, Computação, Controle e Automação, Telecomunicações, cursos técnicos, bem como para profissionais que atuam na área de sistemas embarcados. Além disso, visa preencher uma lacuna identificada pelos autores na bibliografia referente ao tema, refletida na pouca oferta de livros didáticos na área de sistemas embarcados e, especialmente, com relação aos sistemas operacionais de tempo real.

A familiaridade do leitor com a linguagem utilizada nos exemplos facilita o aprendizado, bem como o aproxima das implementações reais. Ainda, o uso massivo de exemplos faz o livro ter um propósito que vai além da leitura, pois o torna uma referência prática de consulta no dia a dia dos profissionais da área e dos estudantes na elaboração de projetos realizados durante as disciplinas da graduação, em atividades extraclasse, em grupos de pesquisa e em projetos de conclusão de curso.

www.blucher.com.br

ISBN 978-85-212-1396-3

9 788521 213963

Blucher

Clique aqui e:

VEJA NA LOJA

Sistemas Operacionais de Tempo Real e Sua Aplicação em Sistemas Embarcados

---

Gustavo Weber Denardin , Carlos Henrique Barriquello

ISBN: 9788521213963

Páginas: 474

Formato: 17 x 24 cm cm

Ano de Publicação: 2019

Peso: 0.775 kg
