Stephenson de S. L. Galvão

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

Natal - RN, Brasil

2 de dezembro de 2009

Stephenson de S. L. Galvão

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

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

Orientador:
Prof. Dr. David Déharbe

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

Natal - RN, Brasil

2 de dezembro de 2009

Resumo

Este trabalho apresenta uma contribuição para o esforço internacional do *Verified Software Repository*, o qual tem como alguns de seus objetivos acelerar o desenvolvimento das tecnologias de verificação de software e colecionar especificações de diversas aplicações como *smart card*, bibliotecas padrões de desenvolvimento e sistema operacionais embarcados. Para essa contribuição, será adotada a sugestão do pesquisador britânico Woodcock de especificar formalmente o sistema operacional de tempo real FreeRTOS, o qual tem como características a sua simplicidade, portabilidade e popularidade. Com essa especificação será possível verificar o correto funcionamento desse sistema e, conseqüentemente, garantir uma maior confiabilidade para as aplicações desenvolvidas através dele, cooperando, com isso, também para o desafio da construção de sistemas disponíveis, seguros, escalonáveis e ubíquos, proposto pela Sociedade Brasileira de Computação. O formalismo que será utilizado nessa especificação é o método B, que, devido as suas semelhanças com as linguagens de programação imperativas e suas técnicas de modularização e refinamento, tornou-se viável para a construção dessa especificação. O modelo desenvolvido terá como principal foco as funcionalidades e abstrações de hardware do sistema e o seu desenvolvimento será feito de forma incremental. Para provar a viabilidade desse trabalho, uma especificação abstrata foi inicialmente desenvolvida, na qual algumas propriedades do sistema puderam ser tratadas. Ao final, pretende-se com a especificação do FreeRTOS, além de validar os requisitos desse sistema, criar uma documentação formal do mesmo, a qual poderá ser usada como entrada para a geração de testes, no nível de código, para o sistema especificado. Em suma, a especificação do FreeRTOS auxiliará na resolução dos desafios anteriormente citados, servindo desse modo como uma contribuição para a sociedade de computação, em especial a sociedade de métodos formais.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A Ciência da Computação é uma área relativamente jovem, mas com grande impacto na sociedade atual. Através dela, é possível acelerar-se anos de desenvolvimento e pesquisas das demais áreas da ciência, como ocorreu com o mapeamento do genoma humano. Devido a isso, em [5] ela chegou a ser citada como um dos pilares da ciência, considerada também como um fator crucial para a economia, tecnologia e desenvolvimento de um pais.

Ciente dessa importância, pesquisadores de todo mundo tem buscado reunir esforços em torno de objetivos comuns, definindo diretrizes a serem seguidas pela computação ao longo dos próximos anos. Essas diretrizes são estabelecidas visando suprir as necessidades mais relevantes da sociedade atual, as quais, em geral, são obstáculos ainda não vencidos pela computação, e, por isso, denominados como “Grandes Desafios da Computação”.

Seguindo esse contexto, a Sociedade Brasileira de Computação (SBC) listou, em [6], alguns desses principais desafios, dentre os quais está o desafio de *desenvolvimento tecnológico de qualidade: sistemas disponíveis, corretos, seguros, escaláveis, persistentes e ubíquos*. Esse desafio visa a construção de sistemas disponíveis, sem falhas, previsíveis, escaláveis e seguros. Qualidades que, devido à crescente utilização dos sistemas computacionais, tornaram-se cada vez mais necessárias.

Uma das inspirações para o desafio acima foram os sistema críticos. Nesses sistemas, é de suma importância o seu correto funcionamento, pois a sua principal característica é a criticidade das suas operações, nas quais uma falha pode ter conseqüências catastróficas. E, devido a isso, a grande necessidade do desenvolvimento de técnicas para a construção de sistemas fidedignos.

Uma abordagem utilizada com sucesso para a criação de sistemas “corretos” tem sido os métodos formais. Métodos formais são técnicas matemáticas usadas para desenvolver sistema de *software* e *hardware*. O rigor matemático dessas técnicas permite ao usuário analisar e verificar seus modelos em todas as partes do seu ciclo de vida: requisitos, especificação, modelagem, implementação, teste e manutenção. Atualmente, esses métodos têm sido utilizados, de forma auspiciosa, pela indústria para a construção de pequenas, mas significantes aplicações [7].

Entretanto, a comunidade de métodos formais também possui seus desafios, entre eles está o projeto do "Software Verificado"[8]. O Software Verificado é um conjunto de teorias, ferramentas e experiências usadas para verificação e validação de softwares. Ao final desse projeto, pretende-se criar um verificador de softwares que garanta o comportamento fidedigno do sistema analisado. Outra atividade associada a esse projeto é a criação de estudos de caso a serem utilizados pelas abordagens proposta. Entre tais estudos de caso, um dos mais interessantes e proveitosos é sem dúvida a especificação formal do sistema operacional de tempo real FreeRTOS [9][7].

O FreeRTOS [2] é um estratégico e importante estudo de caso. Isso, devido a sua larga utilização, simplicidade e portabilidades. Atualmente, o seu repositório possui uma taxa maior que 6.000 downloads por mês; o seu núcleo é acessível, aberto e pequeno, são aproximadamente 2.200 linhas de código, com funções comuns a maioria dos sistemas modernos e suportadas oficialmente por 17 arquiteturas de microcontroladores diferentes. Devido a essas qualidades, a especificação desse sistema seria uma grande contribuição, principalmente para o desenvolvimento da computação, em especial a comunidade de métodos formais.

Entre as possíveis abordagens formais para a especificação do FreeRTOS está o método B [10][11]. O método B é uma abordagem de especificação, validação e construção de sistemas. Através dele é possível modularizar e refinar uma especificação até um nível algoritmo, passível de transformação para as linguagens de programação imperativas. Além disso, ele é apoiado por um pacote de ferramentas que suportam todos os seus ciclos de desenvolvimento, o que tornou essa abordagem uma forte candidata para a especificação do FreeRTOS.

Em suma, especificação formal do FreeRTOS através do método B, além ser um esforço para a resolução dos desafios apresentados, é uma grande contribuição para a computação, principalmente à comunidade de métodos formais. Tal especificação poderá ainda agregar valor ao sistema, servindo como documentação e entrada para técnicas de teste de sistemas construídos utilizando o Sistema Operacional em questão [12].

Prosseguindo nesta introdução, tem-se, na seção 1.1, o objetivo a ser atingido com esse trabalho, na seção 1.2, a metodologia para atingir tal objetivo, e, na seção 1.3, os trabalhos relacionados a essa proposta. Na continuação do trabalho, as fundamentações teóricas referentes ao FreeRTOS e ao Método B são apresentada, respectivamente, nos capítulos 2 e 3. Em seguida, a parte do trabalho já realizada é demonstrada no capítulo 4 e a proposta, junto com as atividades e o cronograma de trabalho, são exibidos no capítulo 5. Por último, tem-se a conclusão no capítulo 6.

1.1 Objetivo

O objetivo desse trabalho é, através da especificação do FreeRTOS em B, contribuir para os desafios do software verificado e da construção de software fidedigno. A especificação desenvolvida modelará funcionalmente as funcionalidades que compõem o FreeRTOS, sendo a mesma capaz de analisar propriedades estáticas do FreeRTOS e validar algumas de seus principais requisitos.

1.2 Metodologia

A metodologia de desenvolvimento desse trabalho pode ser sucintamente resumida pelos itens abaixo, os quais serão comentados nos parágrafos seguintes:

- Estudo do método B;
- Levantamento bibliográfico sobre FreeRTOS;
- Estudo do código fonte do FreeRTOS;
- Elaboração de um plano para a modelagem funcional do FreeRTOS;
- Modelagem funcional do FreeRTOS considerando apenas as entidades Tarefa e Fila de Mensagens
- Refinamento da modelagem desenvolvida para adicionar a característica de prioridade;
- Refinamento da modelagem para adicionar a característica de tamanho da fila;
- Acrescentar as demais entidades do FreeRTOS à especificação; e
- Refinar especificação para tratar demais requisitos do sistema.

A elaboração desse trabalho iniciou-se com o estudo do método B. Esse estudo ocorreu principalmente através do desenvolvimento de uma especificação do microcontrolador 8051 (melhor detalhada em [13]) e com colaborações para os trabalhos [4] e [14].

O estudo sobre o FreeRTOS foi divido em duas partes principais: o levantamento bibliográfico sobre o sistema e a analise do código fonte do mesmo. Essas partes serão detalhadas nos parágrafos seguintes.

No levantamento bibliográfico, a documentação disponível sobre o sistema foi examinada. Assim, os principais requisitos, funcionalidade e características do FreeRTOS poderam ser identificados e classificados de acordo com sua importância. Esse trabalho serviu como base para a análise do código fonte e para a elaboração do plano de desenvolvimento de uma modelagem funcional do sistema.

Durante a análise do código fonte, foi possível identificar a arquitetura do sistema e como suas principais abstrações de hardware e funcionalidades são implementados. Nessa análise, percebeu-se que muita vezes o sistema possui detalhes em níveis de assembler, o que segundo Dantas, em [14], também podem ser tratados utilizando o método B.

Através da análise do FreeRTOS descobriu-se que, apesar dele ser um sistema simples e pequeno, a sua modelagem é um trabalho que levará muito tempo e esforço. Assim, como primeiro passo para especificação do sistema, foi desenvolvido um modelo funcional inicial, bastante abstrato, do FreeRTOS. Entretanto, para uma melhor elaboração e aproveitamento desse modelo, antes da sua criação, foi elaborado um plano para identificar quais elementos e funcionalidades que devem fazer parte dessa modelagem funcional inicial.

No plano de modelagem, determinou-se que, para essa modelagem inicial, devido a sua importância, apenas as entidades tarefas e fila de mensagens serão especificadas, sendo as demais entidades (semáforo, co-rotina e mutex) tratadas nas próximas etapas da especificação. Após essa definição, o plano foi divido em módulos incrementais, os quais foram compostos pelas funcionalidades e representações de cada entidade, sendo cada módulo uma etapa da modelagem funcional.

Através da modelagem funcional do sistema, utilizando-se o mecanismo composicional do método B, foi possível definir um esboço da arquitetura da especificação. Nela, o sistema foi divido inicialmente em sete módulos, organizados de acordo com a figura 4.1. Os módulos mais abaixo fornecem operações e implementam elementos para os módulos superiores realizarem suas funções, sendo o módulo *FreeRTOS* responsável por especificar as funcionalidades do sistema.

Entretanto, por tratar-se de um modelo funcional, a modelagem inicial abstraiu muitas características importantes do FreeRTOS, entre elas a característica de prioridade de uma tarefa. A adição dessa característica ao modelo foi feita na etapa de "Refinamento da modelagem desenvolvida para adicionar a característica de prioridade" utilizando-se o mecanismo de refinamento do método B, no qual uma especificação abstrata é refinada para um modelo mais concreto.

Ao total, nessa especificação inicial foram criadas 1.974 linhas, as quais geraram 538 obrigações de prova (seção 3.3). Entre essas obrigações de prova, somente 49 necessitaram da interação humana para sua resolução, sendo as demais provadas automaticamente pela ferramenta adotada pelo projeto. Como fruto dessa etapa, foi publicado o seguinte trabalho [15].

Dando continuidade na metodologia do trabalho, as próximas etapas são: refinar a modelagem adicionando a característica tamanho da fila; implementar as demais entidades do FreeRTOS; e refinar a modelagem para abranger os requisitos do sistema ainda não tratados.

A etapa de refinar a modelagem adicionando a característica tamanho da fila foi desenvolvida com a ajuda dos alunos da disciplina de Métodos Formais do Período 2009.1 da Universidade Federal do Rio Grande do Norte. Nessa etapa, foram criados módulos separados da modelagem desenvolvida até aqui, restando apenas adaptar esses módulos à especificação do sistema.

Após o refinamento das etapas anteriores, a especificação atingirá um nível razoavelmente maduro, restando ainda acrescentar à especificação as entidades semáforo, mutex e co-rotina. Essa etapa consistirá em acrescentar essas entidades de forma abstrata, tratando apenas algumas das suas principais características.

Com a especificação dos elementos do sistema, resta ao modelo, somente adquirir níveis de abstração capazes de abranger os requisitos do sistema ainda não tratados. Com isso, as próximas etapas da modelagem serão realizadas através de refinamentos e refatoramento da modelagem, até que as características do FreeRTOS sejam totalmente especificadas.

Ao final da especificação, pretende-se atingir uma modelagem concreta do FreeRTOS, na qual funcionalidades do sistema possam ser testadas e validadas. Além disso, a especificação aqui desenvolvida servirá como material de apoio ao FreeRTOS, gerando uma documentação do sistema no ponto de vista formal.

1.3 Trabalhos Relacionados

Alguns trabalhos relacionados ao objetivo dessa qualificação são: [16], [17] e [18]. No primeiro, é demonstrada, utilizando o formalismo CSP [19], uma técnica de análise da concorrência entre processos de uma aplicação. Em [17], uma modelagem bastante simples do microkernel L4 é feita em Event B [20]. E por último, no trabalho [18], Craig demonstra uma modelagem em Z [21] de um sistema operacional clássico dividido em camadas. A seguir, tem-se uma explicação mais detalhada de cada um desses trabalhos.

No trabalho desenvolvido por Kleine em [16], a validação de uma aplicação concorrente é feita através da transformação da linguagem intermediária do compilador LLVM para uma modelagem de baixo nível em CSP. Esse modelo é dividido em três partes: Aplicação, onde o comportamento dos processos do sistema é descritos; Domínio, onde os aspectos comuns do domínio da aplicação são especificados; e Plataforma, onde os detalhes da plataforma utilizada são formalizados. Um fator positivo dessa divisão é que essa especificação pode disponibilizar diversas visões do sistema e as partes de domínio e plataforma podem ser parametrizáveis, podendo assim ser reutilizadas em várias aplicações.

Após a criação do modelo CSP, este é analisada através das ferramentas de verificação para CSP, como FDR2 e ProB, que verificam situações de *deadlock* e condições de corrida do sistema, podendo assim garantir o correto funcionamento da concorrência no sistema. Além disso, com a geração do modelo CSP, as atividades implementadas no sistema são abstraídas para um ponto de vista mais prático e abstrato, permitindo assim uma melhor análise do sistema.

No trabalho de Solá, [17], o formalismo utilizado, Event B, é muito parecido com o método B. Ele também é formado por estados, que devem ser mantidos, e operações, denominadas de eventos, responsáveis por alterar esses estados. Assim, na especificação do L4, as abstrações de hardware do sistema, como *threads* e comunicação IPC, foram especificadas através de estados e as chamadas da API do sistema através de eventos.

O principal objetivo de Solá nesse trabalho foi especificar a API do L4. Com isso, devido a API preocupar-se primordialmente com as chamadas ao sistema, o guia para o desenvolvimento dessa especificação foram as chamadas ao sistema. Assim as abstrações de hardware e suas propriedades foram colocadas de acordo com as necessidades surgidas na especificação das atividades das chamadas ao sistema, o que proporcionou uma formalização bastante abstrata desses elementos, mas completa em relação às interações do sistema através de chamadas. Com a especificação do L4 foi possível: validar propriedades estáticas do sistema; descrever mais precisamente o mecanismo do L4; e obter uma documentação mais extensiva da sua API.

Em seu trabalho, [18], Craig começa afirmando que o parte mais importante de uma aplicação é o sistema operacional, que gerencia os recursos usados pela aplicação. Após isso, ele certifica que os métodos formais há tempos estão relacionados com os sistemas operacionais e que, na maioria das vezes, foram utilizados para especificar as operações de fila dos sistemas. De posse dessas afirmações, Craig defende a especificação formal como uma prática de suma importância a ser realizada antes da codificação, pois através dela o sistema pode ser analisado de forma abstrata, como uma entidade matemática. Assim, propriedades importantes do sistema podem ser provadas antes mesmo de sua codificação, o que diminui os riscos do projeto.

Continuando o trabalho, Craig demonstra a divisão em camadas de um sistema operacional convencional, sendo essas: a camada das primitivas de *hardware*, localizada assim do *hardware*; a camada responsável pelo gerenciamento de disco e interrupções de *hardware* (Relógio do sistema); a camada de gerenciamentos de arquivos e controle de interfaces; e a camada de chamadas do sistema utilizada pelas aplicações. Essa divisão proporciona uma modelagem incremental de um SO, na qual os elementos mais abstratos e importantes são tratados inicialmente.

2 *FreeRTOS*

O FreeRTOS é um Sistema Operacional de Tempo Real - SOTR enxuto, simples e de fácil uso. O seu código fonte, feito em *C* e com partes em *assembly*, é aberto e possui um pouco mais de 2.200 linhas de código, que são essencialmente distribuídas em quatro arquivos: *task.c*, *queue.c*, *croutine.c* e *list.c*. Outra característica marcante desse sistema está na sua portabilidade, sendo o mesmo oficialmente portável para 17 arquiteturas mono-processadores diferentes, entre elas a PIC, ARM e Zilog Z80, as quais são amplamente difundidas em produtos comerciais através de sistemas computacionais embutidos [2].

Como a maioria dos sistemas operacionais, o FreeRTOS provê, para os seus usuários, acesso facilitado aos recursos de *hardware*, agilizando com isso o desenvolvimento de sistemas de tempo real. Desse modo, ele funciona como na figura 2.1, uma camada de abstração, localizada entre a aplicação e o hardware, que tem o papel de esconder dos desenvolvedores de aplicações detalhes do hardware, no qual a aplicações será utilizada[1].

Figura 2.1: Camada abstrata proporcionada pelo FreeRTOS [1]

Para prover tal abstração, o FreeRTOS possui 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 aos desenvolvedores serviços como gerenciamento de tarefa, comunicação e sincronização entre tarefas, gerenciamento de memória e controle dos dispositivos de entrada e saída[2].

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

Devido a sua portabilidade e ao fato de trabalhar em ambientes com limitações de hardware, o FreeRTOS pode ser pré-configurado antes da sua execução. Essa configuração é feita por uma biblioteca de configuração, que, através de atributos, armazena as definições de configuração do usuário. Com isso, as aplicações desenvolvidas com o FreeRTOS podem ser mais enxutas e moldadas, provendo uma melhor utilização dos recursos de hardware.

Nas seções a seguir, será explicado, em maiores detalhes, os principais serviços providos pelo FreeRTOS, assim como as bibliotecas e funções que os disponibilizam.

2.1 Gerenciamento de Tarefas e Co-Rotinas

2.1.1 Tarefa

Para entender como funciona o gerenciamento de tarefas do FreeRTOS, é necessário primeiramente entender-se o conceito de tarefa. Tarefa é uma unidade básica de execução que compõe os sistemas, os quais, para realizar suas atividades, geralmente possuem várias tarefas com diferentes obrigações[2]. Para o FreeRTOS, as principais características de uma tarefa são:

Estado: Demonstra a atual situação da tarefa;

Prioridade: Indica a importância da tarefa para o sistema. Uma prioridade varia de zero até uma constante máxima pré configurada pelo projetista;

Pilha de execução: Local onde uma tarefa armazena informações necessárias para a sua execução;

Contexto Próprio: Capacidade de uma tarefa armazenar o ambiente de execução quando suas atividades são suspensas; e

Tempo de bloqueio: Tempo que uma tarefa pode permanecer bloqueada a espera de algum evento.

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

EM EXECUÇÃO: Indica que a tarefa está em execução;

PRONTA: Indica que a tarefa está pronta para entrar em execução, mas não está sendo executada;

BLOQUEADA: Indica que a tarefa está esperando por algum evento para continuar a sua execução; e

SUSPENSA: Indica que a tarefa foi suspensa pelo gerenciador de tarefas através da chamada de uma funcionalidade usada para controlar as tarefas.

No FreeRTOS, uma tarefa só possuir um estado em um determinado instante. Assim, as alterações de estado de uma tarefa funcionam como demonstra o diagrama da figura 2.2. Nela, uma tarefa com o estado EM EXECUÇÃO pode assumir os estados PRONTA, BLOQUEADA ou SUSPENSA. Uma tarefa com o estado PRONTA pode ser suspensa ou entrar em execução, e as tarefas com o estado BLOQUEADA ou SUSPENSA só podem ir para o estado PRONTA.

Um fator preocupante, observado nas possíveis troca de estados de uma tarefa, ocorre quando uma tarefa bloqueada é suspensa e logo em seguida retornada. Assim o tempo de bloqueio de uma tarefa pode ser "enganado" e a tarefa pode retornar antes do tempo indicado para o desbloqueio.

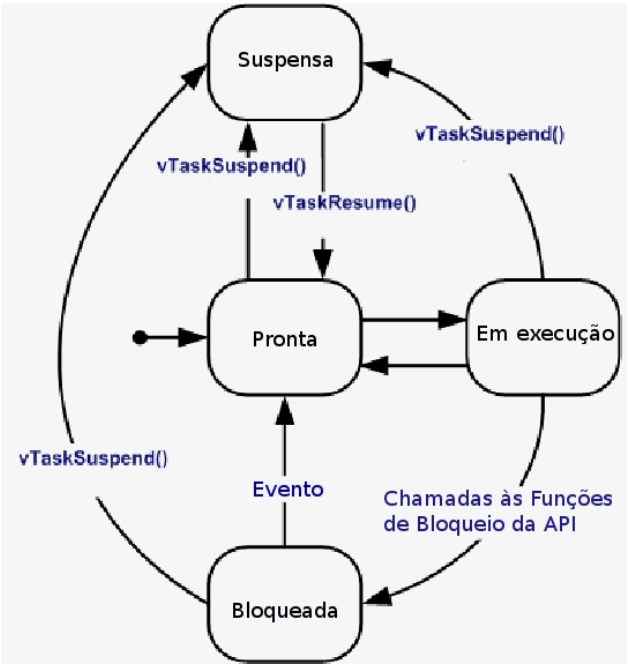


Figura 2.2: Diagrama de estados de uma tarefa no FreeRTOS [2]

Por fim, vale enfatizar que, por tratar-se de um sistema operacional para arquiteturas monoprocessadores, o FreeRTOS não permite que mais de uma tarefa seja executada ao mesmo tempo. Assim, em um determinado instante, apenas uma das tarefas com estado PRONTA pode

assumir o processador e entrar no estado EM EXECUÇÃO. Com isso, para decidir qual tarefa entrará em execução, o FreeRTOS possui um mecanismo denominado escalonador, o qual será detalhado na seção 2.1.3.

Tarefa Ociosa

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

2.1.2 Co-rotinas

Outro conceito importante suportado pelo FreeRTOS é o conceito de co-rotina. Co-rotinas, assim como as tarefas, são unidades de execução independentes que formam uma aplicação. Elas também são formadas por uma prioridade e um estado responsáveis, respectivamente, pela importância da co-rotina no sistema e pela situação da mesma. Para o FreeRTOS, o suporte às co-rotinas é opcional e pré-configurável antes da execução, em tempo de compilação [2].

Uma diferença crucial entre co-rotinas e tarefas está no contexto do ambiente de execução. Co-rotinas não possuem contexto de execução próprio. Consequentemente, sua pilha de execução é compartilhada com as demais co-rotinas do sistema. Diferente das tarefas, que possuem uma pilha própria devido ao armazenamento do seu contexto de execução.

Devido ao fato de co-rotinas compartilharem a mesma pilha de execução, a utilização dessa entidade deve ser feita de forma cuidadosa, pois uma informação armazenada por uma co-rotina pode ser alterada por outra. Por exemplo, a co-rotina *A* armazena na pilha de execução o valor dois, outra co-rotina, ao entrar em execução, altera o valor armazenado pela co-rotina *A*. Assim, quando a co-rotina *A* for ler novamente o valor armazenado, este poderá não ser mais dois.

Os estados possíveis para uma co-rotina são:

EM EXECUÇÃO: Indica que uma co-rotina está em execução;

PRONTA: Indica que uma co-rotina está pronta para ser executada, mas não está em execução; e

BLOQUEADA: Indica que a co-rotina está bloqueada esperando por algum evento para continuar a sua execução.

Como as tarefas, co-rotinas possuem somente um estado em um determinado instante. Assim, as transições entre os estados de uma co-rotina ocorrem como demonstra a figura 2.3. Nela, uma co-rotina em execução pode ir tanto para o estado PRONTA como para o estado BLOQUEADA. Uma co-rotina de estado BLOQUEADA só pode ir para o estado PRONTA e uma co-rotina de estado PRONTA só pode ir para o estado EM EXECUÇÃO.

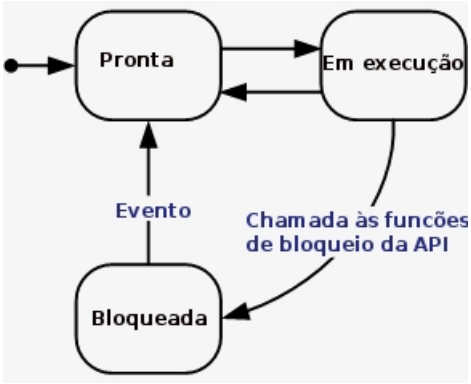


Figura 2.3: Grafo de estados de uma co-rotina [2]

Assim como nas tarefas, a decisão de qual co-rotina irá entrar em execução é feita pelo escalonador, através da chamada à uma funcionalidade específica, esse processo será melhor explicado na seção 2.1.3.

2.1.3 Escalonador de Tarefas

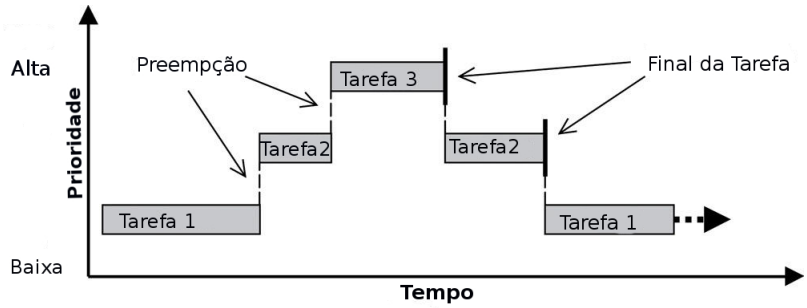
O escalonador é a parte mais importante de um sistema. Ele é quem decide qual unidade de execução[2] irá entrar execução. Além disso, é ele que faz a troca entre a unidade em execução e a nova unidade que irá entrar em execução. No FreeRTOS, o escalonador pode funcionar de três modos diferentes, configuráveis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

- **Preemptivo**: Quando o escalonador interrompe a unidade em execução, alterando assim o seu estado, e ocupa o processador com outra unidade;

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

- **Cooperativo**: Quando o escalonador não tem permissão de interromper a unidade em execução. Assim a interrupção da execução da unidade em processamento e a chamada ao escalonador para decidir quem irá entra em execução devem ser implementadas pelo projetista; e
- **Híbrido**: Quando o escalonador pode comportar-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 é baseada na prioridade e adota a seguinte política: a tarefa em execução deve ter prioridade maior ou igual à tarefa de maior prioridade de estado PRONTA. Assim sempre que uma tarefa, com prioridade maior que a tarefa em execução, entrar no estado PRONTA, 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 PRONTA, como não há nenhuma tarefa em execução esta assume controle do processador entrando em execução.
2. Tarefa 2 entra no estado PRONTA, como esta tem prioridade maior do que a tarefa 1 ela entra em execução passando a tarefa 1 para o estado PRONTA.
3. Tarefa 3 entra no estado PRONTA, como está tem prioridade maior do que a tarefa 2 ela entra em execução passando a tarefa 2 para o estado PRONTA.
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 PRONTA.
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 [3]

Um fato a adicionar sobre a política de funcionamento do escalonador é que, quando duas ou mais tarefas de estado PRONTA tiverem prioridades iguais e maiores que as demais tarefas do mesmo estado, o tempo de execução será divido entre essas tarefas. Assim, ao possuir

duas tarefas de prioridades máximas e estado PRONTA, essas tarefas irão permutar entre si a execução do processador.

Para as co-rotinas, o escalonador funciona de forma cooperativa e baseada na prioridade. Assim, a co-rotina em execução é quem decide o momento da sua interrupção e, em seguida, o sistema deve chamar o escalonador através de uma funcionalidade específica para decidir qual será a próxima co-rotina que irá entrar em execução. A escolha da próxima co-rotina a ser executada também é baseada na maior prioridade, assim como ocorre com as tarefas. Um exemplo do funcionamento do escalonador pode ser visto na seção 2.3.4.

2.1.4 Bibliotecas

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

Criação de Tarefas

Essa biblioteca é responsável pela entidade tarefa e sua criação. Nela, estão presentes um tipo, responsável por representar uma tarefa do sistema, e duas funcionalidades, uma para a criação e outra para a remoção de tarefas do sistema. Em seguida, tem-se o tipo e as funcionalidades que compõem a biblioteca em questão.

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

Controle de tarefas

A biblioteca de controle de tarefas realiza operações sobre as tarefas do sistema. Ela disponibiliza funcionalidades capazes de bloquear, suspender, e retornar uma tarefa do sistema,

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

informar e alterar a prioridade de uma tarefa no sistema. A lista das principais funcionalidades presentes nessa biblioteca pode ser vista a seguir:

- **vTaskDelay**: Funcionalidade usada para bloquear uma tarefa por um determinado tempo. Nessa funcionalidade, para calcular o tempo que a tarefa deve permanecer bloqueada, será levado em consideração o instante da chamada à funcionalidade. Devido a isso, essa funcionalidade não é recomendada para a criação de tarefas cíclicas, pois o instante em que ela é chamada pode variar a cada execução da tarefa, por causa das interrupções que uma tarefa pode sofrer.
- **vTaskDelayUntil**: Funcionalidade usada para bloquear uma tarefa por um determinado tempo. Essa funcionalidade, diferente da *vTaskDelay*, calcula o tempo que a tarefa deve permanecer bloqueada com base no instante do último desbloqueio da tarefa. Assim, se ocorrer uma interrupção no momento da chamada à funcionalidade, o instante que a tarefa foi desbloqueada não ira mudar. Com isso, esse método torna-se recomendável para a criação de tarefas cíclicas.
- **uxTaskPriorityGet**: Funcionalidade usada para informar a prioridade de uma determinada tarefa.
- **vTaskPrioritySet**: Funcionalidade usada para mudar a prioridade de uma determinada tarefa.
- **vTaskSuspend**: Funcionalidade usada para colocar uma determinda tarefa no estado SUSPENSA.
- **vTaskResume**: Funcionalidade usada para colocar uma determinada tarefa suspensa no estado PRONTA.
- **xTaskResumeFromISR** - Funcionalidade usada pelo tratamento de interrupções do sistema. Ela coloca uma determinada tarefa suspensa para o estado PRONTA.

Utilitários de tarefas

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

- **xTaskGetCurrentTaskHandle**: Funcionalidade que retornar uma referência para a atual tarefa em execução.
- **uxTaskGetStackHighWaterMark**: Funcionalidade que retona a quantidade de espaço restante na pilha de uma tarefa.
- **xTaskGetTickCount**: Funcionalidade que retorna o tempo decorrido desde a inicialização do escalonador.
- **xTaskGetSchedulerState**: Funcionaliade que retornar o estado do escalonador.
- **uxTaskGetNumberOfTasks**: Funcionalidade que retorna o número de tarefas do sistema.
- **TaskSetApplicationTag**: Funcionalidade usada para associa uma 'tag' a uma tarefa. Essa tag será utilizada principalmente pelas funcionalidades de rastreamento do sistema. Entretanto, é possível usar essa 'tag' para associar uma função gancho a uma tarefa. Essa função é executada através da chamada à funcionalidade *TaskCallApplicationTaskHook*, informando a tarefa associada.
- **TaskCallApplicationTaskHook**: Funcionalidade usada para chamar a função gancho de uma determinada tarefa.

Controle do Escalonador

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

- **vTaskStartScheduler**: Funcionalidade usada para iniciar as atividades do escalonador, ou seja inicializar o sistema.
- **vTaskEndScheduler**: Funcionalidade usada para encerrar as atividades do escalonador, ou seja finalizar o sistema.
- **vTaskSuspendAll**: Funcionalidade usada para suspender as atividades do escalonador.
- **xTaskResumeAll**: Funcionalidade usada reativar o escalonador quando o mesmo está suspenso.

- **taskYIELD**: Funcionalidade usada para força a troca de contexto[4] entre tarefas.
- **taskENTER_CRITICAL**: Funcionalidade usada para indicar o início de uma região crítica. Ela desabilita temporariamente a característica de preempção do escalonador impedindo que a tarefa em execução seja interrompida por outra.
- **taskEXIT_CRITICAL**: Funcionalidade usada para indicar o final de uma região crítica, permitindo que ocorrá novamento escalonamento preemptivo.
- **taskDISABLE_INTERRUPTS**: Funcionalidade usada para desabilita as interrupções do microcontrolador.
- **taskENABLE_INTERRUPTS**: Funcionalidade usada para habilitar as interrupções do microcontrolador.

Co-rotina

A última biblioteca do serviço de gerenciamento de tarefa e co-rotina é a biblioteca co-rotina. Nela, estão presentes as funcionalidades e tipos responsáveis por criar e gerenciar o elemento co-rotina. A lista completa das funcionalidades presentes nessa biblioteca pode ser vista a seguir:

- **xCoRoutineHandle**: Tipo responsável por representar uma co-rotina.
- **xCoRoutineCreate**: Funcionalidade usada para criar uma nova co-rotina no sistema.
- **crDELAY**: Funcionalidade usada para bloquear uma co-rotina durante uma determinada quantidade de tempo.
- **crQUEUE_SEND**: Funcionalidade usada para enviar uma mensagem para uma fila através de uma co-rotina.
- **crQUEUE_RECEIVE**: Funcionalidade usada para receber uma mensagem de uma fila através de uma co-rotina.
- **crQUEUE_SEND_FROM_ISR**: Funcionalidade especial usada para enviar uma mensagem para uma fila, através de uma co-rotina responsável por tratar uma interrupção.
- **crQUEUE_RECEIVE_FROM_ISR**: Funcionalidade especial usada para receber uma mensagem de uma fila, através de uma co-rotina responsável por tratar uma interrupção.

[4]Troca de contexto é a operação na qual a tarefa em execução é trocada por outra. Para que a troca de contexto sejá realizada, deve existir uma tarefa de prioridade igual à da tarefa em execução

- **vCoRoutineSchedule**: Funcionalidade usada para chamar o escalonador para escolher e colocar em execução a co-rotina de maior prioridade entre as co-rotinas de estado pronto.

2.2 Comunicação e sincronização entre tarefa

Frequentemente tarefas necessitam se comunicar entre elas. Por exemplo, a tarefa *A* depende da leitura do teclado, feita pela tarefa *B*, para disponibilizar em uma tela as teclas digitadas pelo usuário. Para que essa comunicação possa ser estruturada e sem interrupções, os sistemas operacionais possuem mecanismos específicos de comunicações.

A maioria dos sistemas operacionais oferece vários tipos de comunicação entre as tarefas. Geralmente esses tipos são: tarefas trocando informações entre si; tarefas utilizando, de forma sincronizada, o mesmo recurso; tarefas dependentes dos resultados produzidos por outras.

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

2.2.1 Fila de Mensagens

Filas de mensagens são estruturas primitivas de comunicação entre tarefas. Elas funcionam como um túnel, através do qual tarefas enviam e recebem mensagem (figura 2.5). Assim, quando uma tarefa necessita comunicar-se com outra, ela envia uma mensagem para o túnel para que a outra tarefa possa ler sua mensagem [3].

Figura 2.5: Funcionamento de uma fila de mensagens

No FreeRTOS, uma fila de mensagens é formada por:

- A lista das mensagens na fila;
- A lista de tarefas que aguardam para enviar uma mensagem para a fila;
- A lista de tarefas que aguardam pela chegada de uma mensagem na fila;
- Uma variável que indica o tamanho das mensagem da fila; e

- Uma variável responsável por indicar o tamanho da fila, quantidade de mensagens que podem ser armazenadas pela fila. Ou seja, a capacidade máxima da fila.

O funcionamento de uma fila de mensagens no FreeRTOS ocorre da seguinte forma. Primeiro a tarefa remetente envia uma mensagem para a fila e, em seguida, a tarefa receptora retira a mensagem da fila. Entretanto, se, no momento do envio da mensagem, a fila estiver cheia a tarefa remetente é bloqueada e colocada na lista de tarefas que aguardam para enviar uma mensagem para a fila. O mesmo ocorre quando a tarefa receptora tenta receber uma mensagem de uma fila vazia.

A retirada de uma tarefa da listas de espera de uma fila é feita levando em consideração a prioridade das tarefas da lista. Assim, quando uma mensagem é retirada de uma fila cheia, a lista de tarefas que aguardam para enviar uma mensagem para a fila é percorrida e a tarefa de maior prioridade é retirada da fila, sendo consequentemente desbloqueada. Fato parecido ocorre com a fila de tarefas bloqueadas em leitura quando uma mensagem chega a uma fila vazia.

No momento de enviar uma mensagem para uma fila, uma tarefa pode especificar o tempo máximo que ela deve permanecer bloqueada, aguardando para enviar a mensagem. Assim como, ao solicitar uma mensagem para a fila, uma tarefa também pode definir o tempo máximo que ela pode ficar bloqueada, esperando pela chegada de uma mensagem na fila. As funcionalidades que tornam possível essas características serão demonstradas na seção 2.2.4.

2.2.2 Semáforo

Os semáforos são mecanismos usados para realizar a sincronização entre tarefas. Eles funcionam como uma chave de pré-condição para uma tarefa executar uma operação sincronizada ou acessar um recurso compartilhado. Assim, antes de executar tal ação, a tarefa deve solicitar o semáforo responsável pela guarda da ação. Caso o semáforo esteja disponível, a tarefa realiza a ação, caso contrário, a tarefa é bloqueada até que o semáforo seja liberado.

O FreeRTOS disponibiliza dois tipos de semáforos para o usuário, o semáforo binário e o semáforo com contador. A diferença entre esses dois tipos de semáforo está no número de tarefas que podem reter o semáforo ao mesmo tempo. No semáforo binário é possível apenas uma tarefa manter o semáforo. Entretanto, no semáforo com contador existe um número fixo de tarefa (maior ou igual a um) que podem reter o semáforo.

Para controlar o acesso de várias tarefas ao semáforo com contador, ele possui uma variável denominada contador, cujo valor é definido no momento da criação do semáforo. Assim, seu

funcionamento ocorre como demonstra a figura 2.6, para cada tarefa que retém o semáforo, o contador é decrementado e, para cada tarefa que libera o semáforo, o contador é incrementado. Com isso, o semáforo estará indisponível quando o valor do contador for igual a zero e seu valor não poderá ultrapassar o número definido inicialmente.

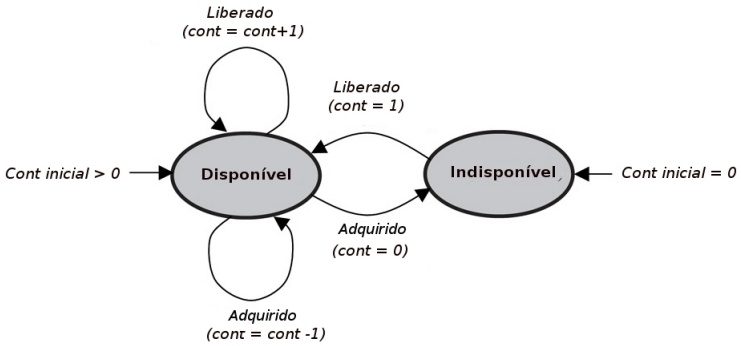


Figura 2.6: Diagrama de estado do semáforo com contador [3]

No FreeRTOS, os semáforos são implementados como uma fila de mensagens que informa o estado do semáforo através da situação da fila. Assim, quando a fila estiver vazia, indica que o semáforo não poderá ser retido e, nas demais situações da fila, indica que o semáforo está liberado. Com isso, para representar-se um semáforo binário, é criada uma fila de capacidade um e, para representar um semáforo com contador, é criada uma fila de capacidade igual ao valor inicial do contador do semáforo.

2.2.3 Mutex

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

O mecanismo de herança de prioridade funciona como demonstra a figura 2.7, quando uma tarefa solicita o mutex, ele primeiro verifica se a tarefa solicitante possui prioridade maior que a tarefa com o semáforo. Caso afirmativo, a tarefa que retém o semáforo tem, momentaneamente, a sua prioridade elevada, para que assim ela possa realizar as suas funções sem interrupções e, conseqüentemente, liberar o semáforo mais rapidamente. Um detalhe interessante desse elemento é que ele utiliza as mesmas funcionalidades do semáforo para reter e liberar o mutex. Essas funcionalidades serão explicadas na seção 2.2.4.

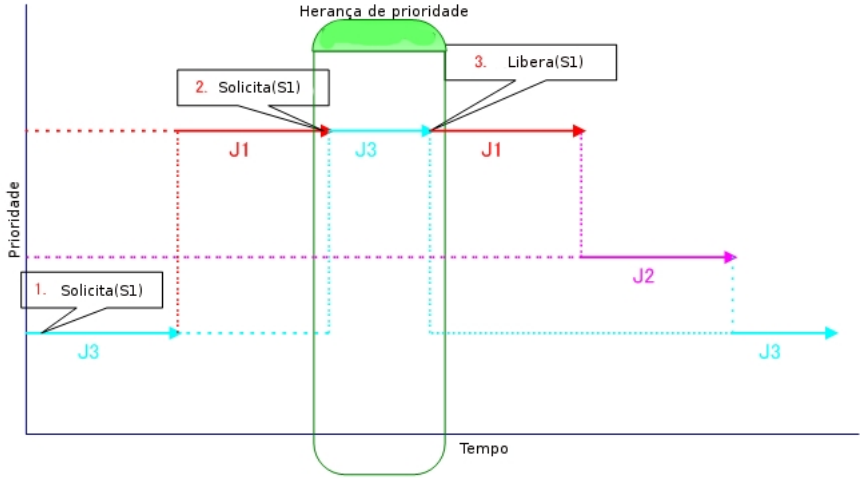


Figura 2.7: Funcionamento do mecanismo de herança de prioridade

2.2.4 Bibliotecas

Para disponibilizar as características de comunicação e sincronização entre tarefas, o FreeRTOS dispõe de um conjunto de funcionalidades e tipos agrupados em duas bibliotecas: gerenciamento de fila de mensagens e semáforo/mutex. Juntas essas bibliotecas possuem vinte e quatro funcionalidades, das quais as principais serão listadas nas seções seguintes.

Gerencialmente de fila de Mensagens

O conjunto de funcionalidades de gerenciamento de uma fila de mensagens é responsável pela criação e utilização da entidade fila de mensagens. Nele, estão presentes funcionalidades responsáveis por instanciar e remover uma fila do sistema e funcionalidades que enviam e recebem mensagens de uma fila do sistema. A seguir, tem-se a lista das funcionalidades mais relevantes dessa biblioteca.

- **xQueueCreate**: Funcionalidade usada para criar uma nova fila de mensagens no sistema.
- **vQueueDelete**: Funcionalidade usada para remover uma fila de mensagens do sistema.
- **xQueueSend**: Funcionalidade usada para enviar uma mensagem para uma fila.
- **xQueueSendToBack**: Funcionalidade usada para enviar uma mensagem para o final de uma fila.
- **xQueueSendToFront**: Funcionalidade usada para enviar uma mensagem para o início de uma fila.

- **xQueueReceive**: Funcionalidade usada para retirar uma mensagem de uma fila.
- **xQueuePeek**: Funcionalidade usada para ler uma mensagem de uma fila, sem removê-la.
- **xQueueSendFromISR**: Funcionalidade especial usada, pelas tarefas quem tratam de interrupções, para mandar uma mensagem para uma fila.
- **xQueueSendToBackFromISR**: Funcionalidade especial usada, pelas tarefas quem tratam de interrupções, para mandar uma mensagem para o final de uma fila.
- **xQueueSendToFrontFromISR**: Funcionalidade especial usada, pelas tarefas quem tratam de interrupções, para mandar uma mensagem para o início de uma fila.
- **xQueueReceiveFromISR** : Funcionalidade especial usada, pelas tarefas quem tratam de interrupções, para mandar uma mensagem de uma fila.

Semáforo/Mutex

Na biblioteca de semáforo e mutex estão implementadas, junto com as suas funcionalidades, as estruturas de sincronização entre tarefas semáforo e mutex. Assim, nesta biblioteca estão presentes funcionalidades que criam e removem semáforos e mutex do sistema, como também funcionalidades utilizadas para solicitar e liberar um semáforo ou mutex. As principais funcionalidades desta biblioteca podem ser vista a seguir.

- **vSemaphoreCreateBinary**: Funcionalidade usada para criar um semáforo binário.
- **vSemaphoreCreateCounting**: Funcionalidade usada para criar um semáforo com contador.
- **xSemaphoreCreateMutex**: Funcionalidade usada para criar um mutex.
- **xSemaphoreTake**: Funcionalidade usada para reter um semáforo ou um mutex.
- **xSemaphoreGive**: Funcionalidade usada para liberar um semáforo ou um mutex retido.
- **xSemaphoreGiveFromISR**: Funcionalidade especial usada, pelas tarefas que tratam de interrupções do sistema, para liberar um semáforo binário ou com contador. Não deve ser utilizada para mutex.

2.3 Utilizando práticas dos elementos FreeRTOS

Para construir uma aplicação de tempo real utilizando o FreeRTOS, o desenvolvedor deve seguir determinadas restrições impostas pelo sistema. A maioria destas restrições são parâmetros de configuração e modelos para a criação dos elementos do sistema. Assim, com o intuito de ajudar o desenvolvedor a criar suas primeiras aplicações, o FreeRTOS disponibilizou, junto com seu código fonte, aplicações exemplos classificadas por plataformas. Desse modo, essas aplicações exemplos podem ser utilizadas como ponto de partida na criação de novos projetos.

Entretanto, a criação e análise de uma nova aplicação no FreeRTOS é uma atividade que necessita de maior conhecimento sobre as suas funcionalidades, fugindo assim do objetivo geral desse capitulo, que é proporcionar uma breve introdução ao FreeRTOS, demonstrando seus principais conceitos, funcionalidades e características. Com isso, para um melhor entendimento das explicações apresentadas neste capítulo, será demonstrada, nas seções seguintes, de forma didática, a utilização das principais entidades aqui discutidas, tarefa, co-rotinas, fila de mensagens e semáforos.

2.3.1 Utilização da entidade tarefa

A tarefa é a parte mais importante de uma aplicação. É nela que são colocadas as ações responsáveis pelo funcionamento da aplicação[3]. No FreeRTOS, as ações realizadas pelas tarefas são colocadas dentro de rotinas, as quais devem seguir uma estrutura pré-definida, demonstrada pela figura 2.8. Nela, tem-se que uma rotina deve ser formada inicialmente por um cabeçalho com o seu nome e seguido de uma lista de parâmetros utilizados por ela. Em seguida, tem-se o código que realiza as finalidades da tarefa, no qual um laço infinito é colocado para abrigar a parte repetitiva desse código. Assim, a atividade de uma tarefa nunca termina, ficando sob o controle do escalonador.

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

Figura 2.8: Estrutura da rotina de uma tarefa

Um exemplo concreto da criação de uma tarefa pode ser visto na figura 2.9. Nela, tem-se a rotina *cyclicalTasks*, que utiliza a funcionalidade *vTaskDelayUntil* (seção 2.1.4) para bloquear a

execução da tarefa em intervalos iguais de tempo. Essa funcionalidade possui como parâmetros, respectivamente, o último tempo que a tarefa foi reativa do estado SUSPENSA e o período que a tarefa deve permanecer bloqueada. Após isso, a tarefa é criada no sistema junto com sua prioridade, pilha de contexto e nome. Essa operação é feita através da funcionalidade *xTaskCreate* que possui como parâmetros os seguintes argumentos:

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

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

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

pvParameters : Lista de valores dos parâmetros de entrada 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 2.9: Aplicação formada por uma tarefa cíclica

Para finalizar o exemplo da figura 2.9, após ser criada a tarefa, o escalonador do sistema deve ser iniciado e com ele a aplicação. Essa operação é feita pela a funcionalidade *vTaskStartScheduler()*, localizada no final do código.

2.3.2 Utilização da fila de mensagens

A utilização de uma fila de mensagens é resumidamente demonstrada na aplicação da figura 2.10. Nela, inicialmente tem-se a estrutura *AMessage*, que 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*. Para isso, o método *xQueueCreate* recebe como parâmetros, respectivamente, a quantidade de mensagens que a fila pode armazenar e o tamanho da mensagens manuseadas por ela.

Estabelecida a fila de mensagens, é necessário agora definir as tarefas que irão enviar e receber mensagens da mesma. Na figura 2.10, estão presentes apenas as rotinas de cada uma dessas tarefas, sendo que a explicação completa de como é criada uma tarefa foi demonstrada na seção 2.3.1. A seguir tem-se a explanação sobre cada uma dessas rotinas.

Para enviar uma mensagem para a fila *xQueue*, a rotina *sendTask* utiliza-se da funcionalidade *xQueueSend*. Essa funcionalidade possui como parâmetros a fila para qual a mensagem será enviada, a mensagem que será enviada para a fila e o tempo máximo que a tarefa poderá ficar bloqueada aguardando para enviar a mensagem.

Por último, na rotina *receiveTask*, uma mensagem é retirada da fila *xQueue* através da funcionalidade *xQueueReceive*. Para isso, ela utiliza como argumentos, respectivamente, a fila, de onde será retirada a mensagem, o local que o endereço da mensagem que irá ser armazenado, e o tempo máximo que a tarefa pode ficar esperando pela fila. Como retorno, essa funcionalidade informa se a mensagem foi retirada da fila com sucesso ou não, sendo, com isso, utilizada como guarda para o código que deve ser executado após a retirada da mensagem.

2.3.3 Utilização do semáforo

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

Um exemplo de uma aplicação que utiliza um semáforo para controlar o acesso de um recurso compartilhado pode ser visto na figura 2.11. Nela, inicialmente é criada a variável *xSemaphore* para armazenar uma referência ao novo semáforo. Em seguida, o método *vSemaphoreCreateBinary* é usado para criar o novo semáforo e retornar uma referência para o mesmo.

```
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 receiveTask( 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 2.10: Aplicação que utiliza uma fila de mensagens

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 pela segunda condição *if*, a 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 retê-lo e usar o recurso compartilhado.

```
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 2.11: Aplicação que demonstra a utilização de um semáforo

A utilização do mutex é bem parecida com a do semáforo binário. A diferença, para o usuário, entre os dois mecanismos está apenas no método de criação da entidade, *xSemaphoreCreateMutex*. As formas e os métodos para solicitar e liberar o mutex são os mesmos do semáforo, *xSemaphoreTake* e *xSemaphoreGive*. Com isso, para transformar a aplicação da figura 2.11 de semáforo para mutex basta apenas trocar o método *vSemaphoreCreateBinary* por *xSemaphoreCreateMutex*.

2.3.4 Utilização das co-rotinas

As co-rotinas, assim como as tarefas, possuem as ações resposáveis pelas atividades da aplicação. Essas ações são colocadas dentro das rotinas executadas pelas co-rotinas, as quais devem seguir o modelo demonstrado na figura 2.12. Nele, a rotina deve possuir um nome, seguido de dois parâmetros *xHandle*, uma referência para a co-rotina que será utilizado pelas funcionalidades de controle da co-rotina, e *uxIndex*, que é usado opcionalmente no desenvolvimento do

código da co-rotina. Por último, o código de execução da co-rotina deve ser iniciado com a função *crSTART(xHandle)* e finalizado pela função *crEND()*.

```
void vACoRoutineFunction(xCoRoutineHandle xHandle,
unsigned portBASE_TYPE uxIndex ){
        crSTART( xHandle );
        for( ;; )
        {
            -- Co-routine application code here. --
        }
        crEND();
    }
```

Figura 2.12: Modela da rotina de execução de uma co-rotina

Um exemplo completo da utilização de uma co-rotina pode ser vito na figura 2.13. Nesse exemplo é criada uma co-rotina que, através da rotina *vFlashCoRoutine*, controla o funcionamento de um LED. Para isso ela interrompe a sua execução durante um tempo de dez ticks (unidade de tempo do FreeRTOS), utilizando a funcionalidade *crDELAY*, que tem como argumentos o gancho da co-rotina e o tempo de bloqueio da co-rotina. Após isso, a função de controle do LED *vParTestToggleLED* é chamada. Observa-se, nesse exemplo, a utilização dos parâmetros da rotina como argumentos para a funcionalidade *crDELAY*.

Com a criação da rotina utilizada pela co-rotina, resta apenas criar a co-rotina no sistema. Esse trabalho é feito através da funcionalidade *xCoRoutineCreate*, que recebe como parâmetros respectivamente a rotina que será associada à co-rotina, a prioridade da co-rotina e o valor do parâmetro *uxIndex* da rotina associada à co-rotina.

Por fim, devido à política de escalonamento cooperativo das co-rotinas, ao final da aplicação de figura 2.13 é colocada na rotina *vApplicationIdleHook* (rotina executada pela tarefa ociosa, seção 2.1.1) a funcionalidade *vCoRoutineSchedule* que chama o escalonador para realizar a troca da co-rotina em execução. Assim, sempre que a co-rotina em execução é bloqueada, a tarefa ociosa entra em ação chamando o escalonador para fazer a troca de co-rotinas.

```
...
void main( void )    {
// This time i is passed in as the index.
xCoRoutineCreate( vFlashCoRoutine, PRIORITY_0, 10 );

    // NOTE: Tasks can also be created here!
// Start the scheduler.
    vTaskStartScheduler();
}

void vFlashCoRoutine( xCoRoutineHandle xHandle,
unsigned portBASE_TYPE uxIndex ){
// Co-routines must start with a call to crSTART().
crSTART( xHandle );
        for( ;; ){
            crDELAY( xHandle, uxIndex);
            vParTestToggleLED;
        }
// Co-routines must end with a call to crEND().
crEND();

}

void vApplicationIdleHook( void ){
        vCoRoutineSchedule( void );
}
```

Figura 2.13: Aplicação que demonstra a utilização de uma co-rotina

3 *Método B*

Métodos Formais provêem abordagens formais para a especificação e construção de sistemas computacionais. Eles utilizam-se de conceitos matemáticos sólidos como lógica de primeira ordem e teorias dos conjuntos para a criação e verificação de sistemas consistentes, seguros e sem ambigüidades. Devido aos seus rigorosos métodos de construção, a sua principal utilização, embora timidamente, tem sido na criação de sistemas críticos para as indústrias de aeronáutica, viação férrea, equipamentos médicos e empresas que movimentam grandes quantidades monetárias, como os bancos[7].

Segundo a Honeywell, uma empresa que desenvolve sistemas para aeronaves, a utilização de métodos formais no processo de desenvolvimento provê várias vantagens, entre eles estão [22]:

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

O método B[11] é uma abordagem formal usada para especificar e construir sistemas computacionais seguros. Seu criador, Jean-Raymond Abrial, junto com a colaboração de outros pesquisadores da universidade de Oxford, procurou reunir no método B vários conceitos presentes nos demais métodos formais. Entre esses conceitos, destacam-se as pré e pós condições (seção 3.2.2), o desenvolvimento incremental através de refinamentos (seção 3.4) e a modularização da especificação. A seguir, tem-se a explicação de como e feito o 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 a criação de um módulo, que define em alto nível um modelo funcional do sistema. Em B, esses módulos são denominados de Máquinas Abstratas (*MACHINE*). Nessa fase de modelagem, técnicas semi-formais como UML podem ser utilizadas e, em seguidas, transformadas para a notação formal do método B. Após a criação dos módulos, esses são analisados estaticamente para verificar se são coerentes e implementáveis.

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

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

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

Atualmente, o desenvolvimento de sistemas utilizando o método B pode ser apoiado por diversas ferramentas com funcionalidades que vão da análise estática da especificação até a geração de código em linguagens de programação. Uma das mais famosas e completas ferramentas de apoio ao desenvolvimento de sistemas utilizando o método B é o AtelierB [23]. Nela, é possível, além da análise sintática e estática da especificação, gerenciar-se projetos, controlando as dependências entre os vários módulos que constituem uma especificação. Devido as suas vastas funcionalidades e popularidade, o AtelierB será adotado como ferramenta padrão

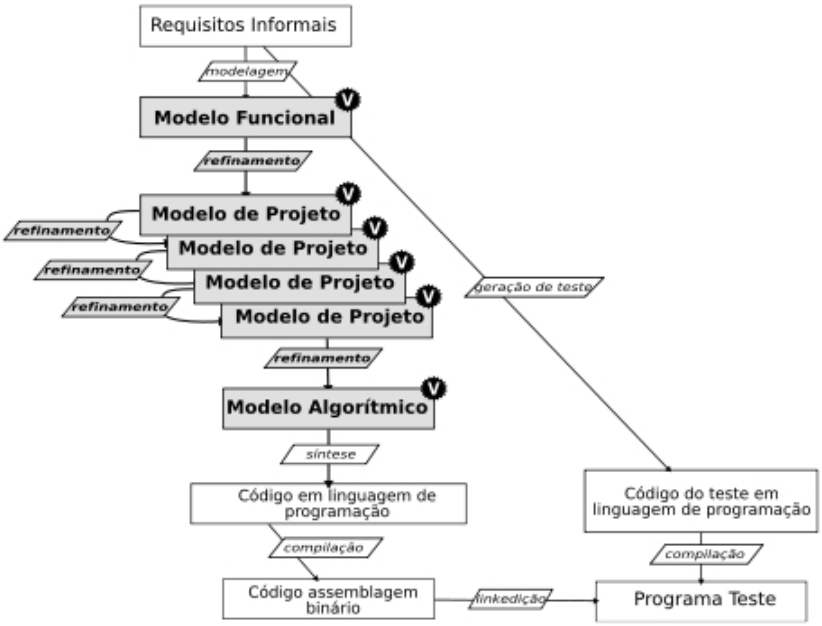


Figura 3.1: Etapas do desenvolvimento de sistema através do método B [4]

desse trabalho.

Um exemplo de sistemas desenvolvidos através da abordagem B é o controle de porta do metrô de paris, desenvolvido pela Clearsy[1], empresa especialista em sistemas críticos. Esse sistema impede que o trem saia da estação quando a porta do mesmo estiver aberta, evitando assim possíveis acidentes[24].

3.2 Máquina Abstrata

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

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

[1] www.clearsy.com

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

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

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

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

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

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

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

OPERATIONS

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

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

Figura 3.2: Maquina abstrata de tarefas

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

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

3.2.1 Especificação do estado da máquina

Nessa parte, são determinados os estados que uma máquina pode assumir. Esses estados são definidos através das variáveis e dos seus invariantes (**INVARIANT**). As variáveis são diferentes elementos que compõem o estado do sistema. Os invariantes são expressões lógicas que determinam os valores que as variáveis podem assumir. Assim, uma especificação só pode definir o correto funcionamento da máquina, quando ela encontra-se em um estado válido, nada sendo especificado para os demais casos.

O estado de uma máquina é especificado por meio de lógica dos predicados, teoria dos conjuntos e aritmética dos inteiros, permitindo com isso uma análise estática da máquina através das expressões lógicas geradas apartir de sua especificação. No exemplo da figura 3.2, o estado da máquina *kernel* foi especificado através da variável *tasks*, sendo $tasks \in \mathbb{P}(TASK)$ e $\mathbf{card}(tasks) \leq 10$, o que define que *tasks* deve ser um conjunto de *TASK* e que o tamanho máximo permitido para o conjunto *tasks* é de dez elementos.

3.2.2 Especificação das operações da máquina

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

O cabeçalho de uma operação é composto por um nome, uma lista de parâmetros de entrada e uma lista de parâmetros de saída [2], sendo os parâmetro de entrada e os parâmetros de saída argumentos opcionais. Um exemplo de uma operação com parâmetros de entrada e saída pode ser visto na figura 3.3. Nela, o nome da operação é *query_task*, o parâmetro de entrada é *task* e o parâmetro de saída é *ans*.

A operação propriamente dita é formada por pré-condição e corpo da operação. Na pré-condição, são colocadas as informações sobre todos os parâmetros de entrada e as condições que devem ser satisfeitas para que a operação seja executada. Com isso, a pré-condição funciona como uma premissa que deve ser suprida para que a operação funcione corretamente. Por exemplo, na figura 3.2, para que a operação (add_task) funcione corretamente, ou seja, não leve

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

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

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

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

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

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

Substituição Simples

A substituição simples é definida da seguinte forma:

$$x := E$$

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

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

Onde se tem que, no predicado P, a variável x deve ser substituída por E.

Substituição Múltipla

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

variáveis tem a seguinte forma:

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

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

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

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

Substituição Condicional

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

Como nas linguagens de programação, a notação de máquina abstrata também permite a construção de atribuições condicionais, as quais são feitas através da substituição condicional. Com isso, uma substituição condicional funciona da mesma forma que nas linguagens de programação. Nela, inicialmente uma expressão lógica é avaliada e, em seguida o caminho que a estrutura deve seguir, quais atribuições devem ser realizadas, é escolhido de acordo com o resultado da expressão. A forma como é especificada uma substituição condicional na ANM pode ser vista a seguir:

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

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

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

.

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

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

Substituição não determinística ANY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Obrigações de Prova

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

Resumidamente, a análise estática de uma máquina abstrata, através das obrigações de prova, avalia primeiramente se a máquina possui estados válidos, ou seja, se pelo menos uma combinação dos estados é alcançada pela máquina. Caso a máquina possua estados válidos, é avaliado se estes são alcançados na inicialização da máquina e ao final de cada operação. 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 dessas obrigações de prova e como elas são geradas.

3.3.1 Consistência do Invariante

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

$$\exists v.I$$

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

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

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

O que pode ser provado como verdadeiro instanciando *tasks* com $\emptyset$.

3.3.2 Obrigação de prova da inicialização

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

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

$$[T]I$$

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

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

O que pode ser facilmente provado como válido.

3.3.3 Obrigação de prova das operações

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

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

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

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

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

3.4 Refinamento

A linguagem abstrata demonstrada até agora é usada principalmente para criar uma modelagem funcional de sistemas e componentes. Nela, o principal objetivo é descrever o comportamento do sistema sem se preocupar com detalhes de como tal comportamento será implementado ou de como os dados serão manipuladas pelo computador. Entretanto, para realizar uma modelagem mais concreta e passível de implementação, é necessário que notações matemáticas abstratas utilizadas na modelagem do sistema, como conjuntos e substituições não determinística, sejam descritas de forma mais concreta, o que é possível através do refinamento. Além disso, o refinamento pode ser usado para construir um modelo abstrato de forma incremental com da adição sucessivas de formalizaçoes dos requisitos[10].

Através da técnica de refinamento, o método B possibilita um desenvolvimento gradativo do sistema. Nele, um sistema é especificado em estágios que vão da modelagem abstrata até um nível algoritmo denominado de implementação. Entre esses níveis de abstração existem modelos intermediários chamados de refinamentos, que combina especificações abstratas com detalhes de implementação.

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

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

Como ocorreu na seção de máquina abstrata 3.2, para um melhor entendimento, a especificação de um refinamento será dividida basicamente em duas partes principais: refinamento do estado da máquina abstrata e refinamento das operações da máquina abstrata. Em seguida, tem-se o delineamento dessas duas partes principais.

3.4.1 Refinamento do Estado

No refinamento de dados, como é reconhecido o refinamento do estado, tem-se o objetivo de especificar o estado de uma máquina em uma forma mais concreta, ou seja, mais próxima à utilizada pelo computador. Para isso, estruturas abstratas, como conjuntos e relações, são substituídas por mecanismo mais implementáveis como vetores e seqüências.

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

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

REFINEMENT *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 o refinamento do estado da máquina, é necessário especificar o refinamento das suas operações. Nesse processo, as operações abstratas são reescrita de forma mais concreta, podendo manipular os estados da máquina refinada e os novos estados criados no refinamento. Além disso, as operações refinadas devem possuir o mesmo comportamento das operações abstratas, garantindo assim uma coerência com a especificação inicial.

As operações de um refinamento devem possuir a mesma assinatura das operações do módulo relacionado à ele, ou seja, ter os mesmo nomes e parâmetros de entrada e saída. Entretanto,

nas operações de um refinamento, não é necessário a declaração da pré condição (**PRE**), uma vez que essa foi definida em um nível mais abstrato e é suficiente para garantir que o tipo do parâmetro de entrada permaneça o mesmo.

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

3.4.3 Obrigação de prova do refinamento

A análise estática que confere se um refinamento é consistente com o nível abstrato acima dele é feita através de obrigações de prova e pode ser dividida em duas partes, obrigação de prova da inicialização e obrigação de prova das operações. Entretanto, na obrigação de prova das operações, são possíveis ainda dois tratamentos diferentes, obrigações de prova para as operações sem parâmetros de retorno e a obrigação de prova para as operações com parâmetros de retorno. A seguir é demonstrado como é realizada cada uma dessas obrigações de prova do refinamento.

Obrigação de Prova da Inicialização

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

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

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

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

falsidade.

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

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

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

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

Obrigação de prova da operação sem parâmetro de retorno

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

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

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

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

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

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

Obrigação de prova da operação com parâmetro de saída

No refinamento de uma operação com saídas, cada saída do refinamento da operação deve estar ligada a uma saída da operação refinada, sendo necessário assim que a operação do refinamento tenha a mesma quantidade de parâmetro de saída da operação refinada. Além disso, renomeando out' como o conjunto de parâmetros de saída do refinamento e deixando *out* como o conjunto dos parâmetros de saída da operação refinada, cada valor de out' deve possuir um correspondente em *out*. Em outra palavra, cada execução de *S1* deve encontrar uma execução *S*, na qual out' produzido por *S1* seja igual ao *out* produzido por *S*.

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

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

Nela, $S1[out'/out]$ significa que, nas atribuições de *S1*, cada ocorrência de *out* deve ser substituída por out' e, antes dessa substuição, a máquina deve possuir um estado válido e sua pré-condição deve ser alcançada, $I \wedge J \wedge P$. As demais verificações são similares à obrigações de prova das operações sem parâmetros de saída.

4 *Primeiro passo da Modelagem*

Toda longa caminhada começa com o primeiro passo. Como primeiro passo desse trabalho, foi desenvolvida, de forma planejada, uma modelagem funcional do FreeRTOS, na qual apenas alguns de seus requisitos foram especificados. Para o planejamento dessa modelagem, levaram-se em consideração principalmente as técnicas de modularização e desenvolvimento incremental, proporcionadas pelo método B, as quais estão descritas a seguir:

1. Parte dos requisitos funcionais do sistema podem ser abstraídos em sua modelagem inicial. Assim, tais requisitos são tratados posteriormente através de refinamentos horizontais ou extensões da especificação, o que proporciona uma especificação incremental do sistema. O planejamento incremental dessa especificação será explicado nas seções 4.1 e 4.2.

2. Quando os requisitos do sistema não apresentarem dependências entre si, eles podem ser especificados em diferentes módulos. Esses módulos comunicam-se entre si utilizando os mecanismos de composição proporcionados pelo método B (visão, inclusão, etc). A divisão da especificação em módulos será detalhada na seção 4.3.

Sabe-se, através do capítulo 2, que as principais entidades do FreeRTOS são tarefas, fila de mensagens, co-rotinas, semáforos, mutex e escalonador. Além disso, sabe-se também que semáforo e mutex são estruturas baseadas na entidade fila de mensagens, sendo esta necessária em qualquer modelagem inicial. Assim, nesse primeiro passo, devido às suas utilidades, as estruturas escolhidas para serem formalizadas, junto com algumas de suas funcionalidades, foram: tarefa, fila de mensagens e escalonador[1].

[1] As funcionalidades e propriedades do escalonador serão tratadas dentro da entidade tarefa

4.1 Tarefa

Nesse ponta pé inicial, para tornar-se possível a criação da modelagem funcional, muitas propriedades da entidade tarefa foram abstraídas. Resumidamente, apenas a propriedade de estado de uma tarefa foi formalizada inicialmente. Através dessa formalização, requisitos importantes do sistema poderam ser especificados, são eles: somente uma tarefa deve estar em execução em um terminado instante e uma tarefa só pode possuir um estado ao mesmo tempo.

A especificação da entidade tarefa foi planejada de forma incremental. Previamente, foram criados os estados necessários para modelar tal entidade no nível de abstração sugerido. Após isso, algumas das funcionalidades do sistema, relacionadas a essa entidade, foram formalizadas a cada etapa de criação do modelo. Ao final, as seguintes funcionalidades foram abrangidas de forma abstrata por essa especificação inicial:

- Criação de tarefas: *xTaskHandle*, *xTaskCreate*, *vTaskDelete*.
- Controle de tarefas: *vTaskDelay*, *vTaskDelayUntil*, *uxTaskPriorityGet*, *vTaskPrioritySet*, *vTaskSuspend*, *vTaskResume*.
- Utilitários de tarefas:*xTaskGetCurrentTaskHandle*, *uxTaskGetNumberOfTasks*, *xTaskGetTickCount*, *xTaskGetSchedulerState*.
- Controle do escalonador: *vTaskStartScheduler*, *vTaskEndScheduler*, *vTaskSuspendAll*, *xTaskResumeAll*.

4.2 Fila de mensagens

Para a entidade fila de mensagens, a principal propriedade especificada foi a quantidade de mensagens que esta pode armazenar. Como ocorreu com a entidade Tarefa, a formalização dessa entidade, junto com suas funcionalidades, foi distribuída entre as várias etapas de desenvolvimento da modelagem inicial. Com essa especificação, os seguintes requisitos de sistemas foram abrangidos pela modelagem: bloquear a tarefa que tentar enviar uma mensagem para uma fila cheia e bloquear a tarefa que tentar receber uma mensagem de uma fila vazia. As funcionalidades relacionadas à fila de mensagens tratadas nessa especificação foram: *xQueueCreate*, *vQueueDelete*, *xQueueSend*, *xQueueSendToBack*, *xQueueSendToFront*, *xQueueReceive*, *xQueuePeek*.

4.3 A modelagem funcional

A modelagem discutida nas seções anteriores foi construída e verifica utilizando a ferramenta AtelierB 4.0[25]. Nessa modelagem, foram especificadas três entidades básicas do sistema: tarefa, fila de mensagens e escalonador. Tal especificação foi estruturada através de sete módulos, os quais serão explicados a seguir:

Módulo Config: Nesse módulo foram tratadas as configurações do comportamento do sistema. Por exemplo, a prioridade máxima que uma tarefa pode possuir é um parâmetro de configuração desse módulo.

Módulo Types: Esse módulo é responsável por definir os tipos utilizados na especificação do FreeRTOS. Um exemplo de um tipo especificado nesse módulo é a prioridade de uma tarefa, que só pode assumir valores de um subconjunto finito dos naturais.

Módulo Task: Nesse módulo são definidos os estados e operações responsáveis por formalizar a entidade tarefa.

Módulo Queue: Esse módulo é similar ao módulo Task, só que nele a entidade especificada é a fila de mensagens.

Módulo Scheduler: Esse módulo simplesmente mantém e manipula o estado do escalonador.

Módulo FreeRTOSBasic: Esse módulo funciona como uma camada abstrata entre o módulo *FreeRTOS* e os demais módulos básicos. Nele, as operações implementadas pelos módulos *Task*, *Queue* e *Scheduler* são agrupadas em funções mais abstratas, que servem como base para especificar as funcionalidades do *FreeRTOS*.

Módulo FreeRTOS: Finalmente, no Módulo *FreeRTOS* são especificadas as funcionalidades das bibliotecas do FreeRTOS.

A organização desses módulos pode ser vista na figura 4.1. Nela, resumidamente os módulos inferiores servem de base para a especificação dos módulos superiores, restando os módulos *Config* e *Types*, que servem de apoio para toda a especificação. Nas seções seguintes tem-se em detalhe como foi o desenvolvimento dessa especificação inicial.

4.3.1 Tarefa

Para representar o elemento tarefa existem diversas abordagens, funções de mapeamento, seqüências e conjuntos. A abordagem escolhida nesse trabalho inicial foi a representação atra-

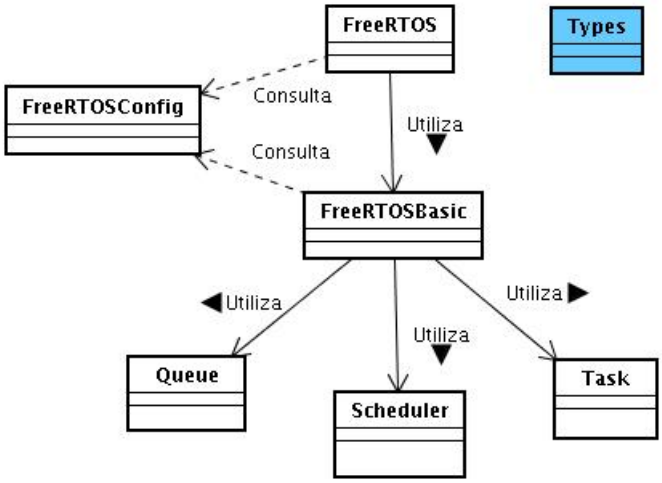


Figura 4.1: Esbouço da arquitetura da especificação

vés de conjuntos, o que facilita a verificação da especificação pelo provador de teoremas. Assim, a modelagem da entidade tarefa e suas propriedades foram feitas de acordo figura 4.2. Nela, a variável *active* informa quando o sistema está ativo. A variável *task* armazena todas as tarefas criadas no sistema. Em seguida, as variáveis *ready*, *blocked*, *running* e *suspended* armazenam as tarefas dos estados EM EXECUÇÃO, PRONTA, BLOQUEADA e SUSPENSA, respectivamente. Por fim, a variável *idle* representa a tarefa ociosa.

Um exemplo do funcionamento da especificação da figura 4.2 seria o seguinte. Enquanto o sistema estiver inativo, o valor da variável *active* deve ser falso. As novas tarefas criadas no sistema serão armazenadas na variável *tasks* e na variável de seu respectivo estado, exceto a tarefa ociosa que é armazenada em *idle*. Por fim, ao iniciar-se a execução do sistema a variável *active* receberá o valor verdadeiro.

VARIABLES
$active, tasks, blocked, running, ready, suspended, idle$
INVARIANT
$active \in \mathbf{BOOL} \wedge tasks \in \mathbf{FIN}(TASK) \wedge running \in TASK \wedge idle \in TASK$
$\wedge\, blocked \in \mathbf{FIN}(TASK) \wedge ready \in \mathbf{FIN}(TASK) \wedge suspended \in \mathbf{FIN}(TASK)$

Figura 4.2: Representação de uma tarefa pela máquina *Task*

Estendendo um pouco mais a especificação do módulo *Task*, tem-se, na figura 4.3, a continuação do invariante da máquina. Nele, propriedades importantes do sistema são especificadas. São elas: uma tarefa deve possuir somente um estado em determinado momento; Enquanto o escalonador não estiver ativado todas as tarefas estarão no estado pronto; quando o escalonador estiver ativo a tarefa ociosa deve estar pronta ou em execução; sempre existirá somente uma tarefa em execução.

$$
\begin{array}{l}
blocked \subseteq tasks \wedge ready \subseteq tasks \wedge suspended \subseteq tasks \wedge \\
ready \cap blocked = \emptyset \wedge blocked \cap suspended = \emptyset \wedge suspended \cap ready = \emptyset \wedge \\
(active = FALSE \Rightarrow tasks = ready) \wedge \\
(active = TRUE \Rightarrow \quad (idle = running \vee idle \in ready) \wedge \\
\qquad\qquad\qquad\qquad running \notin (blocked \cup ready \cup suspended) \wedge \\
\qquad\qquad\qquad\qquad tasks = \{running\} \cup suspended \cup blocked \cup ready)
\end{array}
$$

Figura 4.3: Continuação do invariante da máquina *Task*

Em adição a essa especificação, no módulo *Task*, foram criadas operações básicas para manipular as variáveis de estado. Assim, essas operações servem como base para a modelagem das funcionalidades do FreeRTOS relacionadas à entidade Tarefa. Ao total, foram criadas doze operações elementares, das quais duas são apresentadas pelas figuras 4.4 e 4.5. As demais operações e suas obrigações de prova podem ser consultadas no repositório do projeto[2].

```
result ⟵ t_create(priority) =
PRE
  priority ∈ PRIORITY ∧
  active = FALSE
THEN
  ANY task WHERE
    task ∈ TASK ∧ task ∉ tasks
  THEN
    tasks := {task} ∪ tasks ||
    ready := {task} ∪ ready ||
    result := task
  END
END;
```

Figura 4.4: Especificação da operação *t_create*

Na figura 4.4, tem-se a operação *t_create*, que é responsável por criar um nova tarefa. Essa operação pode ser utilizada somente quando o escalonador não estiver acionado. Seu parâmetro de entrada, *priority*, indica a prioridade da tarefa que será criada. Entretanto, esse parâmetro só será utilizado no refinamento da operação. Como retorno da operação tem-se a tarefa criada.

```
t_startScheduler =
PRE
  active = FALSE
THEN
  active := TRUE ||
  blocked, suspended := ∅, ∅ ||
  ANY idle_task WHERE
    idle_task ∈ TASK ∧
    idle_task ∉ tasks
  THEN
    tasks := {idle_task} ∪ tasks ||
    idle := idle_task ||
    ANY task WHERE
      task ∈ ready ∪ {idle_task}
    THEN
      running := task ||
      ready := (ready ∪ {idle_task}) − {task}
    END
  END
END;
```

Figura 4.5: Especificação da operação *t_startScheduler*

A segunda operação a ser explicada é a *t_startScheduler*, figura 4.5, responsável por iniciar o funcionamento do sistema. Durante a execução dessa operação, a tarefa ociosa do sistema

[2]Página do repositório do projeto: http://code.google.com/p/freertosb/

é criada e o escalonador escolhe uma tarefa para entrar em execução. Observa-se que, devido à prioridade ainda não ser tratada nessa especificação, a tarefa a ser executada é escolhida de forma aleatória, ficando tal preocupação para o refinamento da operação.

Um exemplo da especificação de uma funcionalidade do FreeRTOS pode ser visto na figura 4.6. Nela, a funcionalidade *xTaskCreate* é demonstrada. O seu comportamento pode ocorrer de duas formas: ou uma nova tarefa do sistema é criada e passada como retorno da função; ou nenhuma tarefa é criada e uma mensagem de erro é retornada. Para criar uma nova tarefa, essa funcionalidade utiliza-se da função *t_create* demonstrada na figura 4.4.

result, *handle* ⟵
  xTaskCreate(*code*, *name*,
  stackSize, *params*,
  priority) =
PRE
  code ∈ *TASK_CODE*∧
  name ∈ *NAME*∧
  stackSize ∈ *NATURAL*∧
  params ⊂ *PARAMETER*∧
  priority ∈ *PRIORITY*∧
  scheduler = *NOT_STARTED*∧
THEN
  CHOICE
    handle ←
      t_create(*priority*) ||
    result := *pdPASS*
  OR
    result := *errMEMORY* ||
    handle :∈ *TASK*
  END
END

Figura 4.6: Especificação da operação *xTaskCreate*

4.3.2 Fila de mensagens

Nessa seção, será demonstrada parte da especificação relacionada ao elemento fila de mensagens. Essa especificação é feita através da máquina *Queue*. Nela, são tratadas principalmente as propriedades de tamanho de uma fila e conjunto de tarefas bloqueadas por uma fila. Aqui, os conjuntos também foram escolhidos como técnica para formalização dessas propriedades.

A especificação do estado da máquina *Queue* pode ser visto na figura 4.7. Nela, foram criadas as variáveis *queues*, *items*, *receiving*, e *sending*. A variável *queues* é responsável por armazenar todas as filas de mensagens criadas pelo sistema. A variável *items* relaciona uma fila de mensagens a um conjunto de itens (mensagens). Por último, as variáveis *receiving* e *sending* relacionam respectivamente uma fila de mensagens a um conjunto de tarefas bloqueadas aguardando a chegada de uma mensagem na fila (fila vazia) e um conjunto de tarefas bloqueadas aguardando para enviar uma mensagem para a fila (fila cheia).

Na máquina *Queue*, também foram criadas operações básicas para manipular os estados anteriormente descritos. Ao total seis operações foram desenvolvidas, entre elas a operação

VARIABLES	**INVARIANT**
queues,	$queues \in \mathbb{P}(QUEUE) \wedge$
items,	$items \in QUEUE \pfun \mathbb{P}(ITEM) \wedge \mathrm{dom}(items) = queues \wedge$
receiving,	$receiving \in QUEUE \pfun \mathbb{P}(TASK) \wedge \mathrm{dom}(receiving) = queues \wedge$
sending	$sending \in QUEUE \pfun \mathbb{P}(TASK) \wedge \mathrm{dom}(sending) = queues$

Figura 4.7: Estado da máquina *Queue*

$sendItem(queue, item, task, pos) =$
PRE
$queue \in queues \wedge$
$item \in ITEM \wedge$
$task \in TASK \wedge$
$pos \in COPY\_POSITION \wedge$
$task \in receiving(queue)$
THEN
$items(queue) :=$
$items(queue) \cup \{item\} \parallel$
$receiving(queue) :=$
$receiving(queue) - \{task\}$
END

Figura 4.8: Especificação da função *sendIten*

sendItem responsável por adicionar um item em uma fila de mensagens, demonstrada através da figura 4.8. Nela, os parâmetros de entrada *queue*, *item* e *task* representam respectivamente a fila que será manipulada, o item que será adicionado à fila e a tarefa que será retirada do conjunto de tarefas bloqueadas aguardando a chegada de uma mensagem na fila. Por fim, devido aos conjuntos não possuírem posições, o parâmetro *pos*, que representa a posição de entrada do item na fila (final ou começo), é declarado mas não é utilizado.

Prosseguindo na especificação, percebeu-se que as funcionalidades relacionadas à entidade fila de mensagens, possuem um comportamento muito parecido. Além disso, para a especificação dessas funcionalidades é necessário ter-se uma ligação com a entidade tarefa e fila de mensagens, pois uma fila é capaz de bloquear e desbloquear uma tarefa.

Nesse contexto, para especificar os comportamentos comuns das funcionalidades da fila de mensagens e fazer a ligação dessas funcionalidades com a entidade tarefa, foram criadas, na máquina *FreeRTOSBasic*, as seguinte funções intermediárias: *xQueueGenericSend* e *xQueueGenericReceived*. Essas funções são responsáveis respectivamente por: enviar um elemento para fila ou bloquear a tarefa remetente se a fila estiver vazia; receber um elemento da fila ou bloquear a tarefa receptora se a fila estiver cheia. A seguir, tem-se a especificação da operação *xQueueGenericSend*, ficando, a cargo do leitor interessado, verificar a especificação da operação *xQueueGenericReceived* no repositório do projeto.

Na função *xQueueGenericSend*, apresentada na figura 4.9, tem-se como parâmetros de entrada: a fila onde será enviada a mensagem, q ; o item que será enviado para a fila, i ; a quantidade de tempo que a tarefa remetente poderá ficar esperando pela fila, *wait*; e a posição que o item será colocado na fila, *pos*. Essa função possui dois comportamentos possíveis. No

```
res ← xQueueGenericSend(q, i, wait, pos) =
  PRE
    q ∈ queues ∧ i ∈ ITEM ∧ wait ∈ TICK ∧
    pos ∈ COPY_POSITION ∧
    active = TRUE ∧ running ≠ idle
  THEN
    CHOICE
      IF wait > 0 THEN
        q_insertTaskWaitingToSend(q, running) ||
        t_delayTask(wait) ||
        res := pdTRUE
      ELSE
        res := errQUEUE_FULL
      END
    OR
      ANY t WHERE
        t ∈ TASK ∧
        t ∈ blocked ∧
        t ∈ receiving(q)
      THEN
        q_sendItem(q, i, t, pos) ||
        t_unblock(t) ||
        res := pdPASS
      END
    END
  END
```

Figura 4.9: Especificação da função *xQueueGenericSend*

```
res ⟵ xQueueSend(q, i, w) =
  PRE
    q ∈ queues ∧ i ∈ ITEM ∧ w ∈ TICK ∧
    active = TRUE ∧ running ≠ idle
  THEN
    res ← xQueueGenericSend(q, i, w, queueSEND_TO_BACK)
    END
  END
```

Figura 4.10: Especificação da função *xQueueSend*

primeiro, representando uma fila cheia, ela bloqueia a tarefa remetente, através da operação *t_delayTask*, e, com o método *q_insertTaskWaitingToSend*, coloca-a no conjunto de tarefas que esperam para enviar um item para fila. No segundo, o item é enviado para a fila, através do método *q_sendItem*, e uma tarefa que aguarda por um item da fila é desbloqueada com a operação *t_unblock*.

Finalmente, para finalizar essa especificação inicial, a funcionalidade de enviar um item para a fila de mensagens, modelada na máquina *FreeRTOS*, é apresentada pela figura 4.10. Nela, basicamente, ocorre uma chamada à função *xQueueGenericSend*, da figura 4.9.

4.4 Refiando a especificação inicial

Para tratar dos requisitos do FreeRTOS ligados à prioridade de uma tarefa, é necessário acrescentar-se tal característica à modelagem inicial. Esse acréscimo está relacionado principalmente à máquina *Task* e pode ser feito de duas formas: escrevendo uma nova versão para essa máquina ou refinando-a. Aqui, a última forma foi escolhida como a mais adequada.

CONSTANTS
$MAX\_PRIO, IDLE\_P$
PROPERTIES
$PRIORITY = 0..(MAX\_PRIO - 1) \wedge$
$MAX\_PRIO > 0 \wedge IDLE\_P = 0$
VARIABLES
$prio$

INVARIANT
$prio \in TASK \rightarrow PRIORITY \wedge$
$\mathrm{dom}(prio) = tasks \wedge$
$(active = TRUE \Rightarrow$
$prio(idle) = IDLE\_P \wedge$
$\forall t.(t \in ready \Rightarrow prio(t) \leq prio(running)) \wedge$
$\forall t.(t \in ready \Rightarrow IDLE\_P \leq prio(t)))$

Figura 4.11: Especificação do estado do módulo $Task_r$

CONSTANTS
$schedule\_p$
PROPERTIES
$schedule\_p : (\mathbf{FIN}(TASK) \times (TASK \rightarrow PRIORITY)) \rightarrow \mathbf{FIN}(TASK) \wedge$
$schedule\_p = \lambda(tasks, prio) \bullet$
$(tasks : \mathbf{FIN}(TASK) \wedge prio : TASK \rightarrow PRIORITY \wedge tasks \neq \emptyset \wedge tasks \subseteq \mathrm{dom}(prio)$
$\mid \quad tasks \cap prio^{-1}(\max(prio[tasks]))))$

Figura 4.12: Especificação da função auxiliar *schedule_p*

O refinamento da máquina *Task* é feito pelo módulo *Task_r*, cujas variáveis e invariantes são exibidos na figura 4.11. Nela, o tipo *PRIORITY* representa a prioridade que uma tarefa pode assumir e a variável *prio* representa a ligação de uma tarefa a sua prioridade. Com esse refinamento, foi possível, através do seu invariante, especificar o seguinte requisito: uma vez inicializado o escalonador, a tarefa em execução deve ter prioridade maior ou igual a das tarefas de estado PRONTA.

Feita as alterações necessárias para tratar a prioridade de uma tarefa, é necessário agora refinar as operações da máquina *Task*. No entanto, para facilitar e modularizar o refinamento dessas operações, foi criada previamente a função auxiliar *scheduler_p*, que recebe como entradas um conjunto de tarefas e uma função que mapeia essas tarefas as suas prioridades. Ao final, essa função auxiliar retorna a tarefa de maior prioridade. A especificação de *scheduler_p* pode ser vista na figura 4.12.

Finalmente, os refinamentos das operações previamente demonstradas, *t_create* (figura 4.4) e *t_startScheduler*(figura 4.5), são exibidos, respectivamente, pelas figuras 4.13 e 4.14. Em *t_create*, o parâmetro *priority* é agora utilizado para indicar a prioridade da tarefa criada. Por fim, em *t_startScheduler*, a tarefa ociosa é criada com uma prioridade pré-definida e a escolha da tarefa que irá entrar em execução é feita através da tarefa de maior prioridade entre as de estado pronto, utilizando a função *scheduler_p*.

```
result ⟵ t_create(priority) =
PRE
  priority ∈ PRIORITY ∧
  running = TASK_NULL
THEN
  ANY task WHERE
    task ∈ TASK ∧ task ∉ tasks
  THEN
    tasks := tasks ∪ {task} ||
    prio := prio ∪ {task ↦ priority} ||
    ready := ready ∪ {task} ||
    result := task
  END
END
```

Figura 4.13: Especificação do refinamento da operação *t_create*

```
t_startScheduler =
  BEGIN
    active := TRUE ||
    blocked, suspended := ∅, ∅ ||
    ANY i WHERE
      i ∈ TASK ∧
      i ∉ tasks
    THEN
    tasks := tasks ∪ {i} ||
    prio := prio ∪ {i ↦ IDLE_P} ||
    idle := i ||
    ANY t WHERE
      t ∈ TASK ∧
      (ready = ∅ ⇒ t = i) ∧
      (ready ≠ ∅ ⇒ t ∈ ready ∧
        t ∈ schedule_p(ready, prio)
    THEN
      running := t ||
      ready := (ready ∪ {i}) − {t}
    END
  END
END
```

Figura 4.14: Especificação do refinamento da operação *t_startScheduler*

5 *Proposta*

O objetivo principal desse trabalho é especificar completamente o sistema operacional de tempo real FreeRTOS. Entretanto, como foi demonstrado no capítulo 4, parte dessa especificação já foi realizada. Portanto, para atingir tal objetivo, é necessário modelar ainda as entidades, funcionalidades e requisitos do sistema não tratados inicialmente.

Mais especificamente, na parte do sistema ainda não especificada tem-se: algumas funcionalidades das entidades tarefa e fila de mensagens; as entidades semáforo, mutex e co-rotina; e alguns requisitos do sistema que serão tratados após a especificação das entidades, com o amadurecimento da modelagem. Nas seções seguintes serão explicada, de forma detalhada, cada uma dessas pendências.

5.1 Entidade Tarefa

A entidade tarefa já foi parcialmente formalizada na modelagem inicial. Entretanto, muitas das suas funcionalidades não foram tratadas nessa especificação. Em suma, tem-se que:

- Na parte de criação de tarefa (seção 2.1.4), todas as suas funcionalidades foram formalizadas;
- Na parte de controle de tarefas (seção 2.1.4), apenas a funcionalidade *xTaskResumeFromISR()*, utilizada pelo tratamento de interrupções para reativar uma tarefa suspensa, não foi especificada;
- Na parte de utilitários de tarefa (seção 2.1.4), foram deixadas de lado as funcionalidades *uxTaskGetStackHighWaterMark()*, que retorna a quantidade de espaços vazios na pilha de uma tarefa, *vTaskSetApplicationTag()*, que associa uma função gancho a uma tarefa, e *xTaskCallApplicationTaskHook()*, que realiza a chamada a uma função gancho de uma determinada tarefa; e

- Na parte de controle de escalonador (seção 2.1.4) faltam especificar as funcionalidades *taskYIELD()*, funcionalidade que força a troca de contexto, *taskENTER_CRITICAL()*, funcionalidade que indica o início de uma região crítica, *taskEXIT_CRITICAL()*, funcionalidade que indica o final de uma região crítica, *taskDISABLE_INTERRUPTS()*, funcionalidade que desabilita o uso de interrupções, e *taskENABLE_INTERRUPTS()*, funcionalidade que habilita o uso de interrupções.

Vale ressaltar que existem funcionalidades, relacionadas à entidade tarefa, que foram tratadas na modelagem inicial e não serão listadas como proposta desse trabalho. Isso ocorrerá devido ao fato de que tais funcionalidades são utilizadas apenas como funções de rastreamento do sistema, coletando informações do funcionamento do sistema sem alterar o seu estado, sendo assim desnecessárias na implementação dos requisitos do sistema.

5.2 Entidade Fila de Mensagem

A entidade fila de mensagens também foi tratada na modelagem inicial. Nela, somente quatro de suas funcionalidades foram formalizadas, restando com isso cinco funcionalidades para completar a biblioteca referente à fila de mensagens. As funcionalidades pendentes são:

- *uxQueueMessagesWaiting()*, funcionalidade que retorna o número de mensagens armazenadas em uma fila;
- *xQueueSendFromISR()*, funcionalidade usada, no tratamento de uma interrupção, para enviar uma mensagem para uma fila;
- *xQueueSendToBackFromISR()*, funcionalidade usada, no tratamento de uma interrupção, para enviar uma mensagem para o final de uma fila;
- *xQueueSendToFrontFromISR()*, funcionalidade usada, no tratamento de uma interrupção, para enviar uma mensagem para o início de uma fila; e
- *xQueueReceiveFromISR()*, funcionalidade usada, no tratamento de uma interrupção, para receber uma mensagem de uma fila.

Assim como ocorreu com a entidade tarefa, existem funcionalidades sobre as filas de mensagens que não serão tratadas como proposta desse trabalho. Isso devido a essas funcionalidades serem usadas somente para captar informações do sistema, sendo assim desnecessárias para os requisitos do FreeRTOS.

5.3 Entidade Semáforo

Após a adição das funcionalidades que faltavam às entidades especificadas inicialmente, é necessário que as demais entidades do sistema sejam acrescentadas ao modelo. Esse acréscimo, assim como ocorreu com as entidades já tratadas, será feito de forma incremental, tornando com isso a especificação mais simples e viável.

A primeira entidade a ser acrescentada ao modelo será o semáforo. Inicialmente será especificado somente o semáforo binário, sendo o semáforo com contador especificado através de alterações no semáforo binário. A especificação desses elementos ocorrerá através da criação de um módulo com estados e operações responsáveis pelas características desses elementos. Com a isso, esse módulo servirá como base para a especificação das funcionalidades do sistema relacionadas à entidade semáforo.

As características da entidade semáforo inicialmente tratadas nessa etapa serão: estado de um semáforo (disponível ou indisponível); conjunto de tarefas bloqueadas à espera do semáforo; referência para a tarefa que retém o semáforo. Ao final dessa etapa, as seguintes funcionalidades irão enriquecer o modelo:

- vSemaphoreCreateBinary(), funcionalidade responsável por criar um semáforo binário no sistema;
- vSemaphoreCreateCounting(), funcionalidade responsável por criar um semáforo com contador no sistema;
- xSemaphoreTake(), funcionalidade utilizada por uma tarefa para solicitar um semáforo;
- xSemaphoreGive(), funcionalidade utilizada por uma tarefa para liberar um semáforo;
- xSemaphoreGiveFromISR(), funcionalidade utilizada pelo tratamento de uma interrupção para liberar um semáforo; e
- xSemaphoreTakeFromISR(), funcionalidade utilizada pelo tratamento de uma interrupção para solicitar um semáforo.

5.4 Entidade Mutex

A criação da entidade mutex (seção 2.2.3) ocorrerá após a especificação do semáforo devido ao fato de o mutex ser considerado um tipo especial de semáforo, no qual o mecanismo

de herança de prioridade é implementado. Assim, para adicionar a entidade mutex na modelagem do FreeRTOS será necessário apenas editar o módulo semáforo para que ele trate das particularidades relacionadas a essa entidade.

As características da entidade mutex que serão tratadas inicialmente são: o estado de um mutex (disponível ou indisponível), o conjunto de tarefas aguardando o mutex; referência para a tarefa que retém o mutex; e o mecanismo de herança de prioridade. Ao final, a funcionalidade *xSemaphoreCreateMutex*(), responsável por criar um mutex no sistema, será acrescentada à modelagem do FreeRTOS.

5.5 Entidade Co-rotina

A última entidade a ser modelada na especificação do FreeRTOS será a co-rotina (seção 2.1.2). Para formalizar essa entidade, será preciso criar-se um novo módulo, com os estados e operações responsáveis pela especificação das características dessa entidade. Além disso, devido a co-rotina também ser uma unidade de execução do sistema, ela é controlada pelo escalonador, o qual deverá ter suas funcionalidades alteradas para suportar a entidade em questão.

A característica inicialmente tratada na especificação da entidade co-rotinas será o estado que uma co-rotina pode assumir. Com a modelagem dessa entidade, serão adicionadas as seguintes funcionalidades à especificação:

- xCoRoutineCreate(), funcionalidade responsável pela criação de uma co-rotina no sistema;
- crDELAY(), funcionalidade utilizada para bloquear uma co-rotina;
- crQUEUE_SEND(), funcionalidade utilizada por uma co-rotina para enviar uma mensagem para uma fila de mensagens;
- crQUEUE_RECEIVE(), funcionalidade utilizada por uma co-rotina para receber uma tarefa de uma fila de mensagens;
- crQUEUE_SEND_FROM_ISR(), funcionalidade utilizada pelas co-rotinas que tratam interruções para enviar uma mensagem para uma fila de mensagens;
- crQUEUE_RECEIVE_FROM_ISR(), funcionalidade utilizada pelas co-rotinas que tratam interruções para receber uma mensagem de uma fila de mensagens; e

- vCoRoutineSchedule(), funcionalidade utilizada para chamar o escalonador responsável pelas co-rotinas.

5.6 Requisitos do sistema

Com a modelagem das entidades, parte dos requisitos do sistema serão tratados. Entretanto, como essa modelagem será realizada de forma abstrata, vários requisitos não poderão ser especificados logo de início. Assim, para que esses requisitos abstraídos sejam formalizados, é necessário adicionar à especifica as características subtraídas das entidades na modelagem inicial. Essa adição pode ocorrer de duas formas, reformulando o modelo criado ou refinando-o.

A seguir, tem-se a lista de todos os requisitos do sistema que serão tratados nessa fase. Uma explicação mais detalhada sobre esses requisitos será mostrada nos parágrafos seguintes:

- Compartilhar o tempo de CPU entre tarefas de iguais prioridade.
- Tamanho de uma fila de mensagens.
- Tempo de bloqueio de uma tarefa.
- Tratamento de *overflow* do tempo de execução.
- Ordenação dos itens em uma fila de mensagens.
- Prioridade entre tarefas nas listas de eventos.
- Gerenciamento de memória.

O compartilhamento do tempo de CPU entre tarefas de mesma prioridade será o primeiro requisito especificado nessa etapa da modelagem. Para tratar tal exigência, serão necessárias algumas alterações no módulo responsável pelo a entidade tarefa. Essas alterações tornarão esse módulo capaz de controlar o tempo de execução entre as tarefas de estado PRONTA e com prioridades iguais a tarefa em execução. Isso é importante porque evita que tarefas de maiores prioridades nunca sejam escalonadas. Ao final, a funcionalidade *incrementTick* deve ser alterada para compartilhar o tempo de execução entre tarefas de mesma prioridade.

A característica de tamanho de uma fila de mensagens define uma capacidade máxima para uma fila de mensagens. Esse requisito será tratado com o refinamento do módulo de fila de mensagens, adicionando a ele estados e operações capazes de controlar o tamanho de uma fila.

Após isso, as funcionalidades que estão relacionadas a essa característica serão refinadas para tratar tal exigência.

Outra particularidade do sistema não tratada nas demais etapas é a característica de uma tarefa possuir um tempo máximo de bloqueio[1]. Para tratar dessas características será adicionado, no módulo de tarefas, um estado para controlar o tempo de bloqueio das tarefas. Com isso, as funcionalidades relacionadas a essa característica devem ser alteradas para tratar tal particularidade.

O estouro do tempo de execução acontece devido às limitações da máquina utilizada pelo sistema. Esse fato ocorre quando o tempo de execução do sistema ultrapassa o limite máximo do tipo que o representa. Para tratar esse acontecimento, o sistema basicamente maneja as tarefas bloqueadas atualizando o seu tempo de desbloqueio a cada estouro. Com isso, a especificação dessa exigência será feita através de alterações no módulo de tarefa e, consequentemente, nas funcionalidades relacionadas a essa entidade, além do refinamento do modulo que preocupa-se com o tempo do escalonador.

Ordenar os itens em um fila de mensagens é implementar, na fila de mensagens, uma estrutura, na qual cada item tem sua posição. Essa característica será adicionada ao modelo através do refinamento do módulo de fila, proporcionando uma estrutura ordenada para os itens de uma fila. Após isso, algumas funcionalidades que cuidam dessa entidade serão refinadas.

O requisito tratar a prioridade das tarefas em uma fila de evento significa que, no momento de desbloquear uma tarefa localizada em uma fila de eventos, será levando em consideração a prioridade das tarefas localizadas nela. Basicamente, as tarefas com maiores prioridades serão desbloqueadas primeiro do que as tarefas de menores prioridades. Esse requisito será especificado através de alterações nas funções que trabalham com fila de evento, como é o caso das funcionalidades que enviam e recebem uma tarefa para uma fila de mensagens, pois são nessas funcionalidades que as tarefas bloqueadas aguardando pela fila são desbloqueadas e retiradas das filas de evento.

Por fim, o requisito de tratamento de memória deve ser incorporado ao sistema. Assim, no momento da criação de cada entidade, a especificação poderá decidir se há, ou não, memória suficiente para a criação da entidade, retirando com isso o indeterminismo existente nas funcionalidades de criação de uma entidade.

[1]Na modelagem inicial da entidade tarefa foi tratada apenas a característica de uma tarefa possuir o estado BLOQUEADA e não o tempo que essa tarefa deve permanecer nesse estado

5.7 Atividades e Etapas

Para o melhor entendimento do cronograma do projeto, que será apresentado na seção 5.8, as atividades discutidas nas seções anteriores foram resumidamente elencadas em etapas a serem realizadas nesta seção. Através desse trabalho obteve-se a seguinte lista:

1. Especificar, em um nível abstrato, as funcionalidades restantes das entidades tarefa e fila de mensagens.
    - Nessa etapa, serão desenvolvidas as funcionalidades restantes relacionadas às entidades tarefa e fila de mensagens, especificadas inicialmente.
2. Modelagem abstrata da entidade semáforo.
    - Nessa etapa, a entidade semáforo (seção 2.2.2) será especificada de forma abstrata, junto com as funcionalidades relacionadas a essa entidade.
3. Modelagem abstrata da entidade mutex.
    - Nessa parte do trabalho será especificado, de forma abstrata, a entidade mutex (seção 2.2.3), assim como as funcionalidades que estão relacionadas a essa entidade.
4. Modelagem abstrata da entidade co-rotina.
    - Essa fase do trabalho encerra a especificação abstrata das entidades do FreeRTOS. Nela, a entidade co-rotina(seção 2.1.2) será modelada, em um nível abstrato, junto com suas funcionalidades.
5. Refinamento da entidade tarefa.
    - Nessa etapa, a entidade tarefa, especificada inicialmente, será refinada. Parte dessa etapa já foi iniciada nos primeiros passos desse trabalho(capítulo 4). Assim, resta apenas especificar as características de tempo bloqueio de uma tarefa, controle da execução de duas tarefas com a mesma prioridade, controle do estouro do tempo de execução e refinar as estruturas abstratas usada na especificação dessa entidade para uma forma mais concreta.
6. Refinamento da entidade fila de mensagens.

- Como a entidade fila de mensagens foi especificada de forma abstrata inicialmente, essa etapa será responsável por realizar uma modelagem mais concreta dessa entidade. Nessa etapa, serão tratadas as características de posição das mensagens e prioridade das tarefas em uma fila de evento.

7. Refinamento da entidade co-rotina.

   - Nessa fase ocorrerá o refinamento da entidade co-rotina, especificando as características de prioridade de uma co-rotina e tempo máximo de bloqueio de uma co-rotina.

8. Refinamento da entidade semáforo.

   - Nessa parte do trabalho, a entidade semáforo será refinada. Através desse refinamento serão tratados os requisitos ligados ao semáforo com contador.

9. Refinamento da entidade mutex.

   - Essa etapa encerra a especificação do FreeRTOS com o refinamento da entidade semáforo. Nela o tratamento de herança de prioridade será acrescentado à especificação.

10. Preparar dissertação e defesa

    - Depois do desenvolvido do trabalho proposto, é hora de preparar a dissertação e finalmente o material para a defesa , sendo este o trabalho dessa etapa.

5.8 Cronograma

Para planejar o desenvolvimento das etapas elencadas acima, foi preparado o cronograma da tabela 5.1, que prevê a finalização do projeto em 3 meses após a qualificação. Nele, as etapas de especificação abstratas das entidades e o refinamento da entidade tarefa são realizadas até o final do mês de novembro, restando o mês de dezembro para as demais atividades de refinamento. Em janeiro, a dissertação, junto com sua defesa, será elaborada, finalizando o projeto.

Etapas	**Novembro**	**Dezembro**	**Janeiro**
Etapa 1			
Etapa 2			
Etapa 3			
Etapa 4			
Etapa 5			
Etapa 6			
Etapa 7			
Etapa 8			
Etapa 9			
Etapa 10			

Tabela 5.1: Cronogramas de etapas do projeto

6 Conclusão

Esse trabalho baseia-se em dois grandes desafios da computação, o desenvolvimento de sistemas fidedignos e, principalmente, o projeto do software verificado, no qual se encontra a proposta de especificar formalmente o sistema operacional de tempo real FreeRTOS, objetivo principal desse trabalho. Com essa especificação, será possível provar o correto funcionamento das funcionalidades que constituem esse sistema e assim garantir uma maior confiabilidade para as aplicações desenvolvidas a partir dele.

A escolha do FreeRTOS como sistema a ser especificado ocorreu devido as suas características. Primeiramente ele é um sistema simples e enxuto, o que facilita a sua formalização. Em segundo, ele possui uma grande popularidade, portabilidade e apresenta, em seu código fonte, conceitos comuns à maioria dos sistemas atuais, o que torna a sua especificação uma grande ajuda para a comunidade da computação.

O formalismo escolhido para realizar tal especificação foi o método B, que proporciona uma modelagem incremental e modular. Assim, essa especificação será realizada por etapas, nas quais serão gerados modulos abstratos com tarefas bem definidas, responsáveis por partes da especificação. Em seguida, utilizando o mecanismo de refinamento, os módulos criados serão especificados de forma mais concreta, tornando assim a especificação mais completa e próxima à realidade.

Para provar a viabilidade desse trabalho, foi desenvolvido, como etapa inicial, uma modelagem abstrata de alguns dos principais conceitos do FreeRTOS. Nessa etapa, os conceitos de tarefa e fila de mensagens foram especificados de forma abstrata e, em seguida, refinados. O resultado dessa etapa foi uma especificação simples, mas com várias características do FreeRTOS provadas. Como fruto desse trabalho foi publicado o seguinte artigo [15].

Outra utilidade da especificação do FreeRTOS será a criação de uma documentação do sistema na perspectiva formal, o que servirá para um melhor entendimento do sistema. Além disso, essa especificação formal poderá ser usada para verificar a implementação atual usando ferramentas como FRAMA-C[26] ou VCC[27]. Por fim, essa modelagem formal poderá servir

como entrada para a criação de testes de sistema a nível de codificação, criando-se assim mais uma forma de verificação do correto funcionamento do FreeRTOS.

Por último, como continuidade desse trabalho tem-se, através das ferramentas disponibilizadas pelo método B, a possibilidade da geração de código a partir da especificação criada. Para isso será necessário realizar-se refinamentos mais concretos a nível de linguagem algorítmicas e em seguida a adaptar tal modelagem para o uso das ferramentas, que geralmente não aceitam determinados tipos de construção da especificação em B.

Referências Bibliográficas

[1] David Kalinsky. *Basic concepts of real-time operating systems*. 2003.

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

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

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

[5] COMMITTEE, P. I. T. A. *Computational Science: Ensuring America's Competitiveness*. 2005. URL:http://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf.

[6] SBC. *Grandes Desafios da Pesquisa em Computacão no Brasil 2006 ·*2016.[*S.l.*], 2006.

[7] Jim Woodcock et al. Formal methods: Practice and experience. In: ACM (Ed.). *ACM Computing Surveys*. [S.l.: s.n.], 2009. v. 5, p. 1–40.

[8] J.C. Bicarregui; C.A.R. Hoare; J.C.P. Woodcock. The verified software repository:a step towards the verifying compiler. In: *Formal Aspects of Computing*. [S.l.]: Springer, 2008. v. 1, p. 277â280.

[9] Tony Hoare; Jay Misra. *Verified software: theories, tools, experiments*. 2005. URL:http://vstte.ethz.ch/pdfs/vstte-hoare-misra.pdf.

[10] SCHNEIDER, S. *The b-method: an introdution*. [S.l.]: Palgrave, 2001.

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

[12] E. Jaffuel; B. Legeard. Leirios test generator: Automated test generation from b models. In: *The 7th International B Conference*. [S.l.: s.n.], 2007. p. 277â280.

[13] Valério Medeiros Jr.; Stephenson Galvão. Modelagem de micro controladores em b. In: *Anais do VIII Encontro Regional de Matemática Aplicada e Computacional (ERMAC)*. [S.l.]: EUFRN, 2008. v. 1.

[14] Bartira Dantas et al. Applying the b method to take on the grand challenge of verified compilation. In: *Brazilian Symposium on Formal Methods (SBMF2008) - Proceedings*. [S.l.]: Editora Gráfica da UFBA - EDUFBA, 2008. v. 1.

[15] David Déharbe; Stephenson Galvão; Anamaria Moreira. Formalizing freertos: First steps. In: *Brazilian Symposium on Formal Methods (SBMF2009) - Proceedings*. [S.l.]: Editora Gráfica da UFRGS - EDUFRGS, 2009. v. 1.

[16] Moritz Kleine; Steffen Helke. Météor: A successful application of b in a large project. In: *World Congress on Formal Methods*. [S.l.]: Springer, 1999. v. 1708.

[17] Roger Solá et al. *Microkernel API Formal Modelisation*. [S.l.]: SAME 2005 Forum, 2005.

[18] Iain D. Craig. *Formal Models of Operating System Kernels*. 1. ed. [S.l.]: Springer, 2006.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. 1. ed. [S.l.]: Prentice Hall International, 2004.

[20] J.-R. Abrial. *Event-B Language*. 1. ed. [S.l.]: Prentice Hall International, 2004.

[21] Jim Woodcock; Jim Davies. *Using Z Specification, Refinement, and Proof*. 1. ed. [S.l.]: Prentice Hall, 1995.

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

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

[24] P. Behm et al. Low-level code verification based on csp models. In: *Brazilian Symposium on Formal Methods (SBMF2009) - Proceedings*. [S.l.]: Editora Gráfica da UFRGS - EDUFRGS, 2009. v. 1.

[25] ClearSY. *Atelier B 4.0*. 2009. URL:http://www.atelierb.eu.

[26] CEA LIST. *Frama-Câs value analysis plug-in*. 2009. URL:http://frama-c.cea.fr/.

[27] Markus Dahlweid MichaÅ Moskal, T. S. *VCC: Contract-based Modular Verification of Concurrent C*. [S.l.], 2008.