Stephenson de S. L. Galvão

Modelagem do Sistema Operacional de Tempo Real FreeRTOS

Natal - Rn, Brasil

1 de junho de 2009

Stephenson de S. L. Galvão

Modelagem do Sistema Operacional de Tempo Real FreeRTOS

Qualificação de mestrado apresentada ao programa de Pós-gradução 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

Falar dos grandes desafios (SBC) e do desafio do compilador “Verifiging compile”, “Verified Software repository” desafio de Jim Woodcock

1.1 Objetivos

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

- Abragência 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 compoem o FreeRTOS.

1.2 Metodologia

Métodologia da dissertação, no contexto do que já foi feito

2 FreeRTOS

O FreeRTOS é um sistema operacional de tempo real 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 monoprocessadores diferentes, entre elas a PIC, ARM e Zilog Z80, as quais são amplamente difundidas em produtos comerciais através de sistemas computacionais embutidos.

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

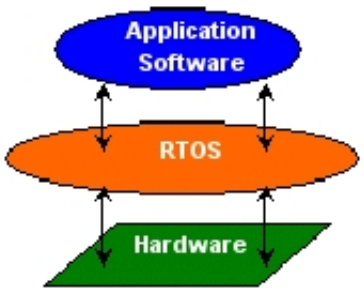


Figura 2.1: Camada abstrata proporcionada pelo FreeRTOS

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.

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

A criação de uma aplicação utilizando o FreeRTOS pode ser divida em duas partes. Na primeira parte são criadas, de acordo com modelos fornecidos pelo FreeRTOS, as tarefas e demais estruturas disponibilizadas pelo FreeRTOS que serão utilizadas pela a aplicação. Na segunda parte é feito o cadastramento das tarefas utilizadas pelo o sistema assim como a inicialização do mesmo. Por fim, o sistema é transformado em uma aplicação da arquitetura desejada.

A seguir tem-se em detalhe os principais serviços providos pelo o FreeRTOS junto com a biblioteca que os disponibilizam.

2.1 Gerenciamento de Tarefas e CoRotinas

2.1.1 Tarefa

Para entender como fuciona 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, que 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 o usuário
- Uma pilha na qual é armazenada o ambiente de execução (estado dos restradores) da tarefa quando está é interrompida.

Em um sistema uma tarefa pode assumir vários estados de acordo com a sua circustância. Assim o FreeRTOS disponibiliza quatro tipos de estados diferentes para uma tarefa. Os estados que uma tarefa pode assumir no FreeRTOS são:

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

A permutação que ocorre no estado de uma tarefa funciona como demonstra a 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.

Entranto, vale enfatizar que por tratar-se de um SOTR para arquiteturas monoprocessadores o FreeRTOS não permite que mais de uma tarefa seja executada ao mesmo tempo. Assim,em um determinado instante, apenas uma tarefa pode assumir o estado pronto, restando as demais os outros estado. Com isso, para decidir qual tarefa será executada o FreeRTOS possui um mecanismo denominado escalonador, o qual será detalhado na sessão 2.1.3.

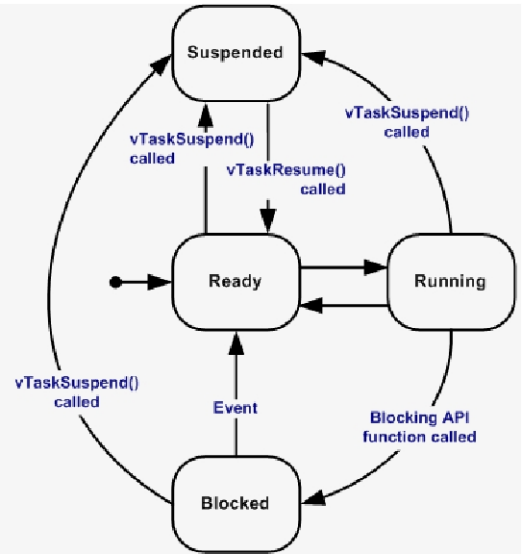


Figura 2.2: Grafo de estados de uma tarefa

Tarefa Ociosa

No FreeRTOS existe também uma tarefa denominada de tarefa ociosa, a qual é executada quando nenhuma tarefa está em execução. A tarefa ociosa tem como principal finalidade excluir da memória tarefas que não serão mais usadas pelo sistema. Assim, quando uma aplicação informa para o sistema que uma tarefa não será mais utilizada, esta tarefa só será excluida quando a tarefa ociosa entrar em execução. A tarefa ociosa possui a menor prioridade dentre as tarefas que compoem um sistema.

2.1.2 Corotinas

Outro conceito importante suportado pelo FreeRTOS é o de Corotina. Assim como as tarefas, corotinas são unidades de execução independentes que formam uma aplicação. Por isso

assim como as tarefas, uma corotina é formada por uma prioridade e um estado. Entretanto, divido ao fato de corotinas não possuirem contexto de execução próprio, ela não possuem uma pilha para armazenar o contexto de execução, como ocorre nas tarefas.

Os estados que uma corotina pode assumir são:

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

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

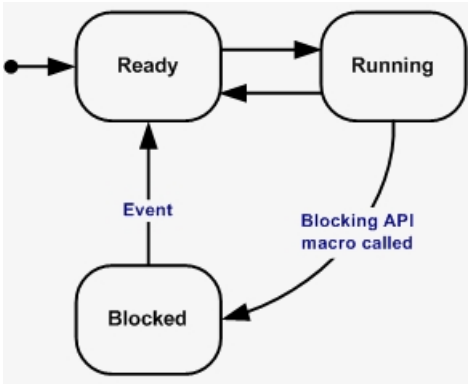


Figura 2.3: Grafo de estados de uma corotina

Assim como nas tarafes, a decisão de qual corotina irá entrar em execução também é feita pelo o escalonador, que será explicado na seção 2.1.3

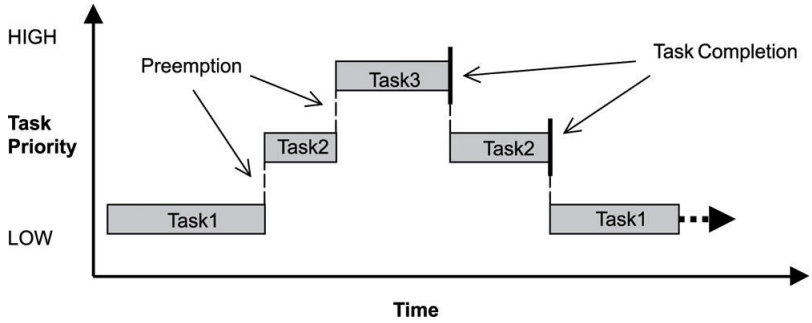
2.1.3 Escalonador de Tarefas

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 sitado apenas como unidade, refere-se a todas as tarefas e corotinas(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 exução 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 exução 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

Para as corotinas o escalonador funciona de forma cooperativa e baseada na prioridade. Com isso, a corotina em execução é quem decide o momento de sua interrupção, sendo que

a próxima corotina a entrar em execução será a de maior prioridade entre as corotinas com o estado pronto

2.1.4 Bibliotecas

Para disponibilizar as caraterísticas discutidas nesta seção o FreeRTOS disponhe das seguintes bibliotecas: Criação de Tarefas, Controle de Tarefas, Utilidades de Tarefas, Controle do Kernel e Corotinas. A seguir tem-se em detalhe a descrição de cada uma dessas bibliotecas junto com as funcionalidades que as copõem

Criação de Tarefas

Essa biblioteca é responsável pelo conceito de tarefa. Nela estão presente, um tipo, responsável por representar uma tarefa do sistema, e duas funcionalidades, uma para a criação de uma tarefa e outra para a remoção de uma tarefa do sistema. Em seguida, tem-se a lista de todos os tipos e funcionalidades disponibilizados por essa biblioteca.

- **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 indicar ao sistema que uma tarefa deve ser removida[3]

Controle de tarefas

A biblioteca de controle de tarefas realiza operações sobre as tarefas do sistema. Ela disponiliza funcionalidades capazes de bloquear, suspender e retornar uma tarefa do estado suspenso.Além dessas ela também possui funcionalidades capazes de alterar e informar a prioridade de uma determinada tarefa. A lista das principais funcionalidades presentes nessa biblioteca pode ser vista a seguir:

- **vTaskDelay** - Método usada para suspender uma tarefa por um determinado tempo. Nesse método, para calcular qual será o tempo que a tarefa deve retornar, é levando em

[3] A remoção de uma tarefa do sistema só é feita pela tarefa ocisa 2.1.1, nesse método é apenas indicado para o sistema quais as tarefas que devem ser removidas

consideração o tempo relativo, ou seja, o tempo que o método foi chamado. E por isso, é uma método não indicado para a criação de tarefas cíclicas, pois o tempo que o método é chamado pode variar em cada execução da tarefa devido as interrupções que a mesma pode sofrer.

- **vTaskDelayUntil** - Método usado para suspender uma tarefa por um determinado tempo. Esse método difere do *vTaskDelay* pelo o fato de que tempo em que a tarefa deve retornar é calculado com base no tempo do último retorno da tarefa. Assim, se ocorrer uma interrupção o tempo que a tarefa foi retornada não ira mudar, o que torna esse método recomendável para 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

Utilitários de tarefas

É atravé dessa biblioteca que o FreeRTOS diponibiliza para o usuário informações importantes a respeito das tarefas e do escalonador do sistema . Nela estão prensentes funcionalidades com a 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 listagem 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 do sistema. Nela encontramos funcionalidades que iniciam e finalizam as atividades do escalonador, e também, suspendem e retornam a atividades do mesmo. 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

2.2 Comunicação e sicronização entre tarefa

Frequentemente tarefas necessitam se comunicar um com as outras. Por exemplo a tarefa "A" depende da leitura do teclado feito pela tarefa "B". Com isso, a uma necessidade de que está comunição sejá feita de maneira bem estruturada e sem interrupções.

A isso a maiorida dos sistemas operacionais oferencem vários tipos de comunicação entre as tarefas. Que podem ocorrer da seguinte forma: uma tarefa deseja passar informações para outra, duas ou mais tarefas querem utilizar o mesmo recurso e uma tarefa dependo do resultado produzido por outra tarefa.

No FreeRTOS assim como nos demais sistemas operacionais os mecanismos responsáveis pela a comunicação entre as tarefas são a fila de mensagem, o semáforo e o mutex (*Mutal Excluision*). Para entender melhor como funciona a comunicação entre tarefas no FreeRTOS, cada um desses mecanismo serão detalhados a seguir

2.2.1 Fila de Mensagens

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

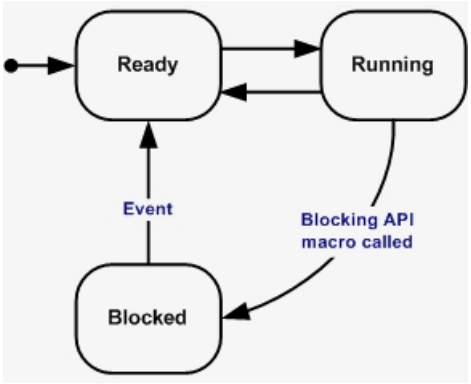


Figura 2.5: Funcionamento de uma fila de mensagens

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 dela é armazenada na lista de mensagens para que outra tarefa possa utiliza-lá. Entranto, no lugar de copiar toda mensagem para a lista de mensagens, é possível também armazenar-se apenas uma referência para a mesma, o que torna mais complicado um trabalho do desenvolvedor, pois assim o acesso à mensagem ficará compartilhado entre as tarefas, necessitando com isso de uma estrutura de sicronização para coordenar a utilização da mensagem pelas tarefas. Na maioria da aplicação a troca de mensagens é feita por cópia.

Além da lista de mensagens uma fila de mensagens é composta por mais duas outras 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 aguardam para enviar uma mensagem para fila até que um lugar na fila sejá liberado. O mesmo acontece quando uma tarefa tenta ler uma mensagem de uma fila vazia, sendo que nesse caso, ela 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

Semáforos são mecanismos usados na sincronização entre tarefas. Eles funcionam como uma chave que libera, ou não, o uso de um determinado recurso comparilhado. Assim, quando

uma tarefa deseja acessar um recurso compartilhado, ela primeiramente deve solicitar o semáforo que coordena o uso do recurso, caso o semáforo esteja livre, a tarefa tem a permissão de utilizar o recurso e, em seguida, libera o semáforo, caso contrário, a tarefa é bloqueada até que o semáforo seja liberado.

O FreeRTOS disponibiliza dois tipo de semáforos: os semáforo binário e o semáforo com contador. A diferença entre os dois está apenas no número de tarefas que podem reter o semáforo ao mesmo tempo. No semáforo binaário apenas uma tarefa pode reter o semáforo e acessar o recurso compartilhado. E no semáforo com contador existe um número fixo de tarefa, determinado por um contador, que podem reter o semáforo. Assim, no semáforo com contador, para cada tarefa que retêm o semáforo, o contador é decrementado de "um", e para cada tarefa que libera o semáforo, o contador é incrementado de "um". O semáforo torna-se indisponível quando o contador for igual a zero e disponível caso contrário.

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 parecidos 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 uma tarefa de menor prioridade, causando uma inversão de prioridade.

O mecanismo de herança de prioridade funciona da seguinte forma, quando uma tarefa solicita o semáforo ele 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 mais rapidamente o semáforo.

2.2.4 Bibliotecas

A característica de comunicação e sicronização entre tarefas está dividida em duas bibliotecas, Gerenciamento de fila de mensagens e Semáforo/Mutex. A seguir tem-se a 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 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 desejada. Abaixo tem-se uma lista com as principais funcionalidades dessas biblioteca.

- **xQueueCreate** - Cria uma nova instância de uma fila de mensagens.
- **vQueueDelete** - Remove uma fila de mensagem do sistema.
- **xQueueSend** - Envia um mensagem para a fila.
- **xQueueSendToBack** - Envia 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 são implementadas as características de sicronização entre tarefas (semáforo e mutex) junto com as suas operações. Assim nesta biblioteca estão presentes funcionalidades que criam e removem semáforos e mutex, além das funcionalidades que solicitam e liberam os semáforos e os mutex. As principais funcionalidades desta biblioteca pode ser vista a seguir.

- **vSemaphoreCreateBinary** - Cria um semáforo binário
- **vSemaphoreCreateCounting** - Cria um semáforo com contador
- **xSemaphoreCreateMutex** - Cria um mutex
- **xSemaphoreTake** - Solicita 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 FreeRTOS. Assim, com o intuido 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 o FreeRTOS suporta.

Entretanto, a criação e analise de uma nova aplicação no FreeRTO 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 FreRTOS, demonstrando a suas principais funcionalidades. Com isso, para efeito de exemplificação de como são utilizadas as funcionalidades apresentadas neste capítulo, será demonstrado de forma abstrata nessa seção com é criada uma aplicação no FreeRTOS, abordando pricipalmente 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 de uma aplicação. No FreeRTOS a rotina que compõem uma tarefa devem seguir a estrutura demonstrada pela figura 2.6. Nele tem-se inicialmente o nome da rotina, *functionName*, seguido da lista de parâmentros utilizados 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 excluida 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-se da funcionalidade *vTaskDelayUntil* (seção 2.1.4) para

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

bloquear a execução da tarefa em intervalos cíclicos de tempo. Essa funcionalidade possui como parâmetros, respectivamente, o último tempo que a tarefa foi retornada e o periodo que a tarefa deve permancer bloqueada.

Entretando 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 feito 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 função

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

STACK_SIZE : Tamanho da pilha de execução da função especificado de acordo com o número de variável 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

```
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 ciclica

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.

2.3.2 Utilização da fila de mensagens

A utilização de uma fila de mensagens é resumidamente demonstra 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 fila de mensagem. 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 um fila de mensagens.

Estabelecida a fila de mensagens, é necessário agora, definir-se 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 criar 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 parâmentros a fila para qual a mensagem será enviada, a mensagem que será enviada para a fila e tempo máximo que a tarefa poderá ficar bloqueada aguardando para enviar uma mensagem para fila, caso ela esteja cheia.

Por último, na figura 2.8 a rotina da tarefa responsável por receber mensagens da fila, é criada. Ela é denominda de *sendTask*. Para receber as mensagens enviadas para a fila, *sendTask* utiliza a operação *xQueueReceive*, a qual possiu 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 sicronização entre tarefas. Eles são utilizados para coordenar o uso de recurso compartilhado por uma ou mais tarefas. Assim, código da tarefa responsável por acessar o recurso compartilhado deve ser protegido para que a sua execução só ocorra quando o semáforo for retido pela tarefa.

A construção de uma aplicação que utiliza-se do mecanismo de semáforo, são necessário

```
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

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.

O exemplo de uma aplicação que utiliza-se de um semáforo para controlar o acesso a um recurso compartilhado pode ser visto na figura 2.9. Nela inicialmente é criada a variável *xSe-*

maphore 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.

```
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

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 rete-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 mecanismo está apenas no método de criação, *vSemaphoreCreateBinary* no semáforo binário, que será *xSemaphoreCreateMutex*. As formas e os métodos para solicitação do mutex é semelhante ao semáforo binário, sendo *xSemaphoreTake* informando o mutex e o tempo máximo de bloqueio e *xSemaphoreGive* informando o mutex que será liberado. Com isso para transformar a aplicação da figura 2.9 de semáforo pra mutex basta apenas trocar o

método *vSemaphoreCreateBinary* por *xSemaphoreCreateMutex*.

3 *Método B*

Métodos Formais trata-se de uma abordagem formal 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 aeronaltica, viação ferrea e de equipamentos médicos, além de empresas que movimentam grandes quantidade monetárias como os bancos.

Segundo a Honeywell [**?**],uma empresa que desenvolve sistemas para aeronaves, a atulização de métodos formais no processo de desenvolvimento prover várias vantagens, entre eles estão: a produção mensurada pela corretude, métodos formais prove 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; e garantia da corretude, através mecânismo de verificação é possível provar que sistema funcionará de forma coerente com a sua especificação inicial.

O método B[**?**] trata-se de uma abordagem formal usado para especificar e construir sistema computacionais seguros. Seu criado Jean-Raymond Abrial, junto com a colaboração de outros pesquisadores da universidade de Oxford, procuraram reunir no método B várias qualidades presentes nos demais método formais como pré e pós condições[1], refinamento[2] e modularização. A seguir tem-se a etapas de desenvolvimento de sistemas utilizando o método B.

3.1 Etapas do desenvolvimento em B

O processo de desenvolvimento através do método B inicia-se com um módulo que define um modelo funcional de alto nível do sistema. Em B, esses módulos são denominados de Máquina Abstrata (*MACHINE*). Nessa fase de modelagem técnicas sémi-formais como UML

[1] Pré e Pós condições são...

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

podem ser utilizadas e em seguidas transfomadas para a notação formal do método B. Após a criação dos módulos, eles são analisados estaticamentes para verificar se a sua especificação é coerênte e implementável.

Uma vez estabelicido um modelo abstrato incial do sistema, o método B permite que seja construidos módulos mais concretos do sistema denominados refinamentos. Mais especificamente refinamentos tratam-se de uma decisão de projetos quas partes da especificação abstrata do sistema devem ser especificadas em um nível mais concreto. Assim um refinamento deve necessáriamente estar relacionado com o módulo mais abstrato ligeiramente anterior. E, como ocorre na criação da máquinas abstratas, um refinamento também e passível 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. 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 para a especificação, B0, trata-se de um formalismo algorítmico, capaz de ser sintetizados para linguagens de programação com C, Java e JavaCard.

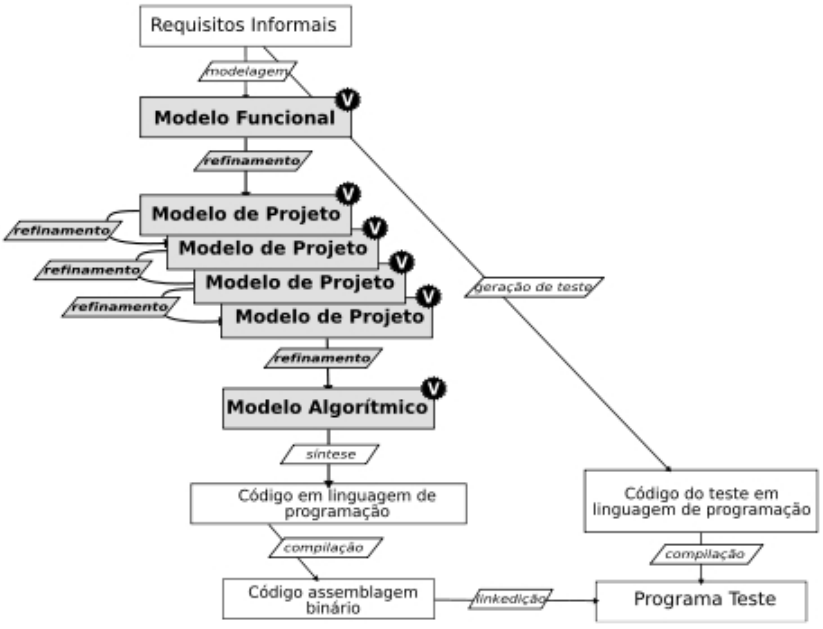


Figura 3.1: Etapas de desenvolvimento de sistema atravé do método B

Assim o desenvolvimento de sistemas utilizando o método B é feito como demonstra a figura 3.1. Nela os requisitos do sistema são especificados inicialmente em um alto nível abstrato e após sucessivos refinamentos um nível concreto e algoritmico do sistema é alcançado. Em seguida é feita a sintese da especificação para o código de linguagem de programação, sendo

possível apartir da especificação funcional inicial gerar teste para o sistema desenvolvido.

Atualmente o desenvolvimento de sistemas utilizando o método B pode ser apoiado por diversas ferramentas que vão desde a análise estática da especificação até a geração de código em linguagens de programação. Fator que isentivou, o método B a utrapassar o mundo acadêmica e passar a ser usado também industrialemente.

Um exemplo de sistemas desenvolvidos através da abrogagem de B é o controle de porta do metor de paris, desenvolvido pela Clearsy[**?**], empresa especilista em sistemas críticos. Esse sistema impede que o trem saida da estação quando a porta inda estiver aberta, evitando assim possíveis acidendes

3.2 Máquina Abstrata

A base do método B está na notação de máquina abstrada(em inglês: *Abstract Machine Notation* - AMN) a qual disponibiliza um framework comum para a especificação e contrução e verificação estática de sistemaso. Em outra palavras, a AMN trata-se de uma linguagem de especificação de sistemas formada por moódulos básicos de construção chamados de Máquina Abstrata ou simplemente 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 Nessa seção inicia-se o código da máquina abstrata. Ela identifica a natureza e o nome do módulo, seguido opcionalmente por um ou mais parâmetros separados por vírgula e limitados por parenteses

SETS introduz um novo tipo de entidade, no exmplo é *TASK*. Nesse momento, nenhum detalhe é fornecido quanto à maneira como essa entidade será implementada.

VARIABLES informa o nome das diferentes variáveis que compoem o estado. No exemplo, apenas há uma variável de estado: *tasks*.

INVARIANT especifica o tipo das variáveis de estado assim também como os estados válidos do sistema. Aqui, *tasks* é um conjunto de até 10 elementos do tipo *TASK*. A caracteri-

zação 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, *tasks* é o conjunto vazio.

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 précondições, a quais são condições que devem ser satisfeitas para que a operaçõas 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 resulmidamente 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 sejá a sua parte dinâmica com a inicialização e operações. Essas duas partes serão melhores discutidas a seguir.

3.2.1 Especificação do estado da máquina

Na parte do estado de uma máquina são colocados os estado que a máquina pode assumir.Este é definido através de suas variáveis e do seu invariante. Nas variável são colocados os nomes dos estados das máquinas e no invariantes suas restrições, sendo que uma especificação só pode garantir o correto funcionamento da máquina quando esta encontra-se em um estado válido, nada podendo afirmar para os demias estados.

O estado de uma máquina é especifica, por meio de calculo de predicados, da teoria do conjunto e relações entre conjuntos, permitindo com isso a anásile estádo da máquina através de 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 codições declaradas no invariante dá máquina deve ser sempre satisfeita no final da operação, levando assim a máquina a um estado válido.

O cabeçalho de uma operação é composto por um nome, uma lista de parâmetro de entra e uma lista de parâmentro de saida [3], sendo que os parâmetro de entrada e os parâmetros de saida são argumentos opcionais. Assim o exemplo de um operação com parâmetros de entrada e saida 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 saida é *belong*.

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 operações sejá 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 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 pertece a máquina *Kernel*

No corpo da operação é especificado o seu comportamento. Nele os parametros saida devem

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

ser obrigatoriamente valorados e os estados da máquina são alterados ou consultados. Assim, para realizar as atualizações de estadas e definições de parâmetros de saida de modo formal, a notação de máquina abstrata possui um conjunto de atribuiçoes abstratas, denominadas substituições, que possuem regras de como tal atribuições é realizada, permitindo assim uma analise estática da operações em realação a consistência da máquina. A seguir são demonstradas algumas das principais substituições da AMN.

Substituição Simples

A substituição simples é definida da seguinte forma:

$$x := E$$

Nela x trata-se da variável da máquina ou parâmentro de saida, para o qual será atribuido o valor da expressão E. Mais precisamente uma substituição é interpreta da seguinte maneira:

$$[x := E]P$$

Assim tem-se que o predicado P deve ser mantido quando a variável x for substituida por E. Por exemplo na operação add_task a substuição $tasks := tasks \cup \{task\}$ pode ser vista como $[tasks := tasks \cup \{task\}]\,\mathbf{card}(tasks) < 10$, Na qual $\mathbf{card}(tasks) < 10$ é o predicado, no caso o invariante da máquina, que deve ser obdecido quando a substituição for realizada, ficando ao final $tasks \cup \{task\} < 10$.

Substituição Multipla

A substituição multipla trata-se de uma generalização da substituição simple. Ela permite que várias variáveis seja atribuidas simutaneamente. Assim uma substituição multipla utilizando duas variável tem a seguinte forma:

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

Na definição acima as variáveis x e y são atribuido os valores das exepressões E e F, respectivamente. Assim da mesma forma que a substituição simples, a multipla substituição é definida da seguinte maneira:

$$[x := E, y := F]P$$

Na qual P é o predicado que deve ser verdadeiro quando suas variáveis x e y forem substituidas por E e F, respectivamente. Por exemplo, uma máquina que possui o predicado $x < y$ e possui a seguinte substituição $x, y := x + 10, y + 5$ é redizida a seguinte forma $[x, y := x + 10, y + 5]x < y$, o que resulta em $x + 10 < y + 5$

Substituição Condicional

As substituições simples e multiplas permitem somente uma opção de especificação onde uma atribuições é sempre feitas de maneira uniforme sem opções e sem levar consideração os estados iniciais da operação. Entretanto, as linguagens de programação convencionais disponibilizam um tipo condicional de atribuição na qual é permitido caminhos diferente de acordo com expressões lógicas que utilizam o valores iniciais das variáveis do sistema. Esse tipo de atribuição é feita através da formação **IF THEN ELSE**.

Como nas liguangens 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, não qual uma expressão lógica é avaliada para saber qual caminha a estrutura deve seguir, e qual atribução deve ser realizada. A forma como é especificada uma substituição condicional na ANM pode ser visto a seguir:

$$\textbf{IF} \quad E \quad \textbf{THEN} \quad S \quad \textbf{ELSE} \quad T$$

Na especificação acima S e T são substituições de qualquer tipo e tem as suas execuções condicionadas pela exepressão lógica E, na qual pode conter variáveis da máquina como também os parâmetros de entrada da operação. Com isso, caso E sejá 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 intrepretação se é for verdadeiro a substituição S deve satisfazer o predicado P. E caso o valor de E seja negativo a substituição T deve satisfazer o 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 sejá afirmativo $ans := yes$ é executada e caso a expressão sejá negativa $ans := no$ é executada.

Substituição não deterministica ANY

Até agora foram vistas apenas substituições deterministicas, ou seja, substituições que possuem um comportamento previsível e que leva a apenas um resultado final é possível. Entretanto, as máquinas abstratas de B servem para fazer especificação abstrata de sistemas e componetes e as vezes em um nível alto de abstração o comportamento das operações do sistema podem não ser tão previsíveis como as substituições deterministicas exigem. Para resolver esse problema foram criadas as substituições não deterministicas

As substituições não deterministicas são substituições que introduzem escolhas aleatórias no corpo das operações, levando a mesma não mais para um estado final previsível e sim para um conjunto de estados finais possíveis. Assim em uma substituição não determinística a especificação define apenas em qual conjunto deve ser feita a escolha abstraindo informações de como tal escolha deve ser realizada.

Um exemplo de substituição não deterministica da AMN é a substituição **ANY**. Essa substituição possui o seguinte formato:

ANY x **WHERE** Q **THEN** T **END**

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

x trata-se de uma lista de variável que será utilizada no corpo da substituição e para as quais serão escolhidos valores abstratos delimitados pelo o predicado Q.

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

T é denominado corpo da substituição. Nele são colocados atribuições que utilizam-se das variáveis x para atualizar estados ou atribuir valores para os parâmetros de saida de uma operação

Um exemplo da substituição **ANY** pode ser visto na figura 3.4 na qual uma tarefa aleatória é adicionada na máquina *Kernel*. Nela primeiramente a variável *task* é criada para atribuir-se um valor aleatório. Em seguida, o tipo e a restrição sobre *task* é definida. Por ultimo a variável

task é adicionada ao conjunto *tasks*. Assim um comportamento não deterministico é atribudo a 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 ∈ *TASK* ∧
    task ∉ *tasks*
  THEN
    tasks := (*tasks*) ∩ *task*
  END
END

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

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

$$\forall x.(Q \Rightarrow [T]P)$$

Indicando que para todo valor que for escolhido para o conjunto de variável x as substituições T devem garantir o 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 impementação. Para realizar tal avaliação o método B dispoem de um conjunto de obrigações de prova, que são asserções criadas apartir da especificação de uma máquina abstrata, e que devem ser provadas para garantir a corretude da especificação.

Resulmidamente a análise estática de uma máquina abstrata através das obrigações de prova avalia se a máquina possui estados válidos, ou seja, se pelo menos uma combinação de estados válidos é alcançada pela máquina e, caso afirmativos, se esses estados são alcançados no final de cada operações. 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 detalhe cada uma dessa 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 em que todos os estados são válidos, ou seja, 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

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 provador como verdadeiro para $tasks = \emptyset$

3.3.2 Obrigação de prova da inicialização

Outra obrigação de prova necessária para uma máquina abstrata é saber se seu estado inicial satisfaz o invariante. Isso significa verificar se o estado inicial da máquina é um 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 os predicados definidos 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 visto como uma verdade.

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 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 desa definição serai que quando a máquina estiver em um estado válido e a pré-condição da operação for satisfieta, a execução da operação deve manter a máquina em um estado válido. Nota-se assim que essa obrigação de prova não torna-se necessário ser avaliada para as operações que não alteram o estado da máquina, chamada operções de consulta como a da figura 3.3, onde apenas o valor da parâmentro 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:

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

3.4 Refinamento

A linguagem abstrata demonstratada até agora é usada 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 as informações serão manipuladas pelo computador. Entretanto, para realizar uma modelagem mais concrete e passível de implementação é necessário que notações matemáticas abstratas utilizadas na modelagem abstratado sistema como conjutos e não determinismo sejam descritas de forma mais concreta. Assim um sistema pode ser desenvolvido gradativamente, nos quais os estágios intermidiários entre a modelagem abstrata e a implementação, são combinações de especificações da construção e detalhes de implementação denominado refinamento.

Refinamentos são decisos de projeto nas quais estruturas abstratas são detalhadas em um nível mais concreto. Com isso, um refinamento deve obrigatorialmente está ligado ao nível mais

abstrato realizando as mesmo funcionalidades que este, só que um nível menos abstrato. Para garantir que essa ligação sejá realizada de forma correte existem mecânismo de analise estática denominados obrigações de prova do refinamento

A construção de um refinamento é muito parecida com a construção de uma máquina abstrata. Ele, assim como a máquina abstrata, é dividos em seções onde são especificadas as informações do sistema, possuindo basicamente as mesma seção de uma máquina abstrata. A diferenteça está na seção **REFINEMENT** onde é coloca o nome do refinamento e na seção **REFINES** informa qual á máquina ou refinamento, será refinado. Além disso assim como a construção de uma máquina abstrata pode ser dividida em especificação do estado e especificação das operações, a construção um refinamento pode ser dividido refinamento dos estados e refinamentos das operações

3.4.1 Refinamento do Estado

No refinamento de dados, como é reconhecido o refinamento de estado, tem objetivo de especificar o estado dá máquina de uma maneira mais concreta. Assim as estruturas utilizadas na modelagem abstratas são especificadas de uma forma mais próxima utilizada pelo computador. Para isso na especificação do refinamento seus estados são criados de forma concreta e necessariamente esses devem possuir uma ligação com os estados da máquina abstrata chamada de *relação de refinamento*

Por exemplo, a máquina *mathitKernel* da figura 3.2 possui o estado *task* que um conjunto de elementos do tipo *TASK*. Um possível refinamento desse estado é demonstrado pelo refinamento *KernelR* demonstrado na figura 3.5. Nele um estado *task* é refinado pelo variável *taskR* que representa uma sequência de tarefas, estrutura mais concreta do que um conjunto, sendo que a relação entre os dois estados, *realação de refinamento* é especificada na preposição $\mathbf{ran}(tasks\_r) = tasks$.

REFINEMENT *Kernel_r*
REFINES *Kernel*
VARIABLES *tasks_r*

INVARIANT $tasks\_r \in \mathbf{seq}(TASK) \wedge \mathbf{ran}(tasks\_r) = tasks$
INITIALISATION $tasks\_r := []$

OPERATIONS
$task\_add(task) =$
BEGIN
$tasks\_r := task \rightarrow tasks\_r$
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, podem até fazer menos ações que a abstrata, sendo proprido apenas as ações adicionais.

As operações de um refinamento devem possuir a mesma assinaturas das operações da máquina abstrata relacionada à ele, com os mesmo nomes e parâmetros de entrada e saida. 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 são suficientes para garantir que o tipo do parâmetro de entrada permaneça o mesmo e que o invariante da máquina seja mantindo quando a operação for executada.

Um exemplo do refinamento de uma operação pode ser visto na figura 3.5, na operação *add_task*. Nela percebe-se a ausência da pré condição e que assinutura da operação permanece a mesma. A parte alterada foi apenas o corpo da operação que agora foi adaptada para trabalhar com o estado *taskR*.

Um exemplo do refinamento da máquina *Kernel* (figura 3.2) pode ser visto na figura 3.5. Nela o conjunto de tarefas da máquina *Kernel* denominando *task* é refinado como sendo uma sequência de tarefas denominada *taskR*, o que é uma estrutura mais próxima da implementação do que um conjunto. Assim as operações que utilizam-se dessa estrutura também devem ser alteradas com o objetivo de mandar a compatibilidade da operações com a nova representação do estado da máquina.

3.4.3 Obrigação de Prova do refinamento

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

Obrigação de Prova da Inicialização

Em um refinamento, a ligação entre os estados da máquina abstrata e o refinamento é feito através do *relação de refinamento* a qual será denomida de J. Assim necessariamento a inicialização da máquina abstrata deve possuir algum estado que satisfaça essa ligação, ou melhor dizendo, essa J deve possuir algum estado especificado pela inicialização de máquina abstrata, denominada T. Isso gera a seguinte asserção:

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

O que define que T deve possuir algum estado válido em J, ou que, J possui algum estado que satisfaça a inicilização T.

Além disso é necessário também que a inicialização do refinamento denominado de *T1* esta liga a esse conjunto de estados válidos definidos por J e T. Essa ligação ocorre da seguinte forma, todo estado gerado na inicialização de *T1* deve possuir um correspondente válido de T. Isso gera a seguinte definição

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

Assim a obrigação de prova entre as máquina *Kernel* e a máquina *KernelR* fica como desmonstra a asserção abaixo:

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

Obrigação de prova da operação sem parâmetro de retorno

Geralmente uma operação é definada como **PRE** P **THEN** S **END** sendo o seu refinamento **PRE** $P1$ **THEN** $S1$ **END**, onde na maioria da vezes $P1$ é abstraído. Com isso, do mesmo modo que na inicialização tem-se que os estado gerados em $S1$ devem estar relacionado com alguma execução S, gerando:

$$S1\neg S\neg J$$

Entretando, diferente da inicialização a execução de uma operação deve levar em consideração o estado da máquina anterior à sua realização. Com isso o estado da máquina abstrata

junto com o seu refinamento devem ser um estado válido. Uma relevante ligação entre esse estados o invariante I e a sua relação de refinamento J. Além disso, para a correta execução da operação a pré condição da mesma deve ser estabelecido. Assim, unido que uma operação só pode ser executada corretamente quando à uma ligação válida entre os estado da máquina e a sua pré condição for estabelicida, 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 pode é estabelicida como de acordo com a preprosição abaixo:

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

Obrigação de prova da operação com parâmetro de saida

As operações com parêmetros de saida necessitam de uma atenção especial devido a obrigação de que cada saiada possível para o refinamento deve está ligada a uma saida da especificação abstrata. Assim denominando out' como sendo os parâmetros de saida do refinamento e out os parâmentros de saida da especificação diz-se que cada valor de out' deve possuir um correspondente em out. Em outra palavra, cada execução de $S1$ deve encontrar uma S, naqual out' produzido por $S1$ sejá igual ao out produzido por S.

Além da ligação entre o parâmetros de saída, a obrigação de uma operação com retorno também deve obedecer todas as retrições impostas para as obrigações de prova das operações sem parâmetro de saida, ficando da seguinte maneira:

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

Nela $[out' out]$ significa que na atribuições de $S1$ os valores de out devem ser substituidos

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