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.

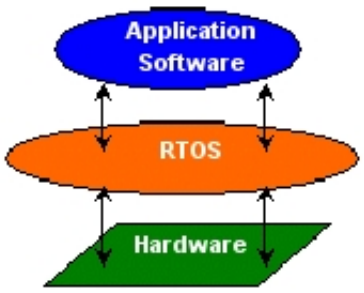


Figura 2.1: Camada abstrata proporcionada pelo FreeRTOS

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]

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 de controle** 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 **compilado** para arquitetura desejada.

A seguir serão detalhadas os principais serviços providos pelo o FreeRTOS junto com a biblioteca que disponibiliza tão serviço. Após isso será também demonstrado como é criada uma aplicação utilizando o FreeRTOS

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 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 o usuário
- Uma pilha naqual é armazenada o ambiente de execução (estado dos restradores) da tarefa quando está é interrompida

Os possível estados que uma tarefa pode assumir são :

- **Em execução**: Indica que a tarefa esta sendo executada 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 entre os estados 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 no mesmo momento. Assim no FreeRTOS apenas uma tarefa pode assumir o estado pronto em um determinado instante, restando as demais os outros estado. Com isso, para decidir qual tarefa deve ser executada existe um mecanismo no sistema operacional denominado escalonador, o qual será detalhado na sessão 2.1.2.

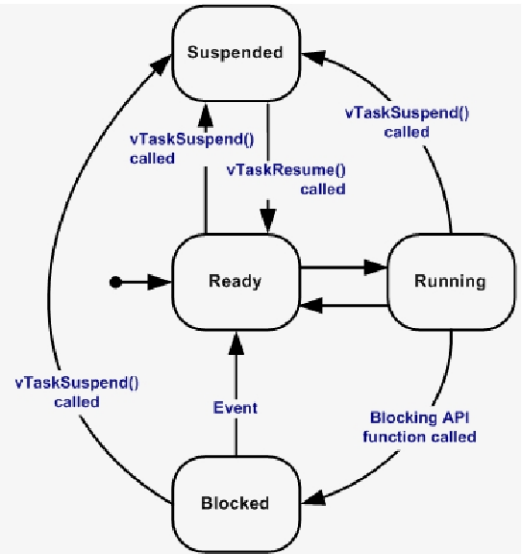


Figura 2.2: Grafo de estados de uma tarefa

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 essa 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 Escalonador de Tarefas

O escalonador é a parte mais importante de um sistema. É ele quem decide qual tarefa deve entrar em execução e realiza entre a tarefa que está no processador, ou seja em execução, com a nova tarefa que ira ocupar o processador, a tarefa que irá entrar em execução. No FreeRTOS o escalonador pode funcionar de três modos diferentes :

- **Preemptivo**: Quando o escalonador interrompe a tarefa em execução mudando o seu estado e ocupa o processador com outra tarefa
- **Cooperativo**: Quando o escalonador não tem permissões de interromper a tarefa em execução, tendo que esperar a mesma interromper a sua execução para que ele possa decidir qual será a próxima tarefa que irá entrar em execução e realizar a troca das mesmas.
- **Híbrido**: Quando o escalonador pode comporta-se tanto como preemptivo como cooperativo.

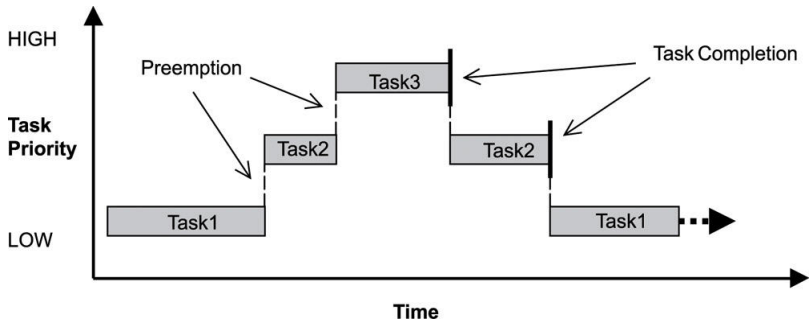
Para as tarefas o escalonador funciona de forma preemptiva, sendo que a decisão de qual tarefa deve entrar em execução e baseada na prioridade e segue a seguinte política: a tarefa em execução deve ter prioridade maior ou igual a tarefa de maior prioridade com o estado "pronta". Assim sempre que uma tarefa, com prioridade maior que a tarefa em execução, entrar no estado pronto, ele deve imediatamente entrar "em execução". Um exemplo claro da política preemptiva de prioridade discutida acima pode de visto na figura 2.3, naqual três tarefas, em ordem crescente de prioridade, disputam a execução do processador.

2.1.3 Corotinas

Outro conceito importante suportado pelo FreeRTOS é o de Corotina. 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 um estado, sendo a principal diferença entre uma corotina e uma tarefa a falta de uma pilha para armazenar o contexto de execução, a qual está presente nas tarefas e nas corotinas não.

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



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 2.3: Funcionamento de um escalonador preemptivo baseado na prioridade

- **Bloqueada**: Quando a corotina esta bloqueada esperando por algum evento para continuar a sua execução.

As transições entre os estados de uma corotina ocorre como demonstra a figura 2.4. 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”.

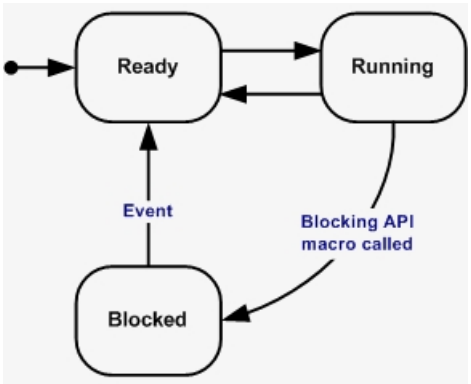


Figura 2.4: Grafo de estados de uma corotina

Assim como nas tarafes, a decisão de qual corotina irá entrar em execução é feita pelo o escalonador. 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 cada uma delas disponibiliza

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 uma lista com todas 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* ela é retornada pela método através do tipo *xTaskHandle*
- **xTaskCreate** - Funcionalidade usada para criar uma nova tarefa para o sistema.
- **vTaskDelete** - Funcionalidade usada para indicar que uma tarefa deve ser removida do sistema[2]

Controle de tarefas

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

[2]A verdadeira remoção de uma tarefa do sistema só é feita pela tarefa ocisa 2.1.1, nesse método é apenas indicado para o sistema qual tarefa deve ser removida

- **vTaskDelay** - Método usada para suspender uma tarefa por um determinado tempo. Nesse método, para calcular quando será o tem que a tarefa deve acordar, é levando em 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 *vTaskDelayUntil* pelo o qual do tempo em que a tarefa deve ser retornada, pois nesse é levado em consideração o tempo em que uma tarefa foi retornada. Assim, se ocorrer uma interrupção o tempo que a tarefa foi retornada não ira mudar, podendo criar assim uma tarefa com intervalos iguais de execução chamada de tarefa cíclica

- **uxTaskPriorityGet** - Método usado para informar 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 utual 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 sistema

- **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 responsáveis por inicializar e finalizar as atividades do escalonador, assim como, suspender e retornar 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 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. Devido 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 na maioria dos 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 será detalhado 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, quando entrar em execução possa ler a mensagem enviada.

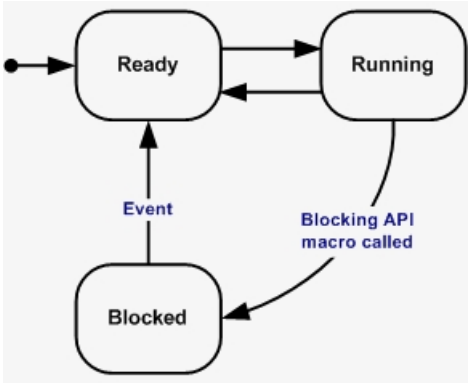


Figura 2.5: Funcionamento de uma fila de mensagens

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 dela é armazenada na lista de mensagens para que outra tarefa possa utiliza-la. 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.

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 esta 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.

O FreeRTOS, disponibiliza para o usuário funcionalidades é possíveis de definir o tempo máximo que uma tarefa pode ficar bloqueada esperado por uma fila (liberação de espaço ou chega de mensagem). E no caso em que existem 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. 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 liberado, 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 de contador. A diferença entre os dois está apenas no número de tarefas que podem reter o semáforo ao mesmo tempo. No semáforo binario apenas uma tarefa pode reter o semáforo e acessar o recurso compartilhado. E no semáforo com contador existe um número fixo de tarefa que podem reter o semáforo, sendo esse número definido na criação do semáfor e controlado pelo contador. Assim no semáforo com contador para cada tarefa que retem o semáforo o contador é decrementado de um e para cada tarefa que libera o semáforo o contador é incrementado, sendo que o semáforo torna-se indisponível quando o contador for igual a zero.

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.

2.2.3 Mutex

Mutex são parecidos com o semáforo 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 o que chamamos de 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 fila de mensagens do sistema assim como também 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 instância de uma nova fila de mensagens no sistema
- **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 inicio da fila
- **xQueueReceive** - Ler e remove uma mensagem da fila
- **xQueuePeek** - Apenas ler uma mensagem da fila

Semáforo/Mutex

Na biblioteca de semáforo e mutex são implementadas as características de sicronização entre tarefas, ou seja, os mecanísmo de semáforo e mutex junto com as suas operações. Assim nessa 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 dessa 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 para a aplicação desenvolvida possa funcionar coretamente. Entre essas restrições estão alguns parâmetros de configurações contento informações sobre o hardware, como tamanho da memória e velocidade clock, assim como restrições de templates para a criação de determinadas estruturas utilizadas pela a aplicação, como tarefas e fila de mensagem. Entretanto, para facilitar o trabalho e o entendimento do desenvolvedores iniciantes o FreeRTOS disponibilizou junto com seu código fonte várias aplicações exemplos que devem ser tomadas como base para o desenvolvimento de novas aplicações. Nelas são encontrados vários tipos de configurações diferentes para cada arquitetura alvo suportada, assim como vários padrões que tornam o desenvolvimento de aplicações através do FreeRTOS uma tarefa mais simples.

Contudo explicar detalhadamente como é desenvolvida um novo sistema utilizando o FreeRTOS foge do escopo desse capítulo, que tem como objetivo realizar uma pequena introdução ao Sistema Operacional de Tempo Real. Assim, com o objetivo finalizar uma introdução a explicação sobre o FreeRTOS e demonstrar como as bibliotecas do sistema são utilizadas para a criação de uma nova aplicação, nessas seção será demonstrada uma explicação didática de como é criada uma aplicação no FreeRTOS.

Para criar-se uma aplicação no FreeRTOS deve-se inicialmente defenir e implementar as tarefas que são execultadas pela a aplicação.

Inicialmente a construção de novas aplicações utilizando o FreeRTOS é feita através de modificações em exemplos disponibilizados junto com o seu código fonte. Assim o desenvolvedor

Com dito no inicio do capítulo, o FreeRTOS é um simples e portável, sendo o mesmo compatível com várias arquiteturas diferentes. Ele disponibiliza, junto com seu código fonte, exemplos de aplicações para vários tipos de arquiteturas diferentes. Inicialmente essas aplicações servem como base para o desenvolvimento de novos sistemas, pois vários detalhes como configuração do hardware e utilização ou não de certo recurso pelo o sistema já vem pre configurados nesses exemplos, sendo necessário o desenvolvedor apenas

Entretanto,

demonstrar detalhamente como é construida uma aplicação no FreeRTOS foge do escopo desse trabalho. Desse

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 a sua rigosa construção os métodos formais tem sido bastante utilizados na criação de sistemas críticos como industria aeronaltica, programas médicos e programas lidam enormes valores monetários.

O método B trata-se de uma abordagem formal usado para especificar e construir sistema computacionais seguros. Ele foi criado por Jean-Raymond Abrial, com a colaboração de outros pesquisadores da universidade de Oxford. Na sua criação foram reunidaes várias qualidades presentes nos demais método formais. Entre elas estão as pré e pós condições, condições necessárias para a execução de um método e alcançadas após a execução do mesmo, mudularização, abstração e refinamento, estratégia de construção/especificação de sistemas através de vários níveis de abstração.

o método B proporciona uma criação de sistemas através de sucessiveis níveis de abstração, na qual inicialmente cria-se um modulo abstrato em uma linguagem de modelagem. Esse módulo é refinado através de vários outros módulos até chegar em uma linguagem algorítmica, denominada B0, que pode ser traduzida automaticamente em algumas linguagem de programação imperativa com C, Ada, Java, JavaCard e C#.

Cada módulo criado no desenvolvimento do sistema com o método B deve ser analisado estaticamente para saber se ele é implementável ou consistente, ou seja, que sua execução não leve a um estado não permitido pela especificação. Assim como também cada nível de abstração deve ser analisado estaticamente para saber se ele é coerente com o nível acima.

Atualmente o desenvolvimento de sistemas utilizando o método B pode ser apoiado por diversas ferramentas que vão desde a analise estática da especificação até a geração de código executável. Devido a isso, o método B utrapassou a barreira acadêmica e passou a ser bastante difundido na indústria de sistemas críticos, principalmente na industrias ferroviárias e auto-

mobilistica, sendo utilizado em sistemas que atuam no metro de Paris e em subsistemas dos automóveis da Peugeot.

3.1 Notação de 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 de sistemas, permitindo também a verificação estática do mesmo. Mais especificamente, a AMN trata-se de uma liguagem 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. Como, por exemplo, a figura 3.1 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 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, *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 3.1: Maquina abstrata de tarefas

Com isso a especificação de sistemas utilizando o método B pode ser dividido em duas partes principais, a especificação do estado da máquina e a especificação das operações da máquina. Essas duas partes serão discutidas a seguir.

3.1.1 Especificação do estado dá máquina

O estado de uma máquina abstrata pode ser definido em termos de suas variáveis e do seu invariante. Assim o estado de uma máquina é formado pelas as variáveis da máquina junto com suas definições e limitações. É no estado que é defino os estado válidos do sistema.

No estado da máquina são especificados, por meio de calculo de predicados, da teoria do conjunto e relações, as propriedades estáticas que o sistema deve obedecer. Assim no exemplo da figura 3.1 o estado foi especificado como sendo a variável *tasks* e seu predicados $tasks \in \mathbb{P}(TASK)$ e $\mathbf{card}(tasks) \leq 10$

3.1.2 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 restrições do estado da máquina, ou seja, as codições declaradas no invariante dá máquina deve ser sempre satisfeita no final da operação.

O cabeçalho de uma operação é composto por um nome, uma lista de parâmetro de entra

e uma lista de parâmentro de saida [1], sendo que os parâmetro de entrada e os parâmetros são argumentos opcionais. Assim o exemplo mais completo de uma operação pode ser visto na figura 3.2, a 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 da operação 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 satisfeita para que a operação funcione corretamente.

Por exemplo, na figura 3.1 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 3.2: Operação que consulta se uma tarefa pertece a máquina *Kernel*

No corpo da operação é especificado o seu comportamento. Nele os parametros de saida são obrigatoriamente valorados e os estados da máquina são alterado ou consultados. Assim, para realizar essas atualizações 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, o que permite 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:

$$x := E$$

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

Nela x trata-se a variável 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:

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

Onde paras variáveis x e y são atribuido os valores das exepressõe 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 um opção de especificação onde uma atribuições é sempre feitas de maneira uniforme sem opções e sem levar consideração os estados iniciais na 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**.

Assim como nas liguangens de programação, a notação de máquina abstrata também per-

mite a construção de atribuições condição, as quais são feitas através da substituição condicional. Com isso, uma substituição condicional funciona da mesma forma que as 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 tem suas execuções condicionadas pela exepressão lógica é 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 a substituição condicional pode ser interpretada da seguinte forma:

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

.

Um exemplo simples da utilização dessa substituição pode ser visto na figura 3.2. 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 executado e caso a expressão sejá negativa $ans := no$ é executado.

Substituição não deterministica ANY

Até agora foram vista apenas substituições deterministacas, ou sejá elas possuem um comportamento previsível 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 certo nível 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 tratam-se de 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 para uma determinada escolha a especificação apenas define em qual conjunto deve ser feita a escolha e não diz nenhuma informação de como tal escolha deve ser feita.

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 assim 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 3.3 naqual uma tarefa aleatória a adicionada na máquina *Kernel*. Nela primeiramente a variável *task* é criada para qual será atribuido um valor aleatório. Em seguida, tipo e uma 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.

$$
\begin{array}{l}
random\_create = \\
\textbf{PRE} \\
\quad \textbf{card}(tasks) < 9 \\
\textbf{THEN} \\
\quad \textbf{ANY} \\
\quad\quad task \\
\quad \textbf{WHERE} \\
\quad\quad task \in TASK \wedge \\
\quad\quad task \notin tasks \\
\quad \textbf{THEN} \\
\quad\quad tasks := (tasks) \cap task \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figura 3.3: 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.

3.2 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 análise 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 provadas afim a corretude da especificação.

Resulmidamente a analise estática de uma máquina abstrata através das obrigações de prova avalia se essa possui estados válidos, ou seja, se pelo menos uma combinação de estados válidos é alcançada pela máquina e, caso afirmativos, se esses são mantidos pelos invariantes. Com isso as principais obrigações de prova gerada em uma máquina abstrata saõ: Consistência do Invariante, Obrigação de Prova da Inicialização e Obrigação de Prova das Operações. A seguir é detalhado o que significa cada uma dessas obrigações de prova e como elas são realizadas.

3.2.1 Consistência do Invariante

Nessa obrigação de prova é analisado se o invariante da máquina possui pelo menos uma combinação naqual todos os estados são válidos. 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 3.1 seria:

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

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

3.2.2 Obrigação de prova da inicialização

Outro obrigação de prova necessária para uma máquina abstrata é saber se seu estado inicial satisfaz o invariante. Em outras palavras 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. Assim a obrigação de prova da inicialização da máquina da figura 3.1 pode ser definido 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.

3.2.3 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 3.2, 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:

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

3.3 Modularização

3.4 Refinamento

- Explicar o que é o método B
- Explicar a base teórica de B (AMN e as substituições)
- Explicar como é especificado um sistema em B (como é criado um módulo)
- Falar das obrigações de prova
- Falar dos mecânismo de composição e refinamento
- Dizer que o refinamento pode chegar em um nível concreto que pode ser sintetizado para algumas linguagens de programação.
- Falar do uso de ferramentas
- Falar do projeto B2ASM

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