Stephenson de S. L. Galvão

Modelagem Formal do Sistema Operacional de Tempo Real FreeRTOS Utilizando o Método B

Natal - RN, Brasil

1 de junho de 2009

Stephenson de S. L. Galvão

Modelagem Formal do Sistema Operacional de Tempo Real FreeRTOS Utilizando o Método B

Qualificação de mestrado apresentada ao programa de Pós-graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte, como requisito parcial para a obtenção do grau de Mestre em Ciências da Computação.

Orientador:
Prof. Dr. David Déharbe

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Natal - RN, Brasil

1 de junho de 2009

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A área da computação é uma área relativamente jovem que ainda necessita de esforços para torna-se matura e confiável como as demais áreas da ciência. Assim para um melhor planejamento da evolução da computação e para agrupar esforços em objetivos comuns, vários pesquisadores procuraram identificar alguns dos principais desafios que a sociedade gostaria de ver desenvolvidos a curto e médio prazo.

Um pesquisador bastante conhecido pela computação, mais especificamente na disciplina de métodos formais, responsável por definir necessidades através de desafios, é o britânico Jim Woodcock. No âmbito nacional existe a Sociedade Brasileira de Computação (SBC) que também está preocupada em definir diretrizes para a área em forma de desafio. Assim, um desafio comum entre as duas partes (SBC e Jim), lançado recentemente por Jim, é o de especificar formalmente o Sistema Operacional de Tempo Real FreeRTOS.

Sistemas de tempo real são sistema que necessitam responder, em tempos bem definidos, aos estímulos causados por eventos externos. Esse tipo de sistema está presente em quase todo o cotidiano da sociedade atual e tem sido utilizado principalmente em ambientes críticos como o controle de transportes (aeronaves, trens e automóveis), monitoramento de aparelhos hospitalares e controle da linha de produção de fábricas.

O FreeRTOS é um software que tem o objetivo de facilitar o desenvolvimento dos sistemas de tempo real. Ele, através de funcionalidades disponibilizadas na forma de bibliotecas, abstrai do desenvolvedor os detalhes do hardware no qual o sistema será utilizado. Uma das principais característica desse software está na sua simplicidade e portabilidade, pois o seu código, feito em *C* com partes em assembler, possui cerca 2.200 linhas de código e é oficialmente portável para dezessete arquiteturas diferentes de mono-processadores.

A especificação formal de um sistema é uma metodologia de especificação, construção e verificação de sistemas capaz de deixa-los mais confiável e fidedigno. Isso é feito utilizando-se conceitos matemáticos sólidos como a teoria do conjunto e a lógica de primeira ordem. Com isso, uma especificação formal do FreeRTOS tornaria mais fidedigno não só próprio sistema

operacional mas também os sistemas que utilizam-se dele para realizar suas tarefas.

Nesse contexto o presente trabalho pretende, através de uma especificação formal do FreeRTOS, resolver os desafios proposto por Jim e pela SBC. O formalismo utilizado na resolução desse problema será o método B, que trata-se de uma metodologia de construção de sistemas por técnicas de refinamentos. Nessa técnica um sistema concreto é gerado a partir de uma especificação abstrata do mesmo e cada etapa de abstração do desenvolvimento é passível de uma verificação estática para garantir a corretude do sistema.

O plano desenvolvido para a resolução desse desafio foi divido em três partes principais. Inicialmente tem-se o estudo do FreeRTOS para descobrir suas características, estruturas e comportamentos. Depois disso, uma especificação abstrata do mesmo é desenvolvida com a finalidade de especificar os seus comportamentos e estruturas mais relevantes. Em seguida, na terceira parte, essa especificação abstrata é refinada de acordo com decisões de projetos para que determinadas estruturas e funcionalidades do sistema torne-se mais concreta e passível de verificação.

Atualmente a primeira e a segunda parte do trabalho estão bem desenvolvidos. Como fruto desse trabalho inicial tem-se uma especificação abstrata de algumas das principais funcionalidades do sistema restando apenas a especificação das demais funcionalidades e o refinamento em um nível concreto da especificação do sistema.

1.1 Objetivos

Falar do objetivo da dissertação e não só da qualificação. Itens a serem discutidos:

- Abrangência da especificação
- Profundidade em aspectos pelo menos da construção do software
- Se necessário tem a possibilidade de estessão até o nível de assemblagem devido aos códigos em assembler que compõem o FreeRTOS.

1.2 Metodologia

Metodologia da dissertação, no contexto do que já foi feito

2 *FreeRTOS*

O FreeRTOS é um sistema operacional de tempo real (SOTR) enxuto, simples e de fácil uso. O seu código fonte, feito em *C* com partes em *assembly*, é aberto e possui pouco mais de 2.200 linhas de código, que são essencialmente distribuídas em quatro arquivos: `task.c`, `queue.c`, `croutine.c` e `list.c`. Uma outra característica marcante desse sistema está na sua portabilidade, sendo o mesmo oficialmente disponível para 17 arquiteturas mono-processadores diferentes, entre elas a PIC, ARM e Zilog Z80, as quais são amplamente difundidas em produtos comerciais através de sistemas computacionais embutidos.

Assim como a maioria dos sistemas operacionais, o FreeRTOS provê para os desenvolvedores de sistemas acesso aos recursos de *hardware*, facilitando com isso o desenvolvimento de sistemas de tempo real. Assim, FreeRTOS trabalha como na figura 2.1, fornecendo uma camada de abstração localizada entre a aplicação e o hardware, que tem o papel esconder dos desenvolvedores de aplicações detalhes do hardware que será utilizado.

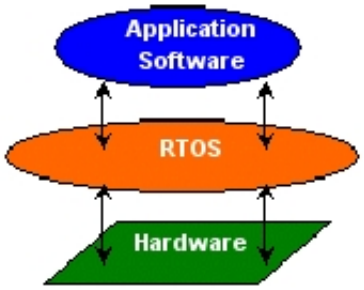


Figura 2.1: Camada abstrata proporcionada pelo FreeRTOS [David Kalinsky 2003]

Para prover tal abstração o FreeRTOS é composto por um conjunto de bibliotecas de tipos e funções que devem ser linkeditadas[1] com o código da aplicação a ser desenvolvida. Juntas, essas bibliotecas fornecem para o desenvolvedor serviços como gerenciamento de tarefa, comunicação e sincronização entre tarefas, gerenciamento de memória e controle dos dispositivos de entrada e saída. A seguir tem-se em detalhe os principais serviços providos pelo o FreeRTOS junto com a biblioteca que os disponibilizam.

[1] Processo que liga o código da aplicação ao código das funcionalidades de outras bibliotecas utilizada por ela.

2.1 Gerenciamento de Tarefas e Co-Rotinas

2.1.1 Tarefa

Para entender como funciona o gerenciamento de tarefas do FreeRTOS é necessário primeiramente entender-se o conceito de tarefa. Uma tarefa é uma unidade básica de execução, com contexto próprio, que compõem as aplicações, as quais geralmente são multitarefas. Para o FreeRTOS uma tarefa é composta por :

- Um estado que demonstra a atual situação da tarefa;
- Uma prioridade que varia de zero até uma constante máxima definida pelo projetista;
- Uma pilha na qual é armazenada o ambiente de execução (estado dos registradores) da tarefa quando esta é interrompida.

Em um sistema, uma tarefa pode assumir vários estados que variam de acordo com a sua atual situação. O FreeRTOS disponibiliza quatro tipos de estados diferentes para uma tarefa, sendo eles:

- **Em execução**: Indica que a tarefa está em execução;
- **Pronta**: Indica que a tarefa está pronta para entrar em execução, mas não está sendo executada;
- **Bloqueada**: Indica que a tarefa está esperando por algum evento para continuar a sua execução;
- **Suspensa**: Indica que a tarefa foi suspensa pelo gerenciador de tarefas através da chamada de uma funcionalidade usada para controlar as tarefas.

No FreeRTOS a alteração do estado de uma tarefa funciona como demonstra o diagrama da figura 2.2. Nela uma tarefa com o estado “em execução” pode ir para o estado pronta, bloqueado ou suspenso. Uma tarefa com o estado pronto pode ser suspensa ou entrar em execução, e as tarefa com o estado bloqueada ou suspensa só podem ir para o estado pronto.

Entretanto, vale enfatizar que por tratar-se de um SOTR para arquiteturas mono-processadores o FreeRTOS não permite que mais de uma tarefa seja executada ao mesmo tempo. Assim, em um determinado instante, apenas uma das tarefa com estado “pronto” pode assumir o processador e entrar no estado “em execução”. Com isso, para decidir qual tarefa entrará em executação

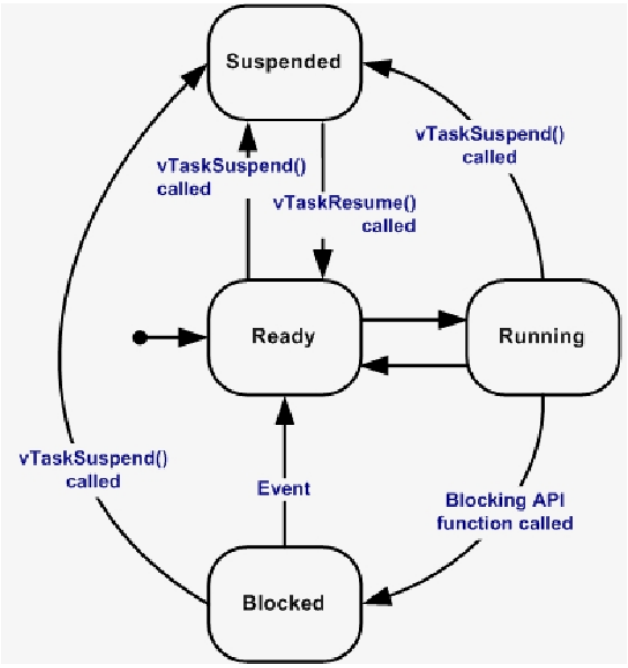


Figura 2.2: Diagrama de estados de uma tarefa no FreeRTOS [Richard Barry 2009]

o FreeRTOS possui um mecanismo denominado escalonador, o qual será detalhado na sessão 2.1.3.

Tarefa Ociosa

No FreeRTOS existe uma tarefa especial denominada Tarefa Ociosa que é executada, como o próprio nome já diz, quando o processador encontra-se ocioso, ou seja, quando nenhuma tarefa estiver em execução. Essa tarefa tem como principal finalidade liberar áreas de memórias que não estão sendo mais utilizadas pelo sistema. Por exemplo, quando uma aplicação cria uma nova tarefa, uma área da memória é reservada à nova tarefa, em seguinda, quando essa tarefa é excluída pelo sistema, a memória destinada ela continua oculpada, sendo esta liberada somente quando a tarefa ocioso entra em execução. A tarefa ociosa deve possui menor igual às demais tarefas do sistema e por isso só é executada quando nenhuma tarefa estiver em execução.

2.1.2 Co-rotinas

Outro conceito importante suportado pelo FreeRTOS é o conceito de co-rotina. Co-rotinas, assim como as tarefas, são unidades de execução independentes que formam uma aplicação. Elas também são formada por uma prioridade e um estado responsáveis respectivamente pela importância da co-rotina no sistema e pela situação da mesma.

Uma diferença crucial entre corotinas e tarefas está na presença da pilha de armazenamento do contexto. Co-rotinas não possuem contexto de execução próprio. Consequentemente, elas também não possuem uma pilha para o armazenar do contexto de execução. Diferentemente das tarefas que possuem contexto próprio devido a sua pilha de armazenamento do contexto.

Os estados que uma co-rotina pode assumir são:

- **Em execução**: Quando a co-rotina está em execução;
- **Pronta**: Quando a co-rotina está pronta para ser executada, mas não está em execução;
- **Bloqueada**: Quando a co-rotina está bloqueada esperando por algum evento para continuar a sua execução;

As transições entre os estados de uma co-rotina ocorrem como demonstra a figura 2.3. Nela uma co-rotina em execução pode ir tanto para o estado bloqueado como para o estado suspenso. Uma co-rotina de estado bloqueado só pode ir para o estado pronto e uma co-rotina de estado pronto só pode ir para o estado "em execução".

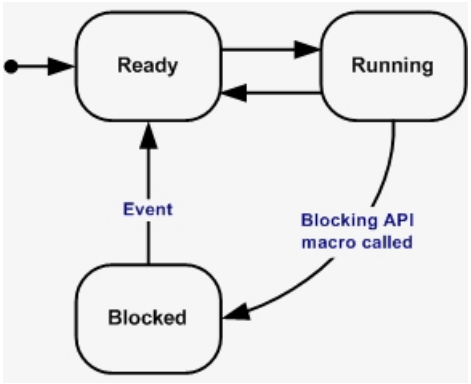


Figura 2.3: Grafo de estados de uma co-rotina

Assim como nas tarefas, a decisão de qual co-rotina irá entrar em execução é feita pelo o escalonador, o qual será demonstrado na seção 2.1.3

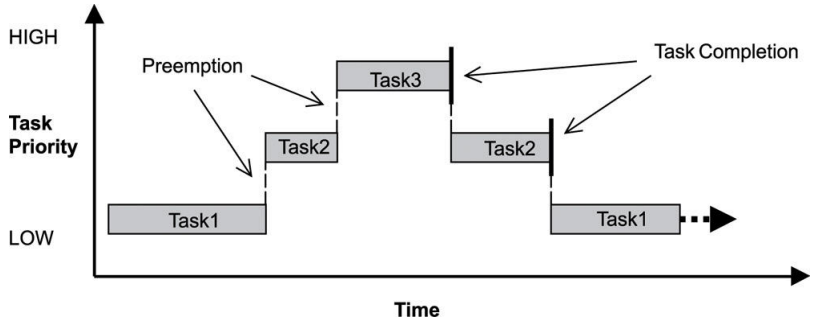
2.1.3 Escalonador de Tarefas

O escalonador é a parte mais importante de um sistema. É ele quem decide qual unidade de execução[2] irá entrar execução. Além disso, é ele que faz a troca entre a unidade em execução e nova unidade que irá entrar em execução. No FreeRTOS o escalonador pode funcionar de três modos diferentes:

[2]Aqui o termo "unidade de execução", as vezes citado apenas como unidade, refere-se a todas as tarefas(seção 2.1.1) e co-rotinas (seção 2.1.2) do sistema

- **Preemptivo**: Quando o escalonador interrompe a unidade em execução, alterando assim o seu estado, e ocupa o processador com outra unidade
- **Cooperativo**: Quando o escalonador não tem permissão de interromper a unidade em execução, tendo que esperar a mesma interromper-se para que ele possa decidir qual será a próxima unidade a entrar em execução e, em seguida realizar a troca.
- **Híbrido**: Quando o escalonador pode comporta-se tanto como preemptivo como cooperativo.

Para as tarefas o escalonador funciona de forma preemptiva, sendo que a decisão de qual tarefa deve entrar em execução e baseada na prioridade e segue a seguinte política: a tarefa em execução deve ter prioridade maior ou igual a tarefa de maior prioridade com o estado "pronta". Assim sempre que uma tarefa, com prioridade maior que a tarefa em execução, entrar no estado pronto, ela deve imediatamente entrar "em execução". Um exemplo claro da política preemptiva pode ser visto na figura 2.4, onde três tarefas, em ordem crescente de prioridade, disputam a execução do processador.



1. Tarefa 1 entra no estado pronto, como não há nenhuma tarefa em execução esta assume controle do processador entrando em execução
2. Tarefa 2 entra no estado pronto, como está tem prioridade maior do que a tarefa 1 ela entra em execução passando a tarefa 1 para o estado pronto
3. Tarefa 3 entra no estado pronto, como está tem prioridade maior do que a tarefa 2 ela entra em execução passando a tarefa 2 para o estado pronto
4. Tarefa 3 encerra a sua execução, sendo a tarefa 2 escolhida para entrar em execução por ser a tarefa de maior prioridade no estado pronto
5. Tarefa 2 encerra a sua execução e o funcionamento do escalonador é passado para a tarefa 1

Figura 2.4: Funcionamento de um escalonador preemptivo baseado na prioridade [Qing Li e Carolyn Yao 2003]

Para as co-rotinas o escalonador funciona de forma cooperativa e baseado na prioridade. Assim, a co-rotina em execução é quem decide o momento da sua interrupção, sendo que a

próxima co-rotina em entrar em execução será a co-rotina de maior prioridade entre as co-rotinas de estado pronto.

2.1.4 Bibliotecas

Para disponibilizar as caraterísticas discutidas nesta seção o FreeRTOS contem de um conjunto de bibliotecas de tipos e funções, as quais estão classificadas da seguinte forma: Criação de tarefas, controle de tarefas, utilidades de tarefas, controle do kernel e co-rotinas. A seguir tem-se em detalhe a descrição de cada uma dessas bibliotecas junto com os tipos e funcionalidades que as compõem.

Criação de Tarefas

Essa biblioteca é responsável pelo tipo tarefa e sua criação. Nela estão presente, um tipo, responsável por representar uma tarefa do sistema, e duas funcionalidades, uma para a criação e outra para a remoção de tarefa do sistema. Em seguida, tem-se em detalhes a composição da biblioteca criação de tarefas.

- **xTaskHandle** - Tipo pelo qual uma tarefa e referenciada. Por exemplo, quando uma tarefa é criada através do método *xTaskCreate*, este retorna uma referência para nova tarefa através do tipo; *xTaskHandle*
- **xTaskCreate** - Funcionalidade usada para criar uma nova tarefa para o sistema;
- **vTaskDelete** - Funcionalidade usada para remover uma tarefa do sistemas[3];

Controle de tarefas

A biblioteca de controle de tarefas realiza operações sobre as tarefas do sistema. Ela disponibiliza funcionalidades capazes de bloquear, suspender, retornar, informar a prioridade e alterar a prioridade de uma tarefa no sistema. A lista das principais funcionalidades presentes nessa biblioteca pode ser vista a seguir:

- **vTaskDelay** - Método usada para bloquear uma tarefa por um determinado tempo. Nesse método, para calcular o tempo que a tarefa deve permanecer bloqueada, é levado em consideração o tempo relativo, ou seja, o instante em que o método de bloqueio foi chamado.

[3] A memória alocada pela tarefa será liberada somente quando a tarefa ociosa entrar em execução 2.1.1

Esse método não é recomendado para a criação de tarefas cíclicas, pois o instante em que o método é chamado pode variar a cada execução da tarefa devido as interrupções que a mesma pode sofrer;

- **vTaskDelayUntil** - Método usado para bloquear uma tarefa por um determinado tempo. Esse método difere do *vTaskDelay* pelo o fato de que o tempo que a tarefa deve permanecer desbloqueada é calculado com base no instante do último desbloqueio da tarefa. Assim, se ocorrer uma interrupção no momento da chamada ao método o instante que a tarefa foi desbloqueada não ira mudar. Por isso esse método torna-se recomendável para a criação de tarefas cíclicas;
- **uxTaskPriorityGet** - Método usado para informar a prioridade de uma determinada tarefa;
- **vTaskPrioritySet** - Método usado mudar a prioridade de uma determinada tarefa;
- **vTaskSuspend** - Método usado para suspender uma determinada tarefa;
- **vTaskResume** - Método usado retornar uma tarefa suspensa.

Utilitários de tarefas

É através dessa biblioteca que o FreeRTOS disponibiliza, para o usuário, informações importantes a respeito das tarefas e do escalonador de tarefas. Nela estão presentes funcionalidades capazes de retornar uma referência para a atual tarefa em execução, retornar o tempo de funcionamento e o estado do escalonador e retornar o número e a lista das tarefas que estão sendo gerenciadas pelo sistema. Uma lista das principais funcionalidades dessa biblioteca é encontrada a seguir:

- **xTaskGetCurrentTaskHandle** - Retorna a uma referência para atual tarefa em execução;
- **xTaskGetTickCount** - Retorna o tempo decorrido desde a inicialização do escalonador;
- **xTaskGetSchedulerState** - Retorna o estado do escalonador;
- **uxTaskGetNumberOfTasks** - Retorna o número de tarefas do sistema;
- **vTaskList** - Retorna uma lista de tarefas do sistema.

Controle do Escalonador

Nessa biblioteca estão presentes as funcionalidades responsáveis por controlar as atividades do escalonador de tarefas. Nela encontramos funcionalidades que iniciam, finalizam, suspendem e retornam as atividades do escalonador. As principais funcionalidades presente nessa biblioteca são :

- **vTaskStartScheduler** - Método que inicia as atividades do escalonador. Usado para a inicialização do sistema;
- **vTaskEndScheduler** - Método que termina as atividades do escalonador. Usado para a finalização das atividades do sistema também;
- **vTaskSuspendAll** - Método que suspende as atividades do escalonador;
- **xTaskResumeAll** - Método que retorna as atividades de uma escalonador suspenso.

2.1.5 Co-rotinas

Biblioteca de co-rotinas

2.2 Comunicação e sincronização entre tarefa

Frequentemente tarefas necessitam se comunicar entre si. Por exemplo a tarefa "A" depende da leitura do teclado feito pela tarefa "B" para disponibilizar em uma tela as teclas digitadas pelo usuário. Com isso, percebe-se uma necessidade de que tal comunicação seja feita de maneira bem estruturada e sem interrupções.

A maioria dos sistemas operacionais oferecem vários tipos de comunicação entre as tarefas. Estas podem ocorrer da seguinte forma: Tarefas trocando informações entre si; Tarefas utilizando, de forma sicronizada, o mesmo recurso; Tarefas dependentes dos resultados produzidos por outras.

No FreeRTOS, assim como nos demais sistemas operacionais, os mecanismos responsáveis por realizar a comunicação entre as tarefas são as filas de mensagens, os semáforos e o mutexes (*Mutal Exclusion*). Para entender melhor como funciona a comunicação entre tarefas no FreeRTOS, cada um desses mecanismo será detalhado a seguir.

2.2.1 Fila de Mensagens

Filas de mensagens são estruturas primitivas de comunicação entre tarefas. Elas funcionam como, um túnel no qual tarefas enviam e recebem mensagem (figura 2.5). Assim, quando uma tarefa necessita comunicar-se com outra, ela envia uma mensagem para o túnel para que a outra tarefa possa ler a mensagem enviada.

Figura 2.5: Funcionamento de uma fila de mensagens

No FreeRTOS, uma fila de mensagens é formada por uma lista de tamanho fixo que armazena as mensagens, também de tamanhos fixos, enviadas para a lista. Assim, quando uma mensagem é envida para uma fila, uma cópia dessa mensagem é armazenada na lista para que posteriormente a outra tarefa possa ler a mensagem. Entretanto, contrária a copiar toda mensagem para a lista de mensagens, existe também a possibilidade de armazenar-se apenas uma referência da mensagem na lista, economizando assim a memória do sistema. Contudo, a técnica de armazenar apenas a referência da mensagen na lista torna mais complicado um trabalho do projetista, pois deste modo o acesso à mensagem na fila será compartilhado entre as tarefas interessadas, necessitando com isso de uma estrutura para sincronização o acesso da mensagem pelas tarefas. Na maioria das aplicações os projetistas utilizam a troca de mensagens através de cópia.

Além da lista de mensagens, uma fila de mensagens é composta por mais duas lista, uma para armazenar as tarefas que estão aguardando enviar uma mensagens para a fila e outra para armazenar as tarefas que estão aguardando receber uma mensagem da fila. Assim, quando uma tarefa tenta enviar uma mensagem para uma fila cheia, a tarefa é bloqueada e colocada na lista de tarefa que aguardam para enviar uma mensagem para fila até que um lugar na fila seja liberado. O mesmo acontece quando uma tarefa tenta ler uma mensagem de uma fila vazia, neste caso, a tarefa vai para a lista de tarefas aguardando por uma mensagem da fila.

No FreeRTOS, é possíveis definir o tempo máximo que uma tarefa pode ficará bloqueada esperado por uma fila (liberação de espaço ou chega de mensagem). E quando existirem mais de uma tarefa bloqueadas aguardando por um evento de uma fila, as tarefas de maior prioridade têm preferência sobre as demais.

2.2.2 Semáforo

Os semáforos são mecanismos usados para realizar a sincronização entre tarefas. Eles funcionam como uma chave que permite à tarefa executar uma operação ou acessar um recurso compartilhado. Assim, para uma tarefa acessar um recurso compartilhado, ela primeiramente deve solicitar o semáforo responsável pelo recurso, caso o semáforo esteja disponível, a tarefa utiliza o recurso, caso contrário, a tarefa é bloqueada até que o semáforo seja liberado.

O FreeRTOS disponibiliza dois tipo de semáforos, o semáforo binário e o semáforo com contador, sendo a diferença entre eles apenas o número de tarefas que podem manter o semáforo ao mesmo tempo. No semáforo binário é permitido que apenas uma tarefa mantenha o semáforo. Entretanto, no semáforo com contador existe um número fixo de tarefa, determinado por uma variável denominada contador, que podem reter o semáforo. Por exemplo no semáforo binário apenas uma tarefa pode acessar um recurso do sistema controlado por ele (conter o semáforo), diferente do semáforo com contador que um número de tarefas determinado pelo contador pode ter acesso ao recuros compartilhado por ele ao mesmo tempo.

O funcionamento do semáforo com contador ocorre da seguinte maneira. Primeiro o semáforo é criado com um contador que ira determinar o número de tarefas que podem reter o semáforo ao mesmo tempo. Em seguida, quando um tarefa retem o semáforo essa variáveis é decrementada de uma unidade. Assim ocorre até que o contador fique zerado, indicando que o semáforo está indisponível e o número máximo de tarefas foi atingido. Consequentemente, quando as tarefas liberam o semáforo, o seu contador é incrementado. O semáforo binário trata-se de um semáforo com o contador igual à um.

No FreeRTOS o semáforo binário funciona como uma fila de mensagens com um único item. Assim, quando a fila estiver vazia, indica que o semáforo está sendo usado e, quando a fila estiver cheia, indica que o semáforo está liberado. O mesmo ocorre para o semáforo com contador, só que nesse caso, o tamanho da fila será a quantidade de tarefas que podem reter o semáforo ao mesmo tempo, ou seja, o tamanho inicial do contador

2.2.3 Mutex

Mutexes são estruturas parecidas com os semáforos binário. A única diferença entre os dois é que o mutex implementa um mecanismo de herança de prioridade, o qual impede que uma tarefa, de maior prioridade, fique bloqueada a espera de um semáforo ocupado por outra tarefa, de menor prioridade, causando uma situação de bloqueio por inversão de prioridade.

O mecanismo de herança de prioridade funciona da seguinte forma, quando uma tarefa solicita o semáforo, ele primeiro verifica se a tarefa solicitante possui prioridade maior que a tarefa com o semáforo. Caso afirmativo, a tarefa que retém o semáforo tem, momentaneamente, a sua prioridade elevada, para que assim ela possa realizar a suas funções sem interrupções e, conseqüentemente, liberar o semáforo mais rápido.

2.2.4 Bibliotecas

As características de comunicação e sincronização entre tarefas está dividida em duas bibliotecas, Gerenciamento de fila de mensagens e Semáforo/Mutex. A seguir tem-se uma explicação de cada uma dessas bibliotecas.

Gerencialmente de fila de Mensagens

A biblioteca gerenciamento de fila de mensagens é responsável pela criação e utilização da estrutura de fila de mensagens. Ela é composta por funcionalidades que instanciam e removem filas de mensagens do sistema e pelas funcionalidades que enviam/recebem mensagens para/de uma fila de mensagens. Abaixo tem-se uma lista com as principais funcionalidades dessa biblioteca.

- **xQueueCreate** - Criar uma nova instância de fila de mensagens;
- **vQueueDelete** - Remover uma fila de mensagens do sistema;
- **xQueueSend** - Enviar uma mensagem para uma fila;
- **xQueueSendToBack** - Enviar uma mensagem para o fim da fila;
- **xQueueSendToFront** - Envia uma mensagem para o início da fila;
- **xQueueReceive** - Lê e remove uma mensagem da fila;
- **xQueuePeek** - Apenas lê uma mensagem da fila, sem remove-lá.

Semáforo/Mutex

Na biblioteca de semáforo e mutex estão implementadas as estruturas de sincronização entre tarefas (semáforo e mutex), junto com as suas operações. Assim, nesta biblioteca estão

presentes funcionalidades que criam e removem semáforos e mutex, e funcionalidades que solicitam e liberam os semáforos e os mutex. As principais funcionalidades desta biblioteca pode ser vista a seguir.

- **vSemaphoreCreateBinary** - Criar um semáforo binário;
- **vSemaphoreCreateCounting** - Criar um semáforo com contador;
- **xSemaphoreCreateMutex** - Criar um mutex;
- **xSemaphoreTake** - Solicitar a retensão de um semáforo ou de um mutex;
- **xSemaphoreGive** - Libera um semáforo ou um mutex retido.

2.3 Criação de uma aplicação utilizando o FreeRTOS

Para construir uma aplicação de tempo real utilizando o FreeRTOS o desenvolvedor deve seguir determinadas restrições imposta pelo sistema operacional. A maioria destas restrições são parâmetros de configuração e modelos para a criação de tarefas, rotinas, filas de mensagens e demais estruturas disponibilizadas pelo sistema operacional. Assim, com o intido de ajudar o desenvolvedor a criar suas primeiras aplicações, o FreeRTOS disponibilizou em seu código fonte aplicações exemplos organizadas de acordo com as plataformas alvo que ele suporta.

Entretanto, a criação e analise de uma nova aplicação no FreeRTOS é uma atividade que necessita de maior conhecimento sobre o funcionamento de suas funcionalidades, fugindo assim do objetivo geral desse capitulo que é proporcionar uma breve introdução ao FreeRTOS, demonstrando a suas principais funcionalidades e características. Com isso, para efeito de exemplificação de como são utilizadas as funcionalidades apresentadas neste capítulo, será demonstrado nessa seção, de forma abstrata, com é criada uma aplicação no FreeRTOS, abordando principalmente a criação e utilização de tarefas, filas de mensagens e semáforos.

2.3.1 Criação de tarefas

A tarefa é a parte mais importante de uma aplicação. É nela que são colocadas as rotinas que realizam as atividades da aplicação. No FreeRTOS, a rotina que compõe uma tarefa devem seguir a estrutura demonstrada pela figura 2.6. Nela tem-se inicialmente o nome da rotina, *functionName*, seguido de uma lista de parâmetros utilizados por ela, *vParameters*. O código

que realiza as finalidades da tarefa é colocado dentro de um laço infinito, forçando assim que a tarefa só finalize a sua execução quando for excluída pelo sistema[4].

```
void functionName( void *vParameters )
{
      for( ;; )
      {
          -- Task application code here. --
      }
}
```

Figura 2.6: Estrutura da rotina de uma tarefa

Um exemplo concreto da criação de uma rotina pode ser visto na figura 2.7. Nela tem-se a rotina *cyclicalTasks*, que utiliza a funcionalidade *vTaskDelayUntil* (seção 2.1.4) para bloquear a execução da tarefa em intervalos iguais de tempo. Essa funcionalidade possui como parâmetros, respectivamente, o último tempo que a tarefa foi retornada e o período que a tarefa deve permanecer bloqueada.

Entretanto, o desenvolvimento de uma rotina é apenas um dos passos para a criação de uma tarefa. É necessário ainda que a tarefa seja cadastrada no sistema junto com sua prioridade e pilha de contexto. Essa função é feita através do método *xTaskCreate*, demonstrado na figura 2.7 que possui como parâmetros os seguintes argumentos:

cyclicalTasks : Ponteiro para a rotina que deve ser executada pela tarefa;

"cyclicalTasks" : Nome da função utilizada nos arquivos de log do sistema;

STACK_SIZE : Tamanho da pilha de execução da função especificado de acordo com o número de variáveis declaradas na rotina da função;

pvParameters : Lista de valores dos parâmetros da rotina da função;

TASK_PRIORITY : Prioridade da tarefa;

cyclicalTasksHandle : Gancho de retorno da tarefa criada;

Após ser criada a tarefa que irá realizar a funcionalidade da aplicação, para finalizar o desenvolvimento de uma aplicação, resta apenas que o escalonador do sistema seja iniciado, iniciando assim as tarefas da aplicação. Essa operação é feita pela a funcionalidade *vTaskStartScheduler()*, presente no final da aplicação demonstrada pela figura 2.7.

[4] A execução além de ser finalizada pelo sistema pode também ser suspensa ou bloqueada como demonstrado na seção 2.1.1.

```
void cyclicalTasks( void * pvParameters ){
    portTickType xLastWakeTime;
    const portTickType xFrequency = 10;
    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount();
    for( ;; ){
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );
        // Perform action here.
    }
}

xTaskHandle cyclicalTasksHandle;

xTaskCreate( cyclicalTask, "cyclicalTasks", STACK_SIZE,
                ( void * ) pvParameters, TASK_PRIORITY, &cyclicalTasksHandle);

vTaskStartScheduler();
```

Figura 2.7: Aplicação formada por uma tarefa cíclica

2.3.2 Utilização da fila de mensagens

A utilização de uma fila de mensagens é resumidamente demonstrada na aplicação da figura 2.8. Nela, inicialmente tem-se que a estrutura *AMessage* define o tipo da mensagem que será utilizada. Em seguida, através do método *xQueueCreate* é criada uma fila de mensagens que será referenciada pela variável *xQueue*, do tipo *xQueueHandle*, tipo usado para referenciar uma fila de mensagens. Para isso o método *create* recebe como parâmetros, respectivamente, a quantidade de mensagens que a fila pode armazenar e o tamanho da mensagens manuseadas por ela, atributos necessários para a criação de uma fila de mensagens.

Estabelecida a fila de mensagens, é necessário agora, definir as tarefas que irão enviar e receber mensagens da mesma. Na figura 2.8 estão presentes apenas as rotinas de cada uma dessa tarefas, sendo que a explicação completa de como é criada uma tarefa foi demonstrada na seção 2.3.1.

Para criar a tarefa que enviará mensagens para a fila, a rotina da tarefa *sendTask* é desenvolvida. Nela esse trabalho é feita através da operação *xQueueSend*, que possui como parâmetros a fila para qual a mensagem será enviada, a mensagem que será enviada para a fila e o tempo máximo que a tarefa poderá ficar bloqueada aguardando para enviar uma mensagem para fila, caso esta esteja cheia.

Por último, na figura 2.8 a rotina da tarefa responsável por receber mensagens da fila, *send-*

```
struct AMessage {
    portCHAR ucMessageID;
    portCHAR ucData[ 20 ];
}xMessage;

xQueueHandle xQueue;
//Create a queue capable of containing 10 pointers to AMessage structures.
xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
// Task to create a queue and post a value.
void sendTask( void *pvParameters ) {
struct AMessage *pxMessage;

    if( xQueue == 0 ) {
        // Failed to create the queue.
    }else{
        // Send a pointer to a struct AMessage object. Don't block if the
        // queue is already full.
        pxMessage = & xMessage;
        xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );
    }
// ... Rest of task code.

}

// Task to receive from the queue.
void reciveTask( void *pvParameters ) {
struct AMessage *pxRxedMessage;

    if( xQueue != 0 ) {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) ) {
            // pcRxedMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }
    // ... Rest of task code.
}
```

Figura 2.8: Aplicação que utiliza uma fila de mensagens

Task, é criada. Para receber as mensagens enviadas para a fila, a função *sendTask* utiliza a operação *xQueueReceive*, a qual possui como argumentos, respectivamente, a fila de onde será recebida a mensagem, o local que irá armazenar a mensagem e o tempo máximo que a tarefa pode ficar esperando pela fila, caso está esteja vazia.

2.3.3 Utilização do semáforo

Semáforos são estruturas de sincronização entre tarefas. Eles são utilizados para coordenar o uso de recurso compartilhado por uma ou mais tarefas e para coordenar a execução de uma determinada função. Assim, o código da tarefa responsável por acessar o recurso compartilhado ou executar uma função sicronizada deve ser protegido de maneira que a sua execução só ocorra quando a tarefa possuir o semáforo.

Para a construção de uma aplicação que utiliza-se do semáforo são necessários basicamente três funcionalidades da biblioteca de semáforos, *vSemaphoreCreateBinary*, *xSemaphoreTake* e *xSemaphoreGive*. A primeira funcionalidade cria o semáforo e as demais solicitam e liberam o semáforo respectivamente.

Um exemplo de uma aplicação que utilizando um semáforo para controlar o acesso de um recurso compartilhado pode ser visto na figura 2.9. Nela inicialmente é criada a variável *xSemaphore* para armazenar uma referência para o novo semáforo. Em seguida, o método *vSemaphoreCreateBinary* é usado para criar o novo semáforo e retornar uma referência para o mesmo.

Após a criação do semáforo, a rotina da tarefa que utilizará o recurso compartilhado é desenvolvida. Nela o código que acessará tal recurso está protegido pelo segundo *if*, o qual recebe o retorno do método *xSemaphoreTake*, informando se o semáforo foi retido ou não. Ao final da rotina, o semáforo é liberado pelo método *xSemaphoreGive*, permitindo que outra tarefa possa reté-lo e usar o recurso compartilhado.

A utilização do mutex é bem parecida com a do semáforo binário. A diferença, para o usuário, entre os dois mecanismos está apenas no método de criação, *vSemaphoreCreateBinary* no semáforo binário, que será *xSemaphoreCreateMutex*. As formas e os métodos para solicitação do mutex são semelhante ao semáforo binário, sendo que *xSemaphoreTake* solita o mutex, informando o mutex e o tempo máximo de bloqueio e *xSemaphoreGive* libera o mutex. Com isso, para transformar a aplicação da figura 2.9 de semáforo para mutex basta apenas trocar o método *vSemaphoreCreateBinary* por *xSemaphoreCreateMutex*.

```
xSemaphoreHandle xSemaphore = NULL;


// Create the semaphore to guard a shared resource. As we are using
// the semaphore for mutual exclusion we create a mutex semaphore
// rather than a binary semaphore.
xSemaphore = xvSemaphoreCreateBinary();

// A task that uses the semaphore.
void semaphoreTask( void * pvParameters ) {
    // ... Do other things.
    if( xSemaphore != NULL ) {
        // See if we can obtain the semaphore. If the semaphore is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTake( xSemaphore, ( portTickType ) 10 ) == pdTRUE ) {
            // We were able to obtain the semaphore and can now access the
            // shared resource.
            // We have finished accessing the shared resource. Release the
            // semaphore.
            xSemaphoreGive( xSemaphore );
        } else
            // We could not obtain the semaphore and can therefore not access
            // the shared resource safely
       {
    }
}
```

Figura 2.9: Aplicação que demonstra a utilização de um semáforo

3 *Método B*

Métodos Formais proveêm abordagens formais para a especificação e construção de sistemas computacionais. Eles utilizam-se de conceitos matemáticos sólidos como lógica de primeira ordem e teorias dos conjuntos para a criação e verificação de sistemas consistentes, seguros e sem ambiguidades. Devido aos seus rigorosos métodos de construção, a sua principal utilização, embora timidamente, tem sido na criação de sistemas críticos para as indústrias de aeronáutica, viação férrea, equipamentos médicos e empresas que movimentam uma grandes quantidade monetárias, como os bancos.

Segundo a Honeywell [Honeywell 2005],uma empresa que desenvolve sistemas para aeronaves, a utilização de métodos formais no processo de desenvolvimento prove várias vantagens, entre eles estão:

- A produção mensurada pela corretude, métodos formais provê uma forma objetiva de mensura a corretude do sistema;
- Antecipação na detecção de erros, métodos formais são previamente usados em projetos de artefatos do sistema, permitindo assim uma detecção antecipada de erros;
- Garantia da corretude, através mecanismo de verificação é possível provar que sistema funcionará de forma coerente com a sua especificação inicial.

O método B[Abrial 1996] é uma abordagem formal usado para especificar e construir sistemas computacionais seguros. Seu criador Jean-Raymond Abrial, junto com a colaboração de outros pesquisadores da universidade de Oxford, procurou reunir no método B vários conceitos presentes nos demais métodos formais. Entre esses conceitos destacam-se as pré e pós condições[1], o desenvolvimento incremental através de refinamentos[2] e a modularização da especificação. A seguir tem-se a explicação de como e feito o desenvolvimento de sistemas utilizando o método B.

[1] Pré e Pós condições são...

[2] Refinamento trata-se de um técnica de desenvolvimento de sistema...

3.1 Etapas do desenvolvimento em B

O processo de desenvolvimento através do método B inicia-se com a criação de um módulo que define, em alto nível, um modelo funcional do sistema. Em B, esses módulos são denominados de Máquina Abstrata (*MACHINE*). Nessa fase de modelagem, técnicas semi-formais como UML podem ser utilizadas e em seguidas transformadas para a notação formal do método B. Após a criação dos módulos, esses são analisados estaticamente para verificar se são coerentes e implementáveis.

Uma vez estabelecido o modelo abstrato inicial do sistema, o método B permite que sejam construídos módulos mais concretos do sistema, denominados refinamentos. Mais especificamente, refinamentos correspondem a uma decisão de projetos na qual partes da especificação abstrata do sistema devem ser modeladas em um nível mais concreto. Assim um refinamento deve necessariamente estar relacionado com o módulo mais abstrato imediatamente anterior. Como ocorre na criação da máquinas abstratas, um refinamento também e passível de uma análise estática, na qual é verificada a relação entre refinamento com seu nível abstrato anterior.

Devido a técnica de refinamentos, o desenvolvimento de sistemas utilizando o método B pode chegar a um nível de abstração semelhante aos das linguagens de programação imperativas e sequenciais. Para isso sucessíveis refinamentos devem ser desenvolvidos até a especificação chegar em um último nível de refinamento denominado implementação (*IMPLEMENTATION*). Nesse nível a linguagem utilizada, chamada B0, é um formalismo algorítmico, passível de ser sintetizado para linguagens de programação com C, Java e JavaCard.

Assim o desenvolvimento de sistemas utilizando o método B é feito como demonstra a figura 3.1. Nela, os requisitos do sistema são inicialmente especificados em um alto nível de abstração e, após sucessivos refinamentos, um nível algorítmico do sistema é alcançado. Em seguida, essa especificação é sintetisada para um código de linguagem de programação, para o qual é possível, a partir da especificação funcional inicial, gerar teste para validar a sua correta transformação.

Atualmente, o desenvolvimento de sistemas utilizando o método B pode ser apoiado por diversas ferramentas com funcionalidades que vão da análise estática da especificação até a geração de código em linguagens de programação. Atualmente uma das mais famosas e completas ferramentas de apoio ao desenvolvimento de sistemas utilizando o método B é o AtelierB [ClearSY 2002]. Nela é possível, além da análise sintática e estática da especificação, gerenciar-se projetos, controlando as dependências entre os vários módulo que geralmente constituem uma especificação. Devido a suas vastas funcionalidades e popularidade o AtelierB será ado-

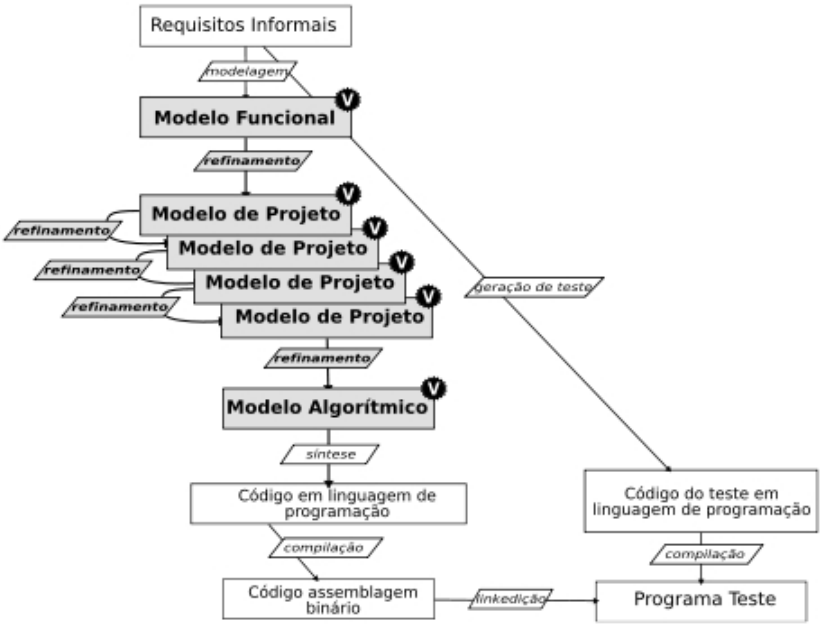


Figura 3.1: Etapas do desenvolvimento de sistema através do método B [Bartira Dantas et al. 2008]

tado como ferramenta padrão desse trabalho.

Um exemplo de sistemas desenvolvidos através da abordagem de B é o controle de porta do metrô de paris, desenvolvido pela Clearsy[**?**], empresa especialista em sistemas críticos. Esse sistema impede que o trem saia da estação quando a porta do mesmo estiver aberta, evitando assim possíveis acidentes.

3.2 Máquina Abstrata

A base do método B está na notação de máquina abstrata (em inglês: *Abstract Machine Notation* - AMN) a qual disponibiliza um framework comum para a especificação, construção e verificação estática de sistemas. Em outra palavras, a AMN trata-se de uma linguagem de especificação de sistemas formada por módulos básicos de construção chamados de Máquina Abstrata ou simplesmente Máquina.

Cada Máquina Abstrata é composta por diferentes seções, sendo que cada seção é responsável por definir um aspecto da especificação do sistema como: parâmetros, tipos, constantes, variáveis de estado, estados iniciais e transições do sistema. Por exemplo, a figura 3.2 contém uma Máquina Abstrata, chamada *Kernel*, a qual especifica um sistema que permite incluir e excluir tarefas até o limite de 10 tarefas e possui as seguintes seções:

MACHINE é onde inicia-se o código da máquina abstrata. Nela identifica-se a natureza e o nome do módulo, seguido opcionalmente por um ou mais parâmetros de máquina, os quais são separados por vírgula e limitados por parenteses;

SETS introduz um novo tipo de entidade, como é o caso de *TASK* no exemplo em questão. Nesse momento, nenhum detalhe é fornecido quanto à maneira como essa entidade será implementada;

VARIABLES informa o nome das diferentes variáveis que compõem o estado da máquina. No exemplo da figura 3.2, há apenas uma variável de estado, *tasks*;

INVARIANT especifica o tipo das variáveis de estado e os estados válidos do sistema. No exemplo em questão a variável *tasks* é um conjunto de até 10 elementos do tipo *TASK*. A caracterização lógica do conjunto dos estados válidos é uma das atividades mais importantes da especificação;

INITIALISATION identifica quais são os possíveis estados iniciais do sistema. No caso da figura 3.2 , *tasks* inicializado como um o conjunto vazio; e

OPERATIONS determina os diferentes tipo de eventos que o sistema pode sofrer. No nosso exemplo, temos operações para adicionar e eliminar um elemento de *tasks*. Uma operação pode ter parâmetros, resultados e pode alterar o valor de variáveis de estado. Um ponto importante encontrados nas operações são as pré-condições, a quais são condições que devem ser satisfeitas para que a operação seja realizada.

MACHINE
Kernel
SETS
TASK
VARIABLES
tasks
INVARIANT
$tasks \in \mathbb{P}(TASK) \wedge$
$\mathbf{card}(tasks) \leq 10$
INITIALISATION
$tasks := \emptyset$

OPERATIONS

$task\_add(task) =$
PRE
$task \in TASK \wedge$
$task \notin tasks \wedge$
$\mathbf{card}(tasks) < 10$
THEN
$tasks := tasks \cup \{task\}$
END;

...
$task\_delete(task) =$
PRE
$task \in tasks$
THEN
$tasks := tasks - \{task\}$
END
END

Figura 3.2: Maquina abstrata de tarefas

Para uma melhor compreensão, a especificação de sistemas através das maquinas abstratas será resumidamente dividida em duas partes principais. Na primeira parte, serão colocadas informação a respeito dos estados da máquina, suas variáveis e restrições. Na segunda parte, será

especificado o comportamento da máquina, ou seja, a sua parte dinâmica como a inicialização e as operações. Essas duas partes serão melhor discutidas a seguir.

3.2.1 Especificação do estado da máquina

Na parte do estado de uma máquina são definidos os estado que a máquina pode assumir. Estes são definidos através de suas variáveis e dos seu invariante. As variáveis são os diferentes elementos que compõem o estado do sistema. O invariante define os valores que essas variáveis podem assumir. Assim uma especificação só pode garantir o correto funcionamento da máquina quando esta encontra-se em um estado válido, nada podendo afirmar para os demais estados.

O estado de uma máquina é especifico, por meio de calculo de predicados, da teoria do conjunto e relações entre conjuntos, permitindo com isso uma análise estática da máquina através da lógica matemática. No exemplo da figura 3.2 o estado da máquina *kernel* foi especificado como sendo a variável *tasks* e seu predicados $tasks \in \mathbb{P}(TASK)$ e $\mathbf{card}(tasks) \leq 10$, os quais definem que *tasks* deve ser um conjunto de *TASK* e que o tamanho máximo permitido para o conjunto é de dez elementos.

3.2.2 Especificação das operações da máquina

Nas operações da máquina é especificado o comportamento dinâmico do sistema. É através das operações que o estado da máquina é alterado, respeitando sempre as suas restrições. Mais especificamente, as condições declaradas no invariante da máquina devem ser sempre satisfeita ao final da operação, levando assim a máquina a um estado válido.

O cabeçalho de uma operação é composto por um nome, uma lista de parâmetros de entra e uma lista de parâmetros de saída [3], sendo que os parâmetro de entrada e os parâmetros de saída são argumentos opcionais. Assim o exemplo de uma operação com parâmetros de entrada e saída pode ser visto na figura 3.3, na qual o nome da operação é *query_task*, o parâmetro de entrada é *task* e o parâmetro de saída é *ans*.

A operação propriamente dita é formada por pré-condição e corpo da operação. Na pré-condição, são colocadas as informações sobre todos os parâmetros de entrada e as condições que devem ser satisfeitas para que a operação seja executada. Com isso, a pré condição funciona como uma premissa que deve ser suprida para que a operação funcione corretamente.

Por exemplo, na figura 3.2 para que a operação (add_task) funcione corretamente e não leve

[3] A notação de máquina abstrata permite que uma operação retorne mais de um parâmetro.

a máquina para um estado inválido, as pré-condições $task \in TASK$, $task \notin tasks$ e $\mathbf{card}(tasks) < 10$ devem ser obedecidas.

$ans \leftarrow query\_task(task) =$
PRE $task \in TASK$
THEN
  IF $task \in TASK$
  THEN $ans := yes$
  ELSE $ans := no$
END

Figura 3.3: Operação que consulta se uma tarefa pertence a máquina *Kernel*

No corpo da operação é especificado o seu comportamento. Nele os parâmetros de saída devem ser obrigatoriamente valorados e os estados da máquina podem ser alterados ou consultados. Assim, para realizar as atualizações de estado e definições de parâmetros de saída de modo formal, a notação de máquina abstrata possuí um conjunto de atribuições abstratas, denominadas substituições as quais serão declaradas explicadas a seguir.

Formalmente uma substituição é um transformador de predicados (ou de formulas). Assim se P for um predicado, $[S]P$ é um predicado resultado da aplicação da substituição de S a P. A seguir são demonstrados algumas das principais substituições da AMN e como são transformados os predicados utilizados por elas.

Substituição Simples

A substituição simples é definida da seguinte forma:

$$x := E$$

Nela x é uma variável de máquina ou parâmetro de saída, para o qual será atribuído o valor da expressão E. Mais precisamente uma substituição é interpreta da seguinte maneira:

$$[x := E]P \Rightarrow P(x \setminus E)$$

Assim tem-se que no predicado P a variável x deve ser substituída por E.

Substituição Múltipla

A substituição múltipla é uma generalização da substituição simples. Ela permite que várias variáveis sejam atribuídas simultaneamente. Assim uma substituição múltipla utilizando duas

variável tem a seguinte forma:

$$x, y := E, F$$

Na definição acima às variáveis x e y são atribuídos os valores das expressões E e F, respectivamente. Assim da mesma forma que a substituição simples, a múltipla substituição é definida da seguinte maneira:

$$[x := E, y := F]P \Rightarrow P[E, F \setminus x, y]$$

Na qual no predicado P as variáveis x e y são substituidos por E e F, respectivamente. Por exemplo, $x, y := y + 5, x + 10$ resulta em $y + 5 < x + 10$.

Substituição Condicional

As substituições simples e múltipla permitem somente uma opção de especificação onde uma atribuição é sempre feita de maneira uniforme, sem opções e sem levar em consideração os estados iniciais da operação. Entretanto, as linguagens de programação convencionais disponibilizam um tipo condicional de atribuição na qual é permitido caminhos diferentes de acordo com expressões lógicas que utilizam os valores iniciais das variáveis do sistema e os parâmetros passados.

Como nas linguagens de programação, a notação de máquina abstrata também permite a construção de atribuições condicionais, as quais são feitas através da substituição condicional. Com isso, uma substituição condicional funciona da mesma forma que nas linguagens de programação. Nela uma expressão lógica é avaliada para saber qual caminho a estrutura deve seguir, e qual atribuições devem ser realizada. A forma como é especificada uma substituição condicional na ANM pode ser visto a seguir:

IF E **THEN** S **ELSE** T **END**

Na especificação acima S e T são substituições quaisquer. Elas tem as suas aplicações condicionadas pela expressão lógica E, que pode conter variáveis da máquina e parâmetros de entrada. Com isso, caso E seja afirmativo a substituição S é realizada e, caso ele seja, negativo a substituição T é executada. Assim uma substituição condicional pode ser interpretada da seguinte forma:

$$[\textbf{IF} \quad E \quad \textbf{THEN} \quad S \quad \textbf{ELSE} \quad T]P \quad = \quad (E \Longrightarrow [S]P) \wedge (\neg E \Longrightarrow [T]P)$$

.

Nessa interpretação se E for verdadeiro a substituição S é aplicada ao predicado P. Caso contrário, a substituição T é aplicada ao predicado P.

Um exemplo simples da utilização dessa substituição pode ser visto na figura 3.3. Nela a expressão $task \in TASK$ é primeiramente analisada para decidir qual substituição simples deve ser executada. Caso a o resultado da expressão seja afirmativo $ans := yes$ é executada e caso a expressão seja negativa $ans := no$ é executada.

Substituição não determinística ANY

As substituições vistas até agora seguem uma metodologia determinística, ou seja, são substituições que possuem um comportamento previsível, que levam a apenas um resultado final e pré-determinado. Entretanto, as máquinas abstratas em B são utilizadas para fazer especificações iniciais de sistemas ou componentes e, na maioria das vezes, no início de uma especificação, o comportamento do sistema não é totalmente conhecido. Assim, para especificar o indeterminismo inicial de um sistema a notação de máquina abstrata disponibiliza um tipo especial de substituições determinadas como substituições não determinísticas.

Substituições não determinísticas são substituições que introduzem escolhas aleatórias no corpo das operações, levando a operação a um conjunto de estados finais diferentes a cada execução da operação. Assim em uma substituição não determinística a especificação define apenas o conjunto sobre o qual deve ser feita a escolha, abstraindo assim informações de como tal escolha deve ser realizada. Em outras palavras, em uma substituição não determinística existe um conjunto de estado finais possíveis que podem ser alcançados a cada execução da substituição.

Uma substituição não determinística definida na AMN é a substituição **ANY**. Essa substituição possui o seguinte formato:

$$\textbf{ANY} \quad x \quad \textbf{WHERE} \quad Q \quad \textbf{THEN} \quad T \quad \textbf{END}$$

Através da definição acima percebe-se que a substituição **ANY** é formada por três elementos:

x é uma lista de variáveis que serão utilizadas no corpo da substituição. Para essa variáveis serão atribuidos valores abstratos delimitados pelo o predicado Q;

Q são os predicados que delimitam o conjunto de opções para as variáveis x. Nessa parte as variáveis x devem obrigatoriamente serem tipificadas; e

T é uma substituição que utilizam-se das variáveis x para atualizar estados ou atribuir valores para os parâmetros de saída da operação.

Um exemplo da substituição **ANY** pode ser visto na operação da figura 3.4. Nela uma tarefa é aleatoriamente adicionada na máquina *Kernel*. Para isso, primeiramente a variável *task* é criada para armazenar um valor aleatório. Em seguida, o tipo e a restrição sobre *task* é definida, conjunto da escolha aleatória. Por último, a variável *task* é adicionada ao conjunto *tasks*. Assim um comportamento não determinístico é atribuído à operação, pois para cada execução da operação a variável *task* pode assumir um valor aleatório.

random_create =
PRE
  card(*tasks*) < 9
THEN
  ANY
    task
  WHERE
    $task \in TASK \wedge$
    $task \notin tasks$
  THEN
    $tasks := \{tasks\} \cap task$
  END
END

Figura 3.4: Operação que cria uma tarefa aleatória na máquina *Kernel*

Uma definição para a substituição **ANY** seria:

$$\textbf{ANY} \quad x \quad \textbf{WHERE} \quad Q \quad \textbf{THEN} \quad T \quad \textbf{END} \Rightarrow \forall x.(Q \Rightarrow [T]P)$$

Indicando que para todo valor que for escolhido para o conjunto de variável x que satisfaça Q as substituições T devem aplicadas ao predicado P.

3.3 Obrigação de Prova

Após a criação de uma máquina abstrata utilizando o método B, essa deve ser avaliada estaticamente para saber se a mesma é coerente e passível de implementação. Para realizar tal avaliação o método B dispõe de um conjunto de obrigações de prova que são expressões lógica geradas a partir de uma especificação em B.

Resumidamente, a análise estática de uma máquina abstrata, através das obrigações de prova, avalia primeiramente se a máquina possui estados válidos, ou seja, se pelo menos uma combinação do estados é alcançada pela máquina. Caso a máquina possua estados válidos, é avaliado se estes são alcançados ao final de cada operações e na inicialização da máquina. Com isso, as principais obrigações de prova gerada em uma máquina abstrata são: Consistência do invariante, Obrigação de prova da inicialização e Obrigação de prova das operações. A seguir tem-se em maior detalhes cada uma dessas obrigações de prova e como elas são geradas.

3.3.1 Consistência do Invariante

Nessa obrigação de prova é analisado se o invariante da máquina possui pelo menos uma combinação na qual todas variáveis tem valores válidos, ou seja, a máquina possui pelo menos um estado válido. Assim essa obrigação de prova é definida da seguinte maneira :

$$\exists v.I$$

Onde v indica o vetor de todos as variáveis da máquina e I representa o invariante da máquina. Com isso, a definição acima pode ser entendida como: deve existir pelo menos um valor para o vetor de variáveis v que satisfaça o invariante I.

Um exemplo da aplicação desse obrigação de prova na máquina da figura 3.2 seria:

$$\exists tasks.(tasks \in TASK \cap \mathbf{card}(tasks) \leq 10)$$

O que pode ser provado como verdadeiro para $tasks = \emptyset$.

3.3.2 Obrigação de prova da inicialização

Outra obrigação de prova necessária na analise estática da máquina abstrata é a obrigação de prova da inicialização. Nela analisa-se se os estados iniciais da máquina satisfazem seu

invariante. Isso significa, verificar se o estados iniciais da máquina são estado válido. Assim, essa obrigação de prova é definida da seguinte maneira :

$$[T]I$$

Nela $[T]$ indica as substituições realizadas na inicialização da máquina e I indica as restrições definidas no invariante. Com isso, a obrigação de prova da inicialização da máquina da figura 3.2 pode ser definida como sendo:

$$[task := \emptyset](tasks \in TASK \cap \mathbf{card}(tasks) \leq 10) \quad \Rightarrow \quad \emptyset \in TASK \cap \mathbf{card}(\emptyset) \leq 10$$

O que pode ser facilmente provado como válido.

3.3.3 Obrigação de prova das Operações

Na obrigação de prova das operações deve ser analisado se, quando satisfeita a sua pré-condição, a execução da operação, apartir de um estado válido, levará a máquina a um estado válido. Assim a definição dessa obrigação de prova pode ser vista da seguinte maneira:

$$I \wedge P \Rightarrow [S]I$$

Na definição acima I representa o invariante da máquina, P representa a pré-condição da operação analisada e S indica as substituições realizadas no corpo da operação. Assim, uma explicação mais precisa dessa definição seria: quando a máquina estiver em um estado válido e a pré-condição da operação for satisfeita, a execução da operação deve manter a máquina em um estado válido. Nota-se assim que esta obrigação de prova não é necessária nas operações que não alteram o estado da máquina, chamadas de operações de consulta, como a da figura 3.3. Pois nessas operações apenas o valor do parâmetro de retorno é alterado.

Um exemplo de uma obrigação de prova da operação *add_task* da máquina da figura 2.2 pode ser visto a seguir:

$$\begin{aligned}&(tasks \in TASK \wedge \mathbf{card}(tasks) \leq 10) \wedge (task \in TASK \wedge task \notin tasks) \Rightarrow \\ &([tasks := task \cap task]((tasks \in TASK \cap \mathbf{card}(tasks) \leq 10))))\end{aligned}$$

3.4 Refinamento

A linguagem abstrata demonstrada até agora é usada principalmente para criar uma modelagem funcional de sistemas e componentes. Nela o principal objetivo é descrever o comportamento do sistema sem se preocupar com detalhes de como tal comportamento será implementado ou de como os dados serão manipuladas pelo computador. Entretanto, para realizar uma modelagem mais concreta e passível de implementação é necessário que notações matemáticas abstratas utilizadas na modelagem do sistema, como conjuntos e substituições não determinística, sejam descritas de forma mais concreta.

Através da técnica de refinamento o método B possibilita um desenvolvimento gradativo do sistema. Nele um sistema é especificado em estágios que vão da modelagem abstrata até um nível algoritmo denominada de implementação. Entre esse níveis de abstração existem modelos intermediário chamados de refinamentos, que combina especificações de construção e detalhes de implementação.

Mais precisamente, refinamentos são decisões de projeto nas quais estruturas abstratas são detalhadas em um nível mais concreto. Com isso, um refinamento deve obrigatoriamente estar ligado a um modelo abstrato anterior e possuir seu comportamento delimitado pelo o modelo a qual está relacionado. Para garantir que essa relação entre módulo seja feita de forma coerente, existem mecanismo de análise estática denominados obrigações de prova do refinamento, o qual será detalhado na seção 3.4.3.

A construção de um refinamento é muito parecida com a construção de uma máquina abstrata. Ele, assim como a máquina abstrata, é dividido em seções onde são especificadas as informações do sistema. Um refinamento possuindo basicamente as mesma seção de uma máquina abstrata, a diferença está na seção, **REFINEMENT** e **REFINES**, onde são colocados respectivamente o nome do refinamento e o módulo que será refinado.

Como ocorreu na seção da máquina abstrata **??**, para um melhor entendimento, a especificação de um refinamento será dividida basicamente em duas parte principais: refinamento do estado da máquina abstrata e refinamento das operações da máquina abstrata. Em seguida tem-se o delineamento dessas duas partes principais.

3.4.1 Refinamento do Estado

No refinamento de dados, como é reconhecido o refinamento do estado, tem-se o objetivo de especificar o estado de uma máquina em uma forma mais concreta, ou seja, mais próxima à

utilizada pelo computador. Para isso, estruturas abstratas como conjuntos e relações são substituídas por mecanismo mais implementáveis como vetores e sequências.

Como foi dito anteriormente, um refinamento necessita estar relacionado com um nível abstrato ligeiramente acima dele. No refinamento de dados essa relação é feita através de um mecanismo denominado *relação de refinamento*. Assim, a *relação de refinamento* nada mais é do que conjunções lógicas que ligam o estado do refinamento ao estado do módulo refinado por ele.

Um exemplo de refinamento de dados pode ser visto na figura 3.5. Nela é feito o refinamento da máquina *Kernel* (figura 3.2),a qual possui o estado *task* especificado como sendo um conjunto de tarefas. Entretanto, conjuntos são representações abstratas de dados. Assim no refinamento *KernelR* o estado *task* é refinado por *taskR*, uma sequência de tarefas, estrutura mais concreta que um conjunto. A relação de refinamento entre os dois estados é feita através da igualdade $\mathbf{ran}(tasks\_r) = tasks$.

```
REFINEMENT   INVARIANT                  OPERATIONS
  Kernel_r     tasks_r ∈ seq(TASK) ∧    task_add(task) =
REFINES        ran(tasks_r) = tasks       BEGIN
  Kernel     INITIALISATION               tasks_r :=
VARIABLES      tasks_r := []                    task → tasks_r
  tasks_r                                 END
                                        END
```

Figura 3.5: Refinamento da maquina abstrata de *Kernel*

3.4.2 Refinamento das Operações

Após feito o refinamento do estado da máquina é necessário agora especificar o refinamento das suas operações. O refinamento das operações de uma máquina deve garantir que está possua os mesmo comportamentos que suas operações abstratas.

As operações de um refinamento devem possuir a mesma assinatura das operações do módulo relacionado à ele, ou seja, ter os mesmo nomes e parâmetros de entrada e saída. Entretanto, nas operações de um refinamento não é necessário a declaração da pré condição (**PRE**), uma vez que essa foi definida em um nível mais abstrato e é suficiente para garantir que o tipo do parâmetro de entrada permaneça o mesmo.

Um exemplo do refinamento de uma operação pode ser visto na operação *add_task* do refinamento *KernelR* (figura 3.5). Nela percebe-se a ausência da pré-condição e que a assinatura

da operação permanece a mesma. A parte alterada foi apenas o corpo da operação que foi adaptada para trabalhar com o estado *taskR*.

3.4.3 Obrigação de Prova do refinamento

A analise estática para saber se um refinamento é consistente com o nível abstrato acima dele é feita através de obrigações de prova e pode ser dividida em duas partes, obrigação de prova da inicialização e obrigação de prova das operações. Entretanto, na obrigação de prova das operações são possíveis dois tratamentos diferentes, obrigações de prova para as operações sem parâmetros de retorno e a obrigação de prova para as operações com parâmetros de retorno. A seguir é demonstrada como é realizada cadas umas dessas obrigações de prova.

Obrigação de Prova da Inicialização

Em geral a inicialiazação da máquina abstrata T e a inicialização do refinamento *T1* possuem um conjunto de execuções possíveis que levam a um conjunto de diferentes estados. Assim em um refinamento é necessário que cada execução de *T1* possua uma execução correspondente em T. Em outras palavras, todo estado encontrado em *T1* deve possuir, via *relação de refinamento* J, um estado gerado por T.

A *relação de refinamento* J é um predicado entre variáveis abstratas e variáveis de refinamento. Com isso T deve possuir pelo menos uma transição que satisfaça esse predicado, ou seja, nem toda transição de T levará J a falsidade. Essa afirmativa pode ser traduzida na expressação abaixo:

$$\neg[T]\neg J$$

O predicado $\neg J$ indica que J é falso e o predicato $[T]\neg J$ representa que todo transformação de T levará J a um estado falso. Assim a negação dessa afirmação $\neg[T]\neg J$ indica quem existem uma transição de T que não levará J à falsidade, ou seja, nem toda transição de T levará J a falsidade.

Assim para encerrar a obrigação de prova da inicialização do refinamento é necessário que para toda transformação de *T1* o predicado $\neg[T]\neg J$ seja estabelecido. Essa afirmação pode ser traduzida na expressão abaixo, a qual representa a obrigação de prova do refinamento

$$[T1]\neg[T]\neg J$$

Um exemplo de uma obrigação de prova da inicialização do refinamento pode ser visto entre as máquina *Kernel* e a máquina *KernelR*. Nela a inicialização dos estados das duas máquina gera a seguinte obrigação de prova:

$$[tasks\_r := []]\neg[tasks := \emptyset]\neg(\mathbf{ran}(tasks\_r) = tasks)$$

Obrigação de prova da operação sem parâmetro de retorno

Geralmente uma operação é definida como **PRE** *P* **THEN** *S* **END**, sendo o seu refinamento **PRE** *P1* **THEN** *S1* **END**, onde na maioria da vezes *P1* é verdadeiro. Com isso, do mesmo modo que na inicialização, tem-se que as trasições geradas por *S1* devem estar relacionada com alguma transição de *S*, o que é definido pela expressão abaixo:

$$[S1]\neg[S]\neg J$$

Entretanto, diferente da inicialização, a execução de uma operação deve levar em consideração o estado da máquina anterior à sua execução. Assim, o estado da máquina abstrata junto com o estado do seu refinamento devem ser estados válidos. Uma relevante ligação entre esse estados e o invariante *I* é a sua relação de refinamento *J*. Além disso, para a correta execução da operação a pré-condição da mesma deve ser estabelecida. Assim, levando em consideração que uma operação só pode ser executada corretamente quando a máquina estiver em um estado válido e a sua pré-condição for estabelecida, a obrigação de prova de uma operação é feita da seguinte forma:

$$I \wedge J \wedge P \Rightarrow S1\neg S\neg J$$

Por exemplo, a obrigação de prova do refinamento da operação *add_task* da máquina *Kernel* é :

$$
\begin{aligned}
&(tasks \in \mathbb{P}(TASK) \wedge \mathbf{card}(tasks) \leq 10) \wedge \\
&tasks\_r) = tasks \wedge \\
&task \in TASK \wedge \\
&task \notin tasks \wedge \\
&\mathbf{card}(tasks) < 10 \Rightarrow [tasks := tasks \cup \{task\}] \\
&\neg [tasks\_r := task \rightarrow tasks\_r] \\
&\neg (tasks\_r) = tasks)
\end{aligned}
$$

Obrigação de prova da operação com parâmetro de saída

No refinamento de uma operação com saídas, cada saída possível para o refinamento de uma operação deve estar ligada a uma saída da operação especificada. Assim denominando out' como os parâmetros de saída do refinamento e *out* como os parâmetros de saída da especificação tem-se que cada valor de out' deve possuir um correspondente em *out*. Em outra palavra, cada execução de *S1* deve encontrar uma execução *S*, na qual out' produzido por *S1* seja igual ao *out* produzido por *S*.

Além da ligação entre os parâmetros de saída, no refinamento de uma operação com retorno, deve-se obedecer todas as restrições impostas para o refinamento das operações sem parâmetro de saída, ficando da seguinte maneira a expressão da obrigação de prova para as operações com retorno:

$$I \wedge J \wedge P \Rightarrow S1[out'/out] \neg S \neg (J \wedge out' = out)$$

Nela $[out'/out]$ significa que na atribuições de *S1* os valores de *out* devem ser substituídos por out' e manter verdadeira a preposição, garantindo que cada produção de *out* de *S1* tenha uma correspondente no conjunto *out*.

4 Revisão Literária

- Enumerar Projetos
- Desafio de software verificado

5 *Proposta*

- Como será feita a modelagem do FreeRTOS
- Falar do estudo do FreeRTOS e identificação dos seus principais conceitos e funcionalidades
- O desenvolvimento progressivo acrescentando novas funcionalidades a cada refinamento
- Ligar a abordagem do compilador verificável ao FreeRTOS
- Dizer como será ou deve feita a união do FreeRTOS para o compilador verificável

6 Atividades e Etapas

Referências Bibliográficas

[Abrial 1996]ABRIAL, J. R. *The B-book: assigning programs to meanings*. [S.l.]: Cambridge University Press, 1996.

[Bartira Dantas et al. 2008]Bartira Dantas et al. Proposta e avaliacao de uma abordagem de desenvolvimento de software fidedigno por construcao com o mÃ©todo b. In: *Anais do XXVIII Congresso da SBC*. [S.l.: s.n.], 2008.

[ClearSY 2002]ClearSY. *atelieb manuel-reference*. 2002. URL:http://www.atelierb.eu.

[David Kalinsky 2003]David Kalinsky. *Basic concepts of real-time operating systems*. [S.l.], 2003.

[Honeywell 2005]Honeywell. *Formal Method: analysis of complex systems to ensure correctenes and reduce cost*. [S.l.], 2005.

[Qing Li e Carolyn Yao 2003]Qing Li; Carolyn Yao. *Real-Time Concepts for Embedded Systems*. [S.l.]: CMP Books, 2003.

[Richard Barry 2009]Richard Barry. *Using the FreeRTOS Real Time Kernel - A Pratical Guide*. [S.l.]: FreeRTOS.org, 2009.