MOS: UM SISTEMA OPERACIONAL E FRAMEWORK MULTIPLATAFORMA E MULTIPROCESSAMENTO BASEADO EM MÓDULOS

Alex Fernandes Neves

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Ramon Romankevicius Costa

Rio de Janeiro
Dezembro de 2013

MOS: UM SISTEMA OPERACIONAL E FRAMEWORK MULTIPLATAFORMA E MULTIPROCESSAMENTO BASEADO EM MÓDULOS

Alex Fernandes Neves

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Ramon Romankevicius Costa, D.Sc.

Prof. Fernando Cesar Lizarralde, D.Sc.

Prof. Walter Fetter Lages, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 2013

Neves, Alex Fernandes

MOS: Um Sistema Operacional e Framework Multiplataforma e Multiprocessamento Baseado em Módulos/Alex Fernandes Neves. – Rio de Janeiro: UFRJ/COPPE, 2013.

XIV, 165 p.: il.; 29, 7cm.

Orientador: Ramon Romankevicius Costa

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2013.

Referências Bibliográficas: p. 161 – 165.

1. Robótica. 2. Sistema Operacional. 3. Microcontrolador. I. Costa, Ramon Romankevicius. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Dedico este trabalho à minha
família

Agradecimentos

Gostaria de agradecer o meu orientador Ramon Romankevicius Costa pela ajuda na dissertação, aos membros da banca Fernando Cesar Lizarralde e Walter Fetter Lages pelo interesse no trabalho, aos meus familiares e amigos pelo apoio durante o Mestrado e aos meus chefes e colegas de trabalho pela compreensão.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

MOS: UM SISTEMA OPERACIONAL E FRAMEWORK MULTIPLATAFORMA E MULTIPROCESSAMENTO BASEADO EM MÓDULOS

Alex Fernandes Neves

Dezembro/2013

Orientador: Ramon Romankevicius Costa

Programa: Engenharia Elétrica

O propósito desta dissertação é desenvolver o Modular Operating System (MOS), um sistema operacional multiplataforma baseado em módulos, e sua linguagem de programação Modular Operating System Language (MOSLanguage). Este sistema operacional tem por objetivo permitir que dispositivos com este mesmo sistema comuniquem entre si e com isto troquem programas especiais chamados de Module. Esta abordagem foi usada para tentar solucionar problemas no desenvolvimento de aplicações que necessitam de múltiplas unidades de processamento.

A abordagem usada na dissertação indica os pontos que o MOS deseja atingir enquanto expõe o funcionamento do sistema. Então a linguagem MOSLanguage é apresentada, mostrando a relação entre os principais elementos do MOS. Por último, as estruturas em C++ que representam o sistema operacional são detalhadas já considerando os microprocessadores com Windows ou Linux, e os microcontroladores AVR (RISC modificado) em que o MOS deverá inicialmente funcionar.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

MOS: A MULTIPLATAFORM AND MULTIPROCESSING OPERATING SYSTEM AND FRAMEWORK BASED ON MODULES

Alex Fernandes Neves

December/2013

Advisor: Ramon Romankevicius Costa

Department: Electrical Engineering

The purpose of the dissertation is to develop the Modular Operating System (MOS), an multiplataform operating system based in modules, and his programing language Modular Operating System Language (MOSLanguage). The objective of this operating system is to allow devices with the this operating system communicate with each other and then exchange programs called Module. This aproach has been used to try to solve problems during the development of application that mey use multiple processing unities.

The dissertation indicates the points that MOS should hit while exposes how the system works. Then the MOSLanguage is presented, showing the relation between MOS's primary elements. At last, the C++ structures that represent the OS are detailed considering microprocessors with Windows or Linux and microcontrollers AVR (modified RISC) in which MOS should function.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

Este trabalho trata do desenvolvimento de um Sistema Operacional e Software Framework apelidado de Module Operating System (MOS). O tipo de aplicação em que o MOS melhor se encaixa é o que envolve o uso de múltiplas unidades de processamento. Sua funcionalidade é equivalente ao de outros Frameworks com o mesmo fim, mas utiliza uma abordagem diferente na sua concepção e uso. Isto se resume em dois itens: no uso de uma linguagem própria chamada de Module Operating System Language (MOSLanguage) para definir Modules, e na capacidade de compilação, transferência e execução remota de Modules.

O trabalho não propõe expor o código do MOS em si, mas mostrar como deve funcionar. A implementação deve seguir este trabalho como roteiro. O conceito do MOS foi desenvolvido pelo próprio autor baseado em sua visão de Framework para o tipo de aplicação mencionado.

1.1 Objetivos

O trabalho tem como principais objetivos:

- Apresentar o Module Operating System (MOS)
- Indicar os principais elementos do MOS
- Compará-lo com outros Sistemas/Frameworks (vantagens e limitações)
- Mostrar a linguagem MOSLanguage
- Definir Modules
- Explicar as conexões entre Modules
- Relação de Modules, threads e sincronização

- Detalhar procedimentos de compilação e ligação de Modules
- Como MOS abstrai redes de comunicação
- Apresentar a rede MOSNetwork
- Mostrar a arquitetura de software do MOS com classes para C++ (exemplifica como o MOS pode ser implementado)
- Explicar a adaptação do MOS para dispositivos

1.2 Motivação

O Module Operating System (MOS) e a linguagem Module Operating System Language (MOSLanguage) são uma possível solução para dificuldades encontradas em aplicações que envolvem o uso de múltiplos centros de processamento. As escolhas iniciais em projetos destas aplicações envolvem os processadores e/ou microcontroladores a serem usados, o sistema operacional nos microprocessadores, as linguagens de programação, pacotes/bibliotecas, os meios de comunicação e os dispositivos extras que deverão ser ligados aos centros de processamento.

Os programadores têm a responsabilidade de integrar os componentes do projeto por software. O objetivo do MOS é criar ferramentas para facilitar o desenvolvimento dessa etapa. Um exemplo de problema na programação é a não disponibilidade do *hardware* no momento em que o software deve começar a ser desenvolvido para atender o cronograma do projeto. Outro exemplo de problema é a necessidade do *software* ser testado em múltiplas máquinas ao mesmo tempo durante sua criação. Dependendo da aplicação usa-se microcontroladores ao invés de microprocessadores e, neste caso, o ambiente de trabalho e a ausência de um sistema operacional torna o desenvolvimento relativamente diferente comparado ao caso em que se emprega microprocessadores.

Para entender melhor a motivação para o desenvolvimento do MOS, três casos de aplicações que envolvem múltiplos centros de processamento serão brevemente analisados. Em seguida algumas soluções (programas de computador, bibliotecas, etc) serão apresentadas tendo alguma relação com as três aplicações. Com isto será possível determinar os pontos em que o MOS e o MOSLanguage atuarão, mostrando sua utilidade e características básicas.

1.2.1 Exemplos de Aplicações Com Múltiplas Unidades de Processamento

Exemplo 1. Aplicativos Clientes-Servidores

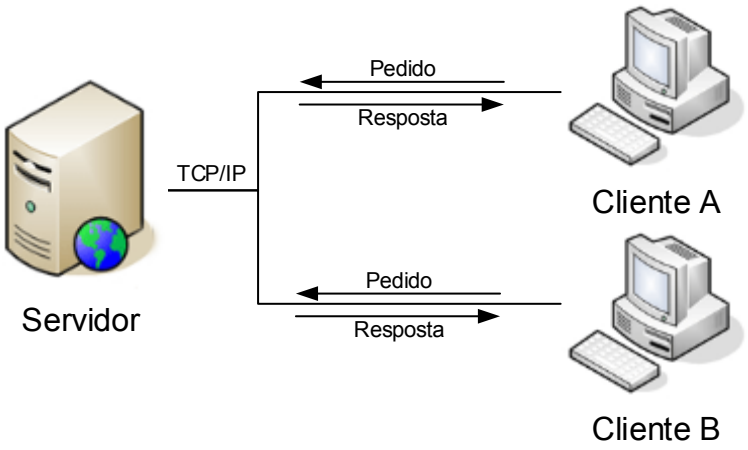

Figura 1.1: Os Clientes A e B se conectam com o Servidor através de uma conexão pelo protocolo TCP/IP. Cada cliente é capaz de enviar mensagens para o Servidor. O Servidor posteriormente retorna uma mensagem de resposta ao pedido. Nesta aplicação não existe uma relação explícita entre os clientes.

A relação cliente-servidor descreve um modelo de aplicações em que um cliente requisita uma ação ou serviço para um provedor de serviço, o servidor. A Figura 1.1 representa esta aplicação.

Este esquema pode ser adaptado para o caso de múltiplos servidores, em que não há mudança aparente para os clientes, no entanto, os servidores devem estar preparados para conversar entre si. Isto ocorre quando a operação é muito pesada ou o número de clientes a serem conectados ao servidor é muito grande e é preciso dividir as tarefas entre múltiplos servidores. A Figura 1.2 ilustra esta aplicação.

Nesta aplicação não existe uma relação explícita entre os clientes. Os servidores devem se comunicar, seja para acessar dados armazenados em outro servidor ou para manter dados idênticos entre os servidores.

O modelo de comunicação cliente-servidor é o exemplo mais básico encontrado na relação entre computadores. Neste trabalho computadores serão vistos como um tipo de centro de processamento. O modelo cliente servidor não se limita à relação entre computadores, visto que os programas usados nos computadores utilizam o protocolo TCP/IP [5][6], que por sua vez utiliza o modelo cliente-servidor como modelo de conexão.

Exemplo 2. Supercomputador

Supercomputador é um conjunto de processadores que podem ser usados em conjunto para realizar tarefas de forma eficiente e mais rápido do que feito por um único computador. Existem abordagens diferentes na arquitetura física de um supercomputador, sendo classificadas em geral na Grid Computing [7] (os computadores podem estar dispersos geograficamente, com nós voláteis) e na Computer Cluster (múltiplas CPU próximas entre si). Não há restrição na rede que conecta

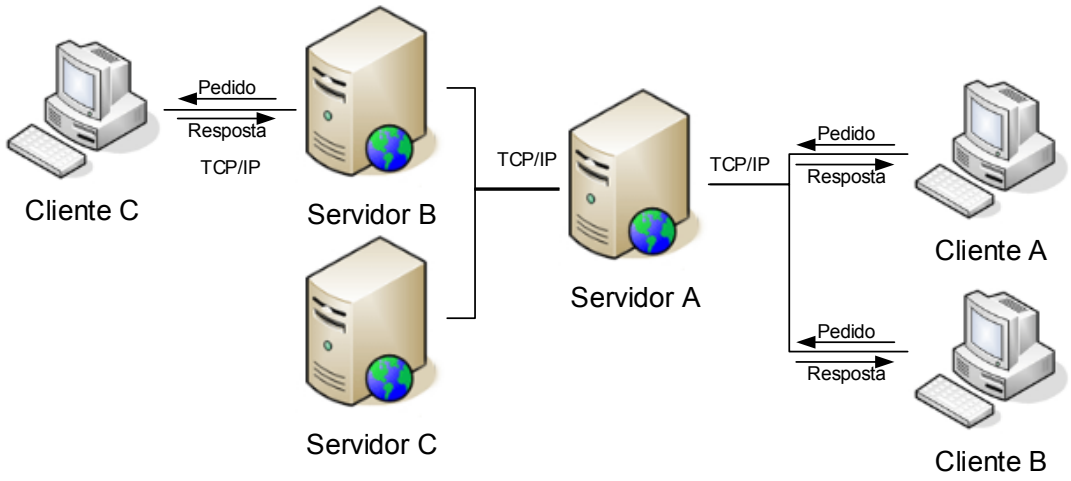

Figura 1.2: Os Clientes A e B se conectam com o Servidor A, o Cliente C se conecta ao Servidor B e os Servidores A, B e C conectam-se entre si. Todas as conexões são feitas através de conexões TCP/IP. Cada cliente é capaz de enviar mensagens para o servidor com que se conecta. Os servidores posteriormente retornam mensagens de resposta do pedido.

os computadores, nas CPUs ou no sistema operacional a ser usado, mas as escolhas influenciam na capacidade do supercomputador. A figura 1.3 representa um Computer Cluster.

Primeiramente os programas devem ser escritos para serem eficientes dentro de uma CPU. Também devem estar preparados para executar operações em computadores diferentes e comunicarem entre si para agregar os resultados. Por último os programas devem ser enviados para estes computadores e executados.

Um exemplo de uso de um supercomputador é a otimização de uma função não linear de múltiplas variáveis. É possível que o algoritmo de otimização encontre soluções localizados em mínimos locais, enquanto o que se deseja é o mínimo global. Neste caso, uma solução é executar o algoritmo para condições iniciais diferentes, e cada condição inicial poderia ser executada em um computador diferente. Em seguida, o usuário poderia comparar os resultados e tentar descobrir o mínimo global.

Exemplo 2. Robótica

A definição de um robô de acordo com a Robot Institute of America (1979) é: "A reprogrammable, multifunctional manipulator designed to move material, parts, tools, or specialized devices through various programmed motions for the performance of a variety of tasks" (Um robô é um manipulador reprogramável e multifuncional projetado para mover materiais, partes, ferramentas ou dispositivos especializados por uma série de movimentos programados para a realização de uma variedade de tarefas).

Um robô é constituído de um conjunto de sensores (enxergam o ambiente), atu-

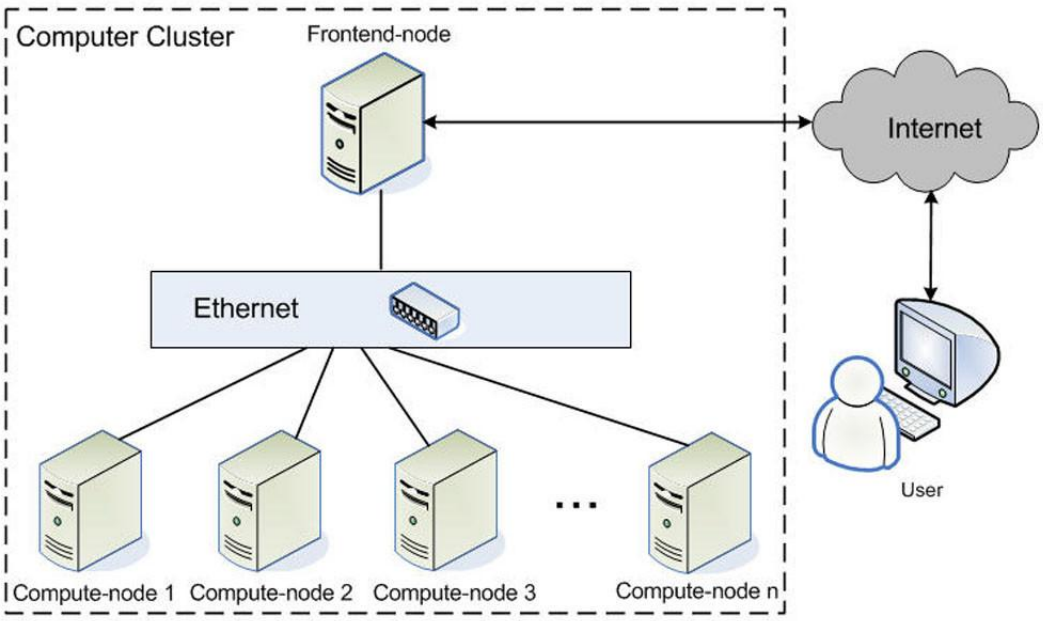

Figura 1.3: Imagem retirada de [1]. O User (usuário) acessa um dos nós do supercomputador pela internet. Os nós do Cluster estão conectados por uma rede ethernet, permitindo que o User acesse qualquer nó.

adores (interagirem com o ambiente) e Unidades de Processamento (decidem como será a interação). O termo Processing Unit (PU) ou, em português, Unidade de Processamento será usado neste trabalho para representar microcontroladores e microprocessadores em sua capacidade de execução.

Robôs não possuem uma padronização. Cada aplicação exige tipos e quantidades diferentes de sensores e atuadores pelo ambiente de operação do robô. Para cada tipo existem inúmeros fabricantes no mercado. Também existem inúmeras opções de arquitetura de hardware para as PUs. Além de poder variar em número, as PUs são classificadas entre microprocessadores (com seus periféricos) e microcontroladores, cada um com uma infinidade de fabricantes e modelos. As arquiteturas devem permitir a aquisição de dados de sensores e permitir que os atuadores sejam controlados pelas PUs. A arquitetura também envolve os tipos de redes que criam a conexão entre processadores, sensores e atuadores.

Sensores podem ser separados em dois tipos básicos, analógicos e digitais. Quando digital, existe alguma eletrônica associada ao sensor que lê os sinais analógicos e os digitaliza. Esta eletrônica possui entradas/saídas digitais que permitem conectar o sensor a determinadas redes. Algumas redes tradicionais para sensores são I2C [8] e SPI [9]. Quando o sensor é analógico alguma eletrônica pode ser usada para, por exemplo, filtrar ou amplificar sinais analógicos, mas eventualmente o sinal é digitalizado e repassado para uma PU.

Atuadores possuem um aspecto dual sobre os sensores na relação de sinais analó-

gico/digital. O sinal para os atuadores pode ser gerado de uma conversão do digital para o analógico. Para gerar a saída analógica podem ser usados controladores digitais comerciais (possuem uma PU) ou microcontroladores. No caso do uso de microcontroladores é comum existir uma eletrônica para amplificar/modular a potência da saída analógica do microcontrolador. Também existe a opção de usar uma saída PWM [10] ao invés de uma saída analógica de um microcontrolador. A saída de controle do atuador deve ser programada no microcontrolador. Os controladores comerciais utilizam redes para que sejam configurados/comandados externamente por microcontroladores, ou por microprocessadores e seus periféricos. Algumas das redes clássicas seriam as seriais RS232 e RS485 [11], e a CAN (Controller Area Network) [12]. Controladores puramente analógicos de atuadores têm sido usados cada vez menos pela digitalização possibilitar o uso de algoritmos mais complexos.

Inúmeras arquiteturas de hardware podem ser implementadas em um robô. A arquitetura mais simples para um robô é aquela que concentra todo o processamento, e também todos os sinais dos sensores e atuadores, em um único microcontrolador. Ao invés de um microcontrolador poderia ser usado um computador que tivesse recursos para fazer a aquisição e para gerar sinais analógicos.

Por inúmeras razões, a arquitetura pode envolver mais de uma PU (computador ou microcontrolador). Por exemplo, quando existe um grande número de sensores/atuadores em um robô, um microcontrolador pode não ter o número de portas necessárias para lidar com todos os sinais analógicos, tornando necessário o uso de múltiplos microcontroladores. Outro caso é um microcontrolador não ter a capacidade de processamento necessária para o robô, tornando necessário o uso de mais um microcontrolador/processador. Nesta situação deve-se permitir que os microcontroladores estejam conectados. Esta conexão pode ser feita através de uma rede como as mencionadas anteriormente.

Além de múltiplas PUs, podem existir outros equipamentos que necessitam se comunicar por redes. Por exemplo, câmeras que devem ser conectadas a um computador por USB ou TCP/IP.

Com isto, um robô passa a ser um conjunto de PUs que devem se comunicar por certo número de redes. Cada microcontrolador ou família de microcontroladores (varia com fabricantes) pode ter uma abordagem diferente para lidar com cada rede. Alguns possuem APIs (Application Programing Interface) ou bibliotecas para lidar com as redes. Outros utilizam registradores especiais para indicar quais pinos estão sendo usados para acessar uma rede. Por exemplo, muitos microcontroladores utilizam módulos USART (Universal Synchronous Asynchronous Receiver Transmitter) [13] que devem ser configurados para acessar redes RS232, RS422 e RS485. A USART envolve uma série de registradores que variam com o fabricante do microcontrolador.

Sistemas operacionais têm certa vantagem em relação aos microcontroladores por possuir APIs que isolam o hardware do programador. Além disto, sistemas operacionais permitem processamento paralelo (permitem o uso de threads) enquanto microcontroladores permitem programas seriais (não há threads), contudo, microcontroladores usam interrupções [13] que são essenciais para certas aplicações. Existem implementações simples de sistemas operacionais para microcontroladores, permitindo a execução paralela de tarefas. Muitos sistemas operacionais não possuem Núcleo (ou Kernel) em tempo real [14], não sendo indicados para muitas aplicações.

Com a arquitetura definida, cabe ao programador desenvolver protocolos de comunicação entre os centros de processamento. Caso os dispositivos na rede não possuam APIs para acessá-las, o programador deve reproduzir o protocolo descrito nos manuais dos dispositivos para gerar mensagens a serem transferidas na rede ou traduzir mensagens recebidas pela rede.

1.2.2 Soluções Atuais

O desenvolvimento de uma aplicação com múltiplas PUs implica em inúmeras dificuldades que não são encontradas nos casos em que há somente uma PU. Entre estes estão o controle da transferência e execução de programas entre PUs, os testes a serem executados em múltiplas PUs ao mesmo tempo e o desenvolvimento do programa para lidar com a comunicação em alguma rede.

Naturalmente surgiram ferramentas para facilitar este desenvolvimento, algumas largamente usadas na indústria. A seguir serão apresentados alguns itens com suas respectivas abordagens para facilitar o desenvolvimento de aplicações em múltiplas PUs.

Acesso Remoto

Existem muitas formas de acessar remotamente um sistema operacional. Um dos mais usados é o Secure Shell (SSH) [15], um protocolo de comunicação que permite um acesso remoto e seguro a um usuário. Ele permite que comandos sejam executados no computador em que a conta de usuário se encontra, e permite o acesso a arquivos do computador. É tipicamente usado em sistemas baseados em UNIX.

Também existem os Remote Desktop Software (programas gráficos ou de linhas de comando que permite um computador acessar outro remotamente, usando ou não SSH). O Netviwer, TightVNC, o TeamViewer e o Windows Remote Desktop Connection são exemplos de softwares que funcionam em muitos sistemas operacionais. Muitos destes permitem o compartilhamento da tela de usuário, do teclado e do mouse. Vale ressaltar que os softwares variam na compatibilidade com sistemas operacionais em que podem ser usados.

Isto ajuda os programadores a trabalharem remotamente na execução e teste de

programas, e na transferência de arquivos. No entanto, os Remote Desktop Software não auxiliam o programador na etapa de programação.

Supercomputadores

Message Passing Interface

Message Passing Interface (MPI) [16] é um protocolo de comunicação usado para programar computadores paralelos, ou seja, para criar programas que devem rodar em múltiplos computadores. Este protocolo é um sistema de transferência de mensagens projetado pela indústria e por acadêmicos para funcionar em uma variedade de computadores paralelos.

O padrão define a sintaxe e a semântica de uma série de rotinas de uma biblioteca. Uma grande variedade de usuários que escreve programas de passagem de mensagens usa MPI, sobretudo para programas de supercomputadores. MPI não é um biblioteca, é um padrão que possui duas grandes versões (MPI-1 e MPI-2). Existem muitas implementações do MPI, como a LA-MPI, Microsoft MPI, AMPI. Por ser um padrão relativamente antigo (øMPI-1 foi lançado em 1994), existem implementações depreciadas (não são mais usadas). Por exemplo, a implementação Open MPI foi criado como uma fusão de outras 3 implementações (FT-MPI, LA-MPI e LAM/MPI).

O MPI permite que um processo seja enviado e executado em outros computadores, compondo um grupo de processos. Cada processo no grupo é associado a um Rank, cujos valores vão de 0 a $N-1$, sendo N o número de processos do grupo. O processo deve estar preparado para executar de acordo com o Rank que recebe.

Cada processo é capaz de criar uma comunicação com outro apenas indicando o Rank de quem deseja receber ou enviar mensagens (comunicação ponto a ponto). A forma de enviar/receber mensagens pode ser com ou sem bloqueio (esperando ou não o envio/recebimento de mensagens). No MPI-2 surgiu a possibilidade de uma comunicação em grupo, ou seja, é possível enviar mensagens para todos os processos com uma única função.

O MPI cria uma abstração do hardware a ser usado. Uma implementação de MPI deve ser instalada em todos os computadores da rede e ser devidamente configurada (considera a rede que será usada, como Ethernet ou InfiniBand), o que envolve a instalação de um Run-Time Environment. O executável utiliza a biblioteca de MPI que acessa este Run-Time Environment, e que por sua vez se comunica com os outros computadores.

Parallel Virtual Machine

Apesar de o MPI ser mais usado atualmente, existem outros softwares com o mesmo propósito, como o PVM (Parallel Virtual Machine) [17]. O MPI é mais simples, focando na passagem de mensagens entre os processos. O PVM, apesar de mais antigo possui mais funcionalidades. O controle sobre os processos é muito

maior, permitindo que os processos sejam interrompidos durante a execução. O PVM também permite que recursos sejam compartilhados com facilidade, como um HD.

Distributed Operating System

Um Distributed Operating System [18] é uma generalização de um sistema operacional tradicional. Estes sistemas operacionais provem todos os serviços e funcionalidades de um sistema operacional. Para um usuário, o sistema funciona como um sistema operacional monolítico em um único nó de uma rede, quando na verdade é constituído de múltiplos nós.

Distributed Operating Systems possuem tipicamente um Microkernel, que possui um conjunto mínimo de funções como o gerenciamento de memória e de threads, e a Inter-Process Communication. Em cima do kernel, ainda dentro do Sistema Operacional funciona um Sistema de Gerenciamento de Componentes. Estes componentes são o que permite que um nó de uma rede se comunique com outro nó, que os recursos de um nó estejam disponíveis para os outros e que permite o controle remoto sobre processos

O SO cria uma camada no acesso aos recursos, permitindo que os recursos sejam acessados como se estivesse localmente. Internamente o SO se comunica com o sistema de outro nó e faz a transferência dos dados necessários. Os processos também podem estar sendo executados em um nó diferente do que o usuário está, mas o SO esconde esta informação do usuário. Distributed Operating System está intimamente relacionado à Cloud Computing [19] por permitir que processos e recursos estejam em servidores e ainda sejam acessíveis ao usuário.

QNX

QNX [20] é um sistema operacional em tempo real voltado para o mercado de sistemas embarcados. O QNX é composto pelo QNX RTOS (QNX Real Time Operating System) e pelo QNX Momentics Tool Suite. O QNX Momentics é um software que deve ser instalado em computadores em que os softwares para o QNX serão desenvolvidos. O Momentics pode ser instalado no Windows, no Linux ou no QNX RTOS.

O QNX Momentics é um ambiente de desenvolvimento baseado no Eclipse para criar projetos em C/C++. O Momentics permite que o usuário adicione targets (computadores com QNX RTOS) indicando IP e porta para a comunicação. Os projetos podem ser compilados, enviados para um target e executados neste target. O Debug no Momentics permite o uso de break points para travar o processo em um target e com isto visualizar cada um dos seus threads com informações detalhadas (código em C/C++, valor de variáveis, pilha de funções).

Um aspecto muito importante do QNX é ser um sistema operacional com kernel de tempo real. SOs em tempo real são usados em muitas aplicações em que

seriam usados microcontroladores. Microcontroladores possuem interrupções que permitem a sincronização de eventos de forma precisa, mas estes SOs acrescentam muitos outros recursos. O kernel de sistema operacionais como o Windows utilizam algoritmos que impedem threads de serem atendidos imediatamente após acordarem de um sleep (impedem o tempo real). Desde o kernel 2.6 o Linux permite aplicações soft real time (tempo real não crítico) [21], mas o patch PREEMPT_RT[22] o torna hard real time [23] (tempo real crítico), como é o caso do QNX. Existe ainda outras formas de tornar o Linux hard real time, como o XENOMAI[24] e o RTAI[25]. O QNX é uma alternativa proprietária para o tempo real que conquistou seu espaço em aplicações graças ao suporte da empresa que o criou. No QNX o SO é um conjunto de processos. Desta forma, outros processos podem interromper os processos do SO (caso tenham prioridade maior) sem causar quaisquer danos ao SO. Apesar de ser UNIX, o funcionamento do kernel é diferente do Linux.

URBI

URBI (Universal Real-time Behavior Interface) [26] é um framework multiplataforma (Windows, Linux, Mac OS X) em C++ usado no desenvolvimento de aplicações para robótica ou sistemas complexos. O URBI foi criado pela Gostai.

O URBI possui a urbiscript, uma linguagem de script (linguagem interpretada) projetada para ser usada em cima de uma arquitetura cliente-servidor para controlar remotamente um sistema.

A principal ideia do URBI é transformar um robô em uma série de componentes (objetos da classe UObject em C++) que serão representados no urbiscript, podendo ser acessados e/ou controlados. Os componentes podem estar presentes no mesmo computador em que o urbiscript é executado ou podem ser acessados remotamente através de uma comunicação TCP/IP. O urbiscript trabalha em cima do URBI ao invés de tratar diretamente com os UObject, portanto para o script não importa se o UObject está em outro computador ou não. Os componentes podem ser adicionados ou retirados diretamente. O urbiscript acaba tratando com a lógica do controle dos objetos enquanto os UObject ficam encarregados de rodar algoritmos, controlar atuadores, requisitar dados de sensores, etc. A figura 1.4 mostra a arquitetura do URBI.

O urbiscript foge do padrão de programação tradicional por permitir que comandos sejam executados de forma paralela com facilidade (por exemplo, não é necessário se preocupar com a criação de threads). Eventos também são criados com facilidade, indicando o que deve acontecer quando uma variável passa a ter certo valor ou uma função é chamada com valores específicos. O código abaixo exemplifica o funcionamento do urbiscript.

```
whenever (ball.visible){
    headYaw.val   += camera.xfov * ball.x &
```

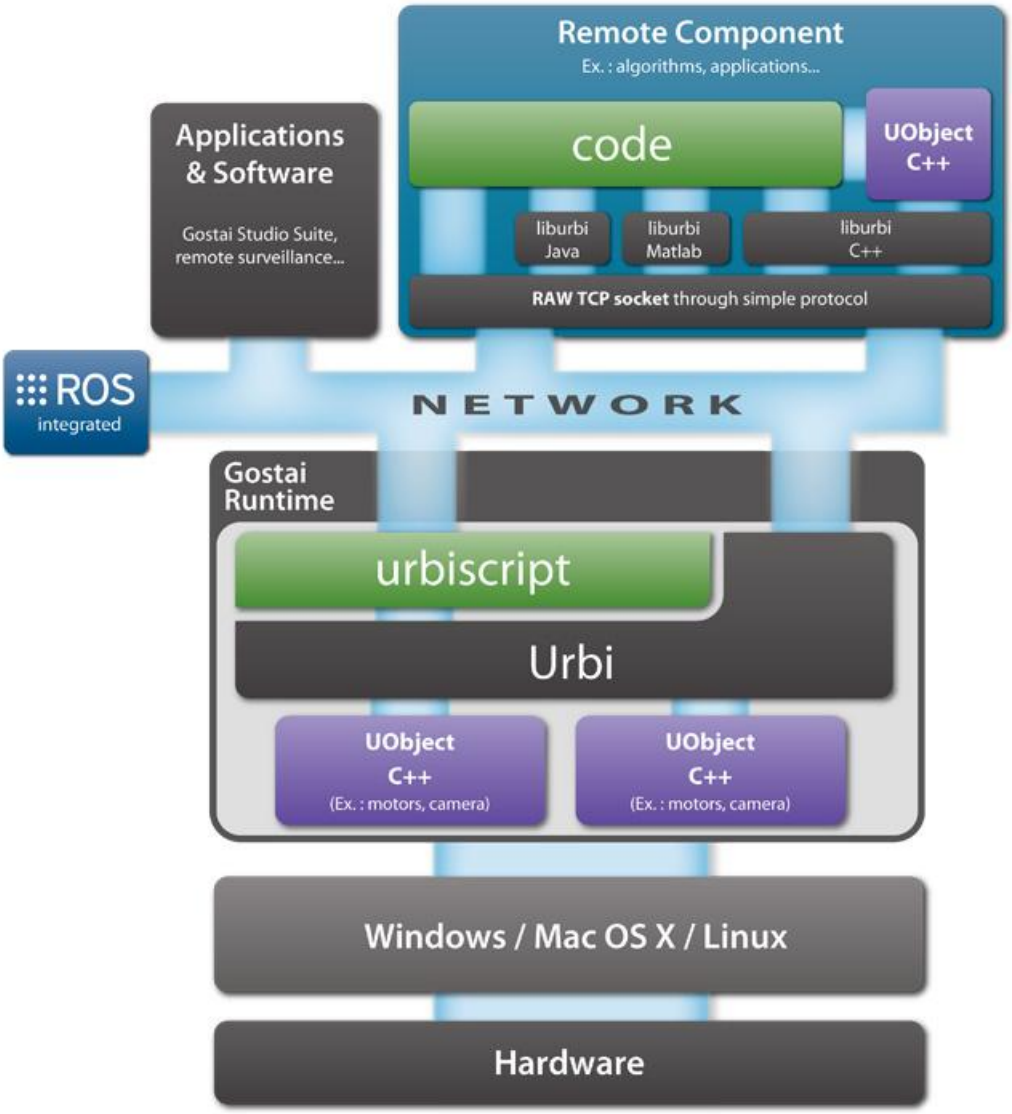

Figura 1.4: Imagem obtida de [2]. O Gostai Runtime executando em um computador é capaz de se comunicar por uma rede com um programa que roda a biblioteca Urbi, representado por um UObject. Estes UObjects podem ser remotamente acessados com um código no urbiscript.

```
    headPitch.val += camera.yfov * ball.y
}
```

Este código foi obtido do exemplo de controle de um robô Aibo [27]. O evento criado permite que a cabeça do robô se movimente para centralizar uma bola na câmera da cabeça do Aibo, quando a bola está visível.

```
at (speech.hear?(``hello''){
   voice.say(``How are you?'') &
   robot.standup();
}
```

Neste outro código, quando o robô detecta a palavra "Hello", o robô responde "How are you?" e se levanta.

Robot Operating System

Robot Operating System, ou ROS [28][29], é um framework para o desenvolvimento de software para robôs. O ROS tem o intuito de facilitar o compartilhamento de código entre diversos autores e com isto diminuir dificuldades no desenvolvimentos de robôs.

O ROS possui implementações para sistemas baseados em Unix, mas é usada em larga escala no Ubuntu, uma distribuição do Linux. O framework consiste em alguns aplicativos e bibliotecas para Python e C++. Os aplicativos são executados por linha de comando.

A estrutura de arquivos no ROS varia com a sua versão (documentação de todas as versões em [30]), mas a abordagem sempre teve um enfoque de reaproveitamento do código com um controle de versão. Um dos conceitos básicos do ROS é a divisão do software usado nos robôs em pedaços (chamados de nodes) e permitir que estes se comuniquem. O código acaba sendo armazenado em um Package. Um Package pode conter ROS nodes, bibliotecas independentes do ROS, arquivos de configuração, etc. Packages podem ser compilados, disponibilizando nodes para serem executados. O ROS possui um grande número de bibliotecas para facilitar o desenvolvimento de softwares relacionados a robôs, mas ao menos uma biblioteca deve ser usada para que o programa compilado represente um node, a API do ROS (roscpp para C++ e rospy para Python). Esta API permite que o aplicativo seja reconhecido como um node no ROS e que faça a comunicação entre os nodes.

Um dos métodos de comunicação entre os nodes é o método Publisher-Subscriber. Nodes podem criar um objeto tipo Publisher para enviar mensagens, ou um Subscriber para receber mensagens. Em ambos os casos deve-se usar uma string para indicar um Topic de comunicação entre os nodes. O ROS cria conexões entre os Publishers

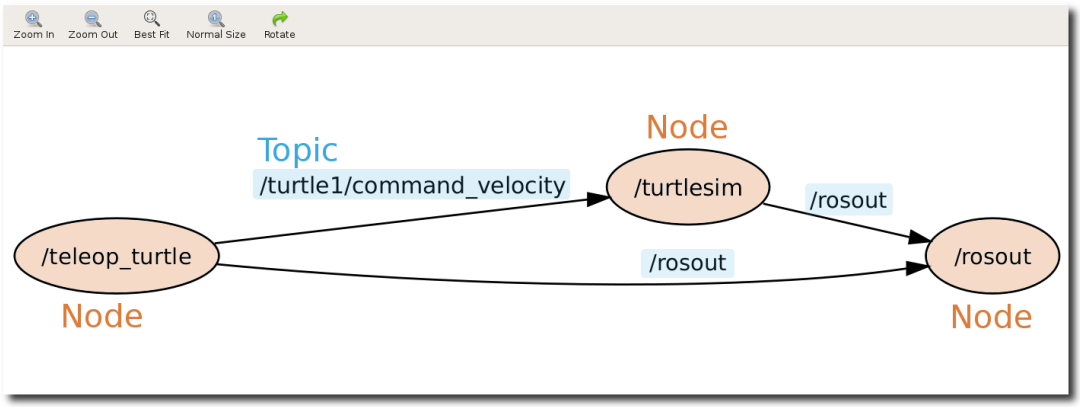

Figura 1.5: O node /teleop_turtle (gera sinais de controle) envia mensagens do Topic /turtle1/command_velocity para o node /turtlesim (simula o movimento de uma tartaruga em uma tela). Desta forma os dois nodes permitem a simulação de um objeto dado os sinais de controle de, por exemplo, um teclado.

e Subscribers relacionados a um mesmo Topic, permitindo que uma mensagem enviada por qualquer Publisher chegue a todos os Subscribers. O programador tem a liberdade para montar as mensagens associadas aos Topics.

A figura 1.5 representa uma comunicação entre dois nodes que é feita por um Topic. O código a seguir mostra a estrutura da mensagem do Topic (a mensagem possui dois floats de 32 bits que indicam as velocidades angular e linear).

```
$ rosmsg show turtlesim/Velocity
float32 linear
float32 angular
```

A comunicação Publisher-Subscriber não é a única no ROS, existe também a Server-Client, que também usa um tópico mas a comunicação ocorre apenas entre um servidor e um cliente. O Client envia um pedido com parâmetros e espera um retorno do servidor. Para facilitar a configuração dos computadores o ROS possui um banco de parâmetros acessível por qualquer node. Outras itens podem ser encontrados/detalhados em [30].

Para iniciar o ROS, deve-se executar o roscore que passa a gerenciar a comunicação entre nodes (cria-se o mestre da comunicação). O ROS pode ser usado em mais de um computador, desde que no outro computador indique-se a localização do roscore (só deve haver um na rede). Isto permite que nodes em qualquer computador se comunique com nodes de qualquer computador.Comandos podem ser invocados em linha de comando durante a execução dos nodes, permitindo, por exemplo, visualizar a relação entre nodes e Topics (como na figura 1.5).

FreeRTOS

O FreeRTOS [31] é um sistema operacional em tempo real muito simples para dispositivos embarcados. O Sistema Operacional possuí um simples kernel escrito

em C, permitindo a fácil portabilidade para outros microcontroladores. Algumas das funcionalidades presentes no FreeRTOS são: presença de um Escalonador [14][32], a criação de tarefas (incluindo corrotinas), controle de recursos (Semáforos e Mutex), funções de timer, entre outras. O FreeRTOS é um Sistema Operacional para microcontroladores muito popular, e é usado em várias aplicações como a robótica.

1.3 Características do Module Operating System

Após descrever algumas das ferramentas utilizadas atualmente, é possível indicar como e onde o Module Operating System se encaixa.

1.3.1 Sistema Operacional versus Framework

O MOS não é necessariamente um sistema operacional. Dependendo de onde o MOS for implementado, ele se comportará como um framework (como o OpenMP [33], URBI e o ROS), funcionando em cima de um Sistema Operacional existente. Neste caso, o MOS é implementado em cima das bibliotecas básicas de outros sistemas operacionais (Windows, Linux, QNX, OS X). As bibliotecas estão relacionadas à manipulação de threads e ao acesso à redes. Os Sistemas Operacionais podem utilizar bibliotecas muito distintas para o mesmo assunto, por isto o documento trata do desenvolvimento do MOS para alguns Sistemas Operacionais.

O MOS faz o papel de um sistema operacional completo quando implementado em microcontroladores. Como dito anteriormente, existem alguns sistemas operacionais para microcontroladores como o FreeRTOS, no entanto, estes não têm as capacidades desejadas para o MOS. Além disto, estes sistemas não possuem uma forma unificada para o acesso às redes. Portanto, é preciso desenvolver um kernel para microcontroladores (permite a criação e sincronização de threads) e criar um acesso padronizado com o acesso às redes (usando, por exemplo a USART).

Este documento trata da implementação do MOS nos Sistemas Operacionais Windows e Linux da arquitetura x86 [34], e no microcontrolador da ATmega2560 [35] da Atmel. Este microcontrolador foi usados como base de estudo por ser usado em um dos populares produtos Arduino[36]. É importante observar que o desenvolvimento do MOS em cada caso possui características únicas. Por exemplo, sendo o kernel do Windows e do Linux diferentes, as opções de prioridade para os threads são diferentes. O MOS é construído de tal forma que o usuário do MOS possa escolher as configurações de seu programa para cada tipo de dispositivo (Linux, Windows ou ATmega2560). Permitir que o usuário tenha acesso às configurações específicas dos dispositivos em que seu programa executará não é tão comum em outros frameworks. A figura 1.6 ilustra o MOS como SO e Framework.

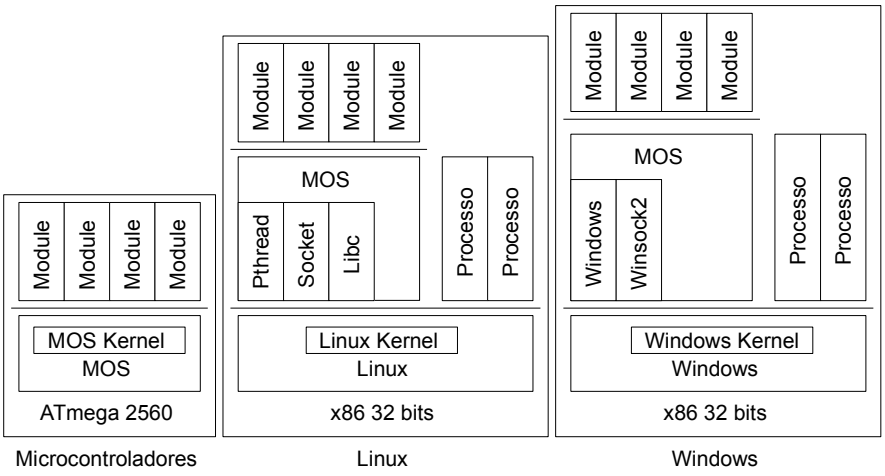

Figura 1.6: MOS como Sistema Operacional e Framework.

Em relação às redes, o MOS é similar aos outros frameworks. As redes devem ser configuradas antes de poderem ser usadas pelo MOS. Por exemplo, a configuração de uma rede TCP/IP pode ser tediosa e complexa, e isto deve ser feito pelo Sistema Operacional nativo, instalando a placa de rede e seu driver. O que realmente importa para o MOS é a biblioteca socket (Windows e Linux) e, para o usuário do MOS, a porta e o IP. Os programas criados com MPI não usam uma comunicação indicando porta/IP, mas a rede ainda deve ser configurada no Run-Time Environment. O ROS, de forma similar, precisa ter configurado a localização do roscore (IP ou nome do computador), e por onde a conexão com ele será criada (porta).

O MOS permite que o usuário configure as redes considerando as redes disponíveis para o dispositivo. Por exemplo, não será possível criar uma rede TCP/IP no microcontrolador ATmega2560, mas será possível criar uma rede RS232 indicando o número da USART em que será usada, o bit rate, uso do bit de paridade, etc.

O MOS também não é um sistema operacional distribuído, já que faz a distinção entre os nós da rede de PUs para seus usuários, enquanto um sistema operacional distribuído equivale a um sistema operacional monolítico para seus usuários (o usuário não sabe onde o programa está rodando).

Não é um dos objetivos deste documento desenvolver o MOS considerando todos os aspectos de um Sistema Operacional. Para Sistemas Operacionais serem bem sucedido foi preciso um grande investimento de corporações ou um grande período de tempo. A vantagem de ser mais visto como um framework é que desta forma o MOS pode ser adaptado para todos os Sistemas Operacionais já bem estabelecidos, apenas complementando sua funcionalidade para as aplicações mencionadas anteriormente. É interessante criar versões do MOS para trabalhar diretamente com processadores de arquitetura x86 e ARM, mas na especificação atual do MOS seriam versões muito limitadas se comparadas aos Sistemas Operacionais que estaria substituindo, tão

limitada quanto é a versão do MOS para microcontroladores.

O MOS pode ser continuamente desenvolvido, aumentando o número de ferramentas disponíveis. Paralelamente, os usuários do MOS podem reproduzir bibliotecas de outras linguagens ou criar bibliotecas e divulgá-las. O MOS também pode ser implementado em outros dispositivos, aumentando o número de Sistemas Operacionais ou microcontroladores em que o MOS funciona.

Os SoCs (System on a Chip) [37] estão sendo usados cada vez mais na indústria. Os SoCs possuem vantagens de custo e de tamanho em relação aos processadores normais por ter vários recursos dentro do próprio chip, dispensando vários itens presentes em uma placa mãe ou periféricos. Analisando por outra ótica, SoCs são microcontroladores mais potentes por possuírem mais tipos de recursos (acesso a redes, câmera, display, sinais analógicos, GPU, etc.), possuírem um clock maior e utilizarem memória RAM (Random Access Memory) externa (permite um uso maior de memória RAM). Os SoCs vêm aumentando em número pois os fabricantes estão sendo capazes de minituarizar e integrar esses componentes dentro do chip do processador. A arquitetura ARM é atualmente a mais usada na produção de SoCs. A ARM padroniza a utilização desses componentes. Apesar do documento não tratar o uso do MOS em SoC, este seria o objetivo final de aplicação do MOS. Enquanto o SoC integra componentes no processador (como um microcontrolador faria), o MOS permitiria os programadores acessarem estes componentes com facilidade.

1.3.2 Module

O Message Passing Interface e o Parallel Virtual Machine são capazes de transferir processos entre os dispositivos de suas redes. O MOS, ao invés de lidar com processos, lida com Modules. Ao invés de um usuário criar um único programa, no MOS ele pode criar diversos módulos. A ideia de dividir um programa em partes menores é parecida com os nodes do ROS ou o UObject no URBI, mas o funcionamento é completamente diferente. O programador tem a liberdade de concentrar todo seu programa em um único Module se assim o desejar.

Modules podem ser trocados por dispositivos contendo MOS, de forma similar ao feito com processos nos outros frameworks. O que diferencia o Module de um processo normal ou de um node no ROS é a forma como os Modules interagem entre si e com o MOS.

Alguns dos mecanismos de IPC (Inter-Process Communication) são a memória compartilhada e a passagem de mensagens. O ROS com o Topic/Publisher/-Subscriber fornece outra forma de comunicação que pode ser usada entre processos executando em qualquer computador com o ROS. O MOS muda um pouco esta abordagem, dividindo a comunicação entre dois tipos básicos, a comunicação

Module-Module, e a comunicação Module-Protocol.

Para falar do Module-Module é necessário apresentar antes os Projects. No MOS, Projects são projetos que o programador cria. Classes e Modules são definidos dentro de um Project através da linguagem MOSLanguage. Nesta linguagem, que é orientada a objetos, um Module é uma classe com algumas peculiaridades extras. Um Module é essencialmente uma classe, portanto possui funções e objetos de outras classes que podem ser classificados como private, public ou protected. Um Module também pode ser derivado de uma classe ou de outro Module. Um Module pode conter um objeto de outro Module, mas uma classe não pode conter um objeto de um Module por causa da interação Module-Module.

Os Modules definidos em um Project possuem um controle de versão. Um número é associado a todo Module sempre que alguma mudança for feita no Module (em seus objetos ou em suas classes base). Além disto, o Project também guarda o número da última versão em que a parte public do Module foi modificada. Um Project pode ter outros Projects como dependência. Existem dois tipos de dependência para os Projects, a Full Dependency (dependência completa) e a Parcial Dependency (dependência parcial). Na completa, um Project tem acesso a todas as classes, Modules e etc., definidos no Project de que depende. No parcial, apenas a parte public dos Modules (e consequentemente as classes dos objetos public dos Modules) são visíveis ao programador. A figura 1.7 ilustra como as versões são vistas na mudança de código de classes e Modules.

Suponha um Module A que possui um objeto do Module B (figura 1.8). Não importa se o B está ou não no Project do A, o que importa é o public do B ser conhecido. O Module A guarda qual a última versão do Module B em que public não foi alterado. O programador utiliza as funções ou objetos do Module B como se fosse uma classe normal. Desta forma, se o Module B for de um Project diferente do Module A, o Project do B estaria incluindo o código fonte de A.

Projects possuem uma lista de instruções escritas pelo programador, que envolve a compilação de Modules (definidos no próprio Project ou em Projects com Full Dependency), transferência destes Modules para outros dispositivos, e a execução ou destruição de Modules (definidos no próprio Project ou de Projects com Full Dependency) em qualquer dispositivo.

O programador possui uma etapa extra ao usar o Module B dentro do Module A. Modules têm acesso a funções extras que não são permitidas em uma classe normal. Algumas destas estão relacionadas à conexão entre dois Modules (Module Connection). O Module A usa uma destas funções para pedir a Module Connection com o Module B. Vale lembrar que o Module A possui o número da versão do Module B que deve se conectar. Esta função faz o MOS verificar se já existe um Module B executando no dispositivo. Caso não esteja executando, irá verificar se existe um

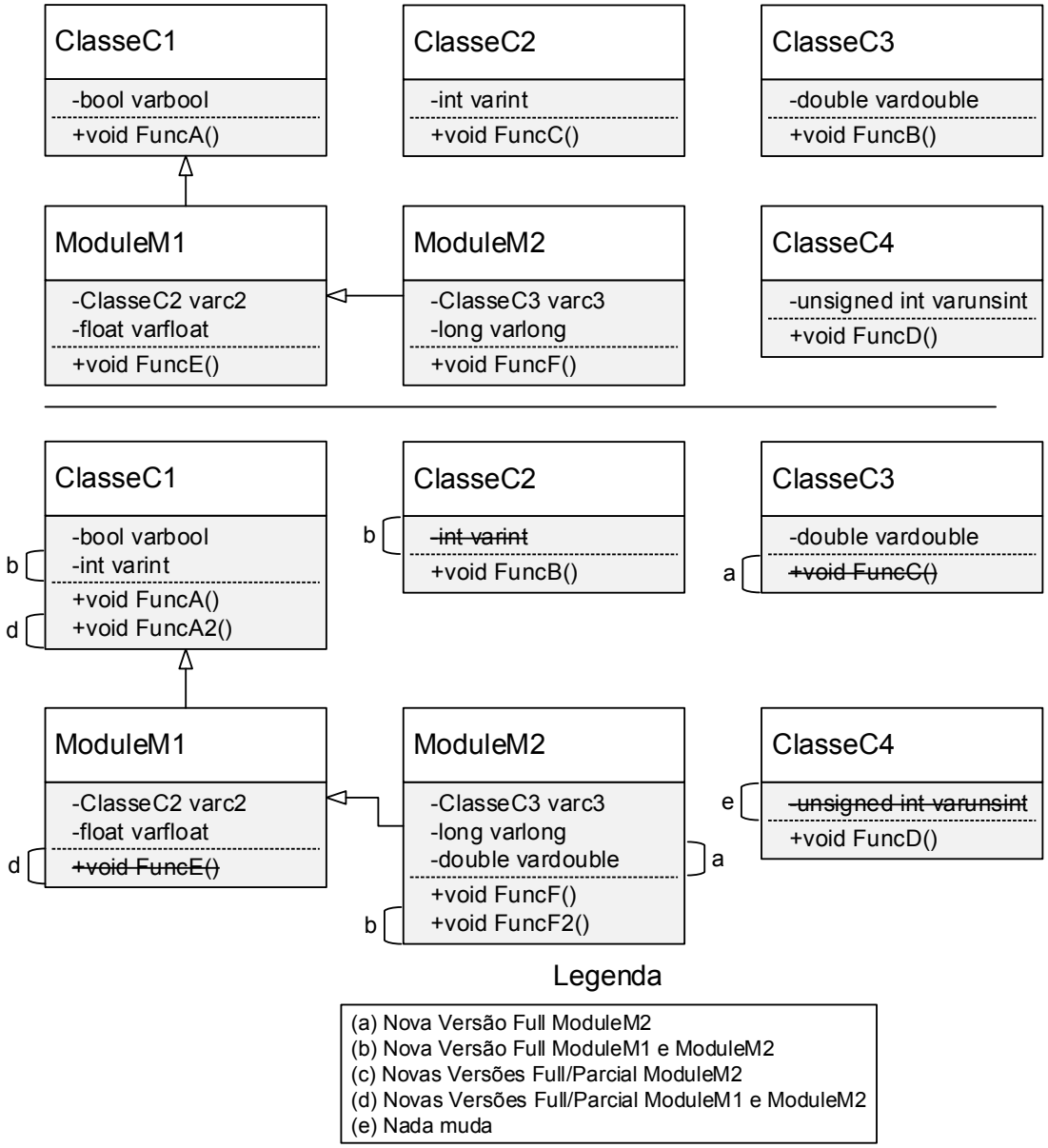

Figura 1.7: Quatro classes e dois Modules são criadas no lado esquerdo, cada um com suas variáveis e funções, sendo ModuleM1 derivado da ClasseC1, e o ModuleM2 derivado do módulo ModuleM1. Para facilitar, todas as funções são públicas e todas as variáveis são privadas. O lado direito mostra mudanças nas classes e Modules, e as cores indicam que tipos de versões as mudanças gerariam.

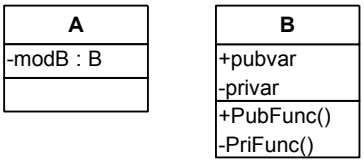

Figura 1.8: Modules A e B usados como exemplo.

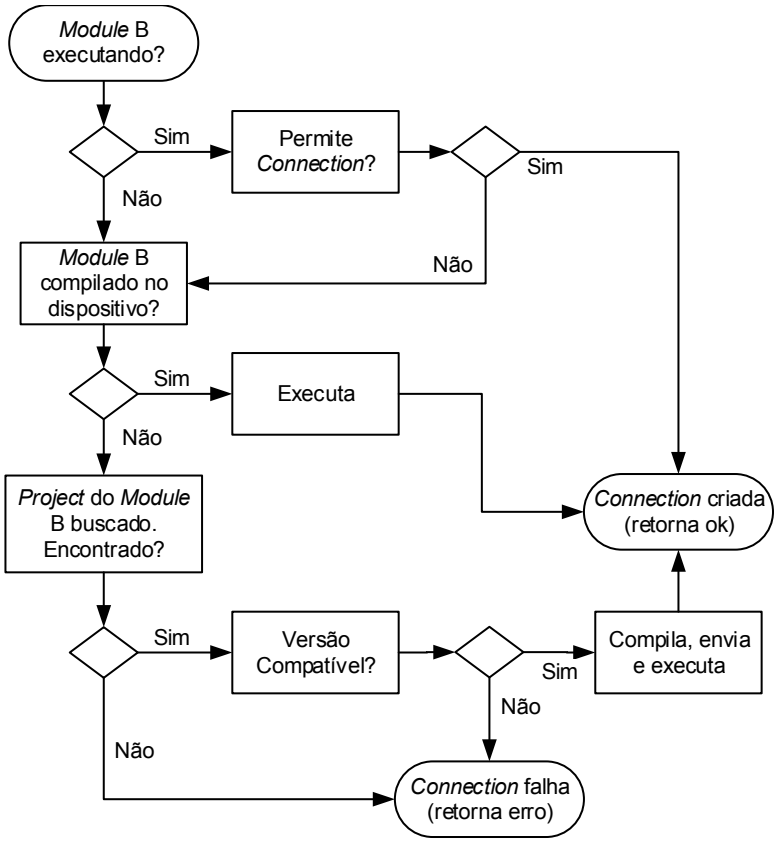

Figura 1.9: Procedimento da Module Connection.

Module B compilado no dispositivo para então iniciar a execução. Se não estiver no dispositivo, o MOS irá procurar o Project do Module B nos outros dispositivos com MOS. Quando encontrar, pedirá para compilar e enviar o Module B para o dispositivo em questão, e depois inicia o Module B. Se o Project não for encontrado ou a versão atual do Module não for compatível, a Module Connection falha. O procedimento é representado na figura 1.9.

Enquanto o Module B não estiver conectado com o Module A, o acesso ao Module B falhará. Isto significa que as funções de B irão retornar "lixo", a leitura de objetos retornará "lixo" e a escrita não ocorrerá. No Project, as funções public de um módulo conectado ganham uma nova variável de retorno que indica se a função falhou ou não, permitindo que o programar saiba da desconexão do Module. A conexão entre 2 Module no MOS parece a de um driver no Linux, e internamente (dentro do MOS), a conexão entre dois Module é similar à de uma biblioteca dinâmica.

Mesmo sem conexão entre Modules, qualquer Module executando pode explicitamente pedir para que outro seja executado em qualquer dispositivo MOS desde que tenha as informações básicas do Module (nome do Module e do Project e a versão. Note que na Module Connection isto não basta, o Module B deve estar executando no mesmo dispositivo e o Project do Module A deve ter acesso ao Module B mesmo

	Module A	Module B
Mensagem	Tratamento	Tratamento
M1	Definido	Definido
M2	Definido	Não Definido

Tabela 1.1: Tratamento das mensagens M1 e M2 nos Modules A e B, sendo B um Module conectado com A (Module Connection)

se for de outro Project.

Em relação ao paradigma de orientação a objetos, o MOS permite que a conexão de Modules ocorra com classes derivadas. Suponha agora o caso em que o Module B possui dois Modules derivados B1 e B2. O Module A continua com um objeto do Module B, mas pode pedir a conexão com B1 ou B2, contanto que a versão do public de B de que ambos são derivados seja uma versão aceitável para o A.

Na comunicação Module-Module existe outra forma de comunicação importante, a passagem de mensagens, uma forma de IPC eficiente e muito difundida. Além dos próprios sistemas operacionais que permitem passagem de mensagens entre processos, as APIs de interfaces gráficas MFC [38], Wx Widgets [39], Qt [40], largamente usadas possuem esquemas próprios de passagem de mensagens dentro do processo, mostrando como funcionam.

Se for o desejo do programador, uma série de mensagens pode ser definida em um Module (nome da mensagem e o que deve ser feito ao recebê-la), sendo que estas mensagens podem ser enviadas para si próprio. O MOS possui um thread exclusivo para tratar as mensagens de todos os Modules. No caso dos Module A e B, suponha que B defina as mensagens M1 e M2 e o tratamento de M1 (tabela 1.1). O Module A pode definir o tratamento das mensagens M1 e M2. Quando M1 é enviado (por A ou por B), A e B tratarão a mensagem, mas quando M2 for enviada, apenas A irá tratar M1. Vale comentar que, diferente do modelo de mensagens usado em outras tecnologias, as mensagens não possuem parâmetros a serem tratados, são apenas nomes/identificadores para invocar um tratamento.

Na primeira forma de comunicação Module-Module, a comunicação ocorre apenas em um sentido (Module A acessa diretamente o B), mas a segunda forma permite uma relação mais ampla entre os dois Modules (o B é capaz de enviar comandos para A, mesmo não conhecendo A).

O outro tipo de comunicação permitido no MOS é a Module-Protocol. Como dito anteriormente, o MOS define o acesso a algumas redes. As redes estarem disponíveis para o programador depende das redes estarem presentes nos dispositivos e estarem propriamente configuradas. Após configuradas, um identificador estará associado ao tipo de rede, permitindo que Modules usem a rede. Algumas redes necessitam de uma etapa extra, a da formação de uma conexão. Por exemplo, na RS232 bastaria

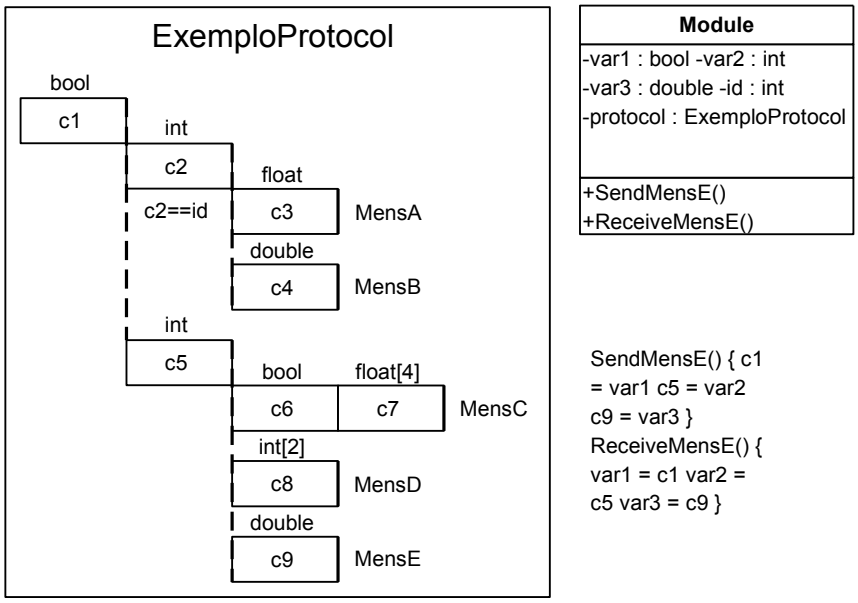

Figura 1.10: Exemplo de aplicação de um Protocol e um Module. O Module Module possui um objeto de Protocol, define como a mensagem MensE será tratada, e como a mensagem MensE é preenchida quando for enviada. A classe também define como as mensagens MensA e MensB são escolhidas, comparando com uma variável interna do Module.

indicar qual porta física deveria ser usada na rede, mas na TCP/IP, a conexão deve estar pronta para os Modules a usarem. Existem funções globais na MOSLanguage (acessadas através do objeto global system) que permitem um Module configurar uma rede, tanto a parte física quanto a conexão.

Projects podem definir Protocols, que definem um protocolo de mensagens. Isto quer dizer que mensagens são definidas com a união de campos, que por sua vez são definidos por uma classe ou um vetor (com o tamanho máximo definido) de uma classe. Quando um Module possuí um objeto de um Protocol, é preciso definir como este Module identifica e age com a chegada das mensagens definidas. Da mesma forma é preciso definir como os campos devem ser preenchidos no envio destas mensagens. O interessante no Protocol é permitir o acesso direto aos campos mencionados tendo ainda o Module como escopo. Por exemplo (figura 1.10), considerando uma mensagem que tenha os campos c1, c3 e c9 e um Module que possui as variáveis var1, var2 e var3, o envio da mensagem poderia ser definido como "c1 = var1; c5 = var2; c9 = var3", e o tratamento da mensagem como "var1 = c1; var2 = c5; var3 = c9".

O Module tem definido quais Protocols ele irá usar nas redes. Um Module pode verificar as redes que estão disponíveis no MOS (configuradas e/ou com conexão estabelecida), e depois escolher uma destas para aplicar o Protocol. O MOS permite que vários Protocols sejam usados em uma mesma rede, permitindo o compartilha-

mento do recurso (a rede) entre todos no MOS, porém seguindo a ordem de entrada dos Protocols na rede em questão (os Protocols são armazenadas em uma lista ordenada). Na implementação do MOS deve-se decidir o que fazer sobre quantos Protocols podem ser usados ao mesmo tempo, o que fazer com uma mensagem que não é propriamente identificada (descartar ou concatenar), se a mensagem deve ser descartada após identificada por um protocolo ou se deve prosseguir para o próximo Protocol, etc.

Cada rede possui um thread para tratar os Protocols com que se conecta, portanto, este thread acessa os Modules para tratar as mensagens após serem positivamente identificadas. O funcionamento do Protocol é melhor explicado na seção 2.11.2.

Esta abordagem no acesso às redes facilita o trabalho do programador pois este não precisa se preocupar em separar os bytes da mensagem manualmente para usá-las. Além disto, quando um Module é derivado de outro, as definições do envio e do recebimento das mensagens em um Protocol se mantêm, mas pode ser modificadas/complementadas com facilidade pelo programador. Quando um Module deve se comunicar com um dispositivo na rede que já tem um protocolo bem definido o programador pode escrever o protocolo no MOS (seguindo o manual do dispositivo) com agilidade.

É importante salientar que o MOS trata redes diferentes de forma diferente (configuração de redes), assim como trata dispositivos diferentes de forma diferente (compilação de Modules). A filosofia do MOS é compreender que podem existir diferenças grandes nas capacidades dos dispositivos MOS e no papel que possuem em uma aplicação e que, portanto, não podem ser tratados igualmente. Com as diferenças claras e fornecendo ferramentas para lidar com as peculiaridades de cada dispositivo, o MOS facilita o trabalho do usuário/programador.

1.3.3 Module Operating System Network

Uma diferença muito grande entre os Sistemas Operacionais convencionais e o MOS é a existência da rede Module Operating System Network, ou MOS Network. Dispositivos com MOS podem ser conectados entre si formando uma rede, como os frameworks ROS, MPI e etc. também fazem.

Um dos objetivos do MOS é ser capaz de conectar dispositivos completamente diferentes dentro da mesma rede (arquiteturas diferentes, redes diferentes e etc.). Para tal, a MOS Network deve funcionar em cima de redes diferentes. Isto é alcançado com um mestre na rede, o Module Operating System Master (MOS Master).

Em uma MOS Network é permitido um único MOS Master, que é responsável por aceitar e fazer a retirada de nós (dispositivos com MOS) da rede. Todo dispositivo

que entra na rede possui um ID único (identificador). O mestre mantém um registro de todos os dispositivos da rede com uma série de informações, o que inclui o nome do dispositivo (string de identificação pessoal), o tipo de dispositivo (ATMega2560, Windows x86, Linux x86, etc) e uma lista dos Projects. O mestre permite que um dispositivo encontre um Project na rede (na busca de um Module) sem verificar todos os dispositivos.

O MOS Master também tem a função de criar conexões entre os dispositivos da rede. No MOSLanguage existe um Protocol especial, o MOS Network Protocol. Ele pode ser usado para adicionar uma rede (RS232, TCP/IP, etc.) à MOS Network. Em contrapartida, um Module pode acessar o sistema para configurar uma conexão na MOS Network. Com a conexão devidamente criada (similar ao TCP/IP, no esquema cliente-servidor) um Protocol qualquer definido em Modules podem ser usados nesta conexão, como seria feito nas outras redes. Para criar uma conexão é necessário indicar o nome ou o ID do dispositivo (se for o cliente da conexão) que se deseja conectar, uma porta e um tempo de espera.

Internamente a MOS Network utiliza um sistema de roteamento de mensagens, que é executado pelo MOS Network Protocol. Toda mensagem contêm no início um número que indica a rota. O outro dispositivo no MOS recebe a mensagem, verifica o número e, seguindo uma tabela, se não for o dispositivo de destino da rota muda o número e repassa a mensagem por certa rede que contêm o MOS Network Protocol. Desta forma o MOS cria túneis por onde a mensagem passa até chegar ao seu destino. O MOS Network Protocol é explicado com mais detalhes na seção 2.11.3.

Quando um MOS Network Protocol é adicionado em uma rede, uma rota é criada para ligar o dispositivo ao MOS Master. Em seguida uma série de mensagens é trocada entre os dois para cadastrar o novo dispositivo e de fato conectá-lo na rede. Se o dispositivo já está na rede o procedimento termina com a criação da nova rota. Esta rota é usado para pedir a localização de um Project, criar rotas para a transferência de Modules entre dispositivos, para criar/destruir uma conexão entre dois dispositivos MOS (outras rotas são criadas), etc.

Com a MOS Network o MOS fornece um meio de comunicação para ser usado entre dois Modules utilizando o mesmo meio que é usado para conectar os dispositivos. Este mecanismo é muito valioso em uma rede que não possui topologia, pois, apesar de dois dispositivos não estarem conectados por uma rede, graças ao MOS eles podem se comunicar como se estivessem conectados diretamente. A figura 1.11 contém um exemplo de rota.

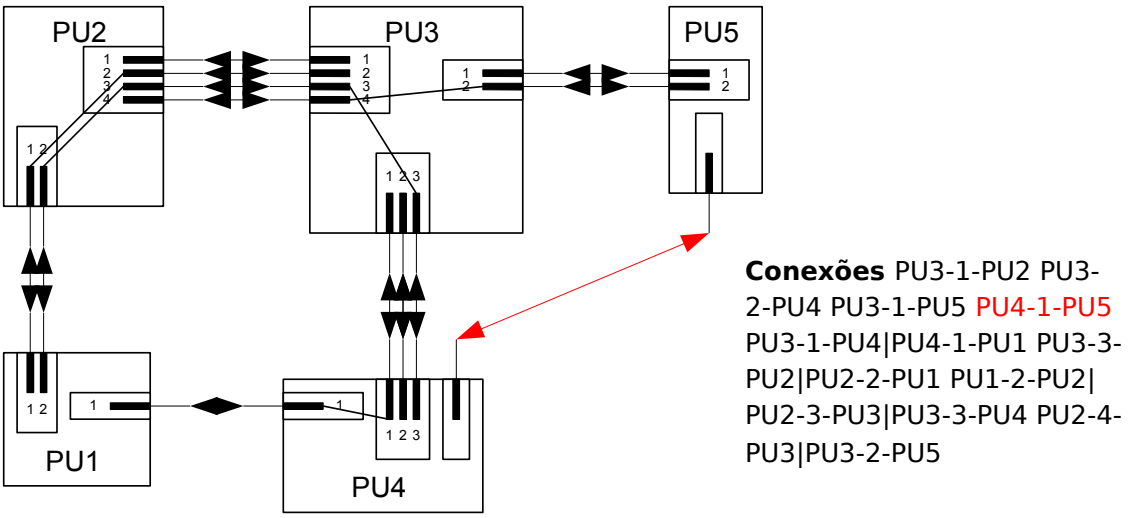

Figura 1.11: Exemplo de rotas em uma arquitetura composta de 5 PUs. A PU3 é o MOS Master. Com a exceção do vermelho, todas as conexões constituem rotas. As redes das PUs possuem apenas o MOS Network Protocol, excetuando a conexão em vermelho, que possui algum outro Protocol. Existe ao menos uma rota em cada redes que liga o dispositivo ao mestre PU3. Não há problema na existência de redundâncias na conexão entre PUs.

1.3.4 Comparações

O MOS é um framework/sistema operacional que tem como objetivo facilitar o desenvolvimento de aplicações em múltiplas PUs. Para tal, o MOS agrega funcionalidades presentes em outros frameworks que têm o mesmo objetivo e adiciona outros.

O MOS permite a passagem de "programas" como o ROS, MPI, PVM, URBI, Distributed Operating Systems, QNX, etc. Também permite que estes "programas" (Modules) se comuniquem como no ROS, MPI e PVM, portanto um Module se parece mais com um node no ROS e um UObject no URBI do que um processo isolado no sistema operacional. O Project e o Module no MOS são similares ao Package e nodes no ROS.

No entanto a forma é diferente. Tudo é feito no mesmo ambiente e na mesma linguagem sendo vantajoso para o desenvolvimento da aplicação (diminui tempo de testes e da integração entre dispositivos). O "programa" pode ser compilado na hora e ser enviado para uma máquina que roda um SO ou microcontrolador diferente. O MOS diferencia a comunicação dentro de uma PU e entre PUs. O MOS permite acesso direto, fácil e padronizado de redes diversas (incluindo a MOS Network) enquanto os outros (ROS, MPI e etc.) permitem acesso apenas às suas próprias redes. Os Modules também se comunicam de uma forma simples e nova dentro de uma PU quando um Module é objeto de outro Module. Também implementa um sistema de mensagem similar ao usado em APIs na relação Module-Module.

Outro aspecto importante no MOS e na MOSLanguage é o uso dos threads. A abordagem escolhida é considerar threads criados pelo programador como parte do MOS e não de um Module porque um thread pode entrar e sair pelos Modules. O MOS também permite um controle sobre a prioridade dos threads de acordo com a configuração do dispositivo que o MOS roda.

Um aspecto importante no desenvolvimento do MOS é de permitir que o programador trabalhe com duas ideias antagônicas em seu projeto. Uma é criar Modules que podem ser executados em quaisquer dispositivos. A MOS Network pode ser usada nesta situação para que um Module seja executado em todos os dispositivos ao mesmo tempo, ou todos que estejam desocupados. A outra é criar Modules que devem ser usados em dispositivos específicos. Neste caso a MOS Network é usada para enviar certos Modules para certos dispositivos e lá usar recursos locais como outras redes, portas analógicas/digitais, configurações de prioridade de thread.

Para ser um framework/SO tão abrangente, o MOS tem de criar algumas concessões. Isto é observado no Capítulo 2, onde o MOS/MOSLanguage são apresentados e suas limitações ficam evidentes. Como MOS não usa as tão difundidas linguagens C/C++/Python/Java, não existem bibliotecas prontas para o MOS (Projects prontos). O principal problema na MOSLanguage é a falta de alocação dinâmica (limitação surge para garantir a compatibilidade do MOS com microcontroladores).

Este trabalho não especifica o MOS em sua versão final, especifica um protótipo do MOS. Não convém focar na IDEs (Integrated Development Environment) para o MOS deve que possibilitem a programação na MOSLanguage, a visualização e o acesso das PUs na MOS Network. Também foge do escopo o uso do disco rígido (x86) e da memória EEPROM [35]| (ATmega2560) no MOS.

Mesmo como protótipo, a especificação é suficiente para exemplificar o uso do MOS nos três exemplos de aplicação.

Exemplo 1. Cliente-Servidor

No cliente-servidor podemos imaginar três casos principais, formados com os clientes pertencendo ou não à MOS Network, e tendo múltiplos servidores ou não.

1. Clientes na rede e um servidor:

   O ProtocolCS é criado (mensagens definidas). O ModuleS e o ModuleC são definidos. Os ModuleS e ModuleC implementam o objeto ProtocolCS protcs. O servidor possui um ModuleS e o cliente possui um ModuleC. O ModuleS tenta criar uma comunicação na MOS Network por uma porta. O ModuleC faz o mesmo indicando o ID da PU do servidor. Ambos usam o objeto protcs nessa conexão. Quando a conexão é desfeita o ModuleS tenta recriar a conexão.

2. Clientes fora da rede e um servidor:

Ocorre o mesmo que no caso 1, mas ao invés da conexão ser pela MOS Network a rede usada é TCP/IP, e se indica porta no ModuleS e IP e porta no ModuleC.

3. Clientes fora da rede e múltiplos servidores:

   Cada um dos servidores terá um ModuleS. Ocorre o mesmo que no caso 2, mas ao invés da conexão ser pela MOS Network entre os servidores e os clientes a rede usada é TCP/IP. Indica-se portas nos servidores e IPs e portas no ModuleC.

Uma observação a ser feita neste exemplo é que um outro Module (ModuleSuperS) pode ser criado nos servidores que contêm objetos do Module ModuleS. Este número de objetos representa o número de conexões que poderão ser feitas ao mesmo tempo entre o servidor e clientes. O ModuleSuperS não possui os objetos da ModuleS, ele deve pedir de um em um. O ModuleS define uma mensagem que indica o estabelecimento de uma conexão (M1) com sucesso, e outra quando a conexão é desfeita (M2) portanto ModuleSuperS pode tratar estas mensagens. A princípio o ModuleSuperS se conecta a ModuleS que iniciar uma conexão TCP/IP. Quando ModuleSuperS receber M1 pedirá ao MOS para conectá-lo a outro objeto de ModuleS. Quando o ModuleSuperS receber M2 verifica se outro ModuleS está buscando uma conexão. Se está, o ModuleS que mandou M2 não é necessário e é desconectado do ModuleSuperS. Se não está, o ModuleS tentará criar uma nova conexão com o cliente.

Exemplo 2. Supercomputadores

Suponha o uso de um supercomputador para realizar uma tarefa. No caso do supercomputador todos as PUs estão na MOS Network. O Protocol ProtTarefa é definido. Os Modules ModFazerTarefa e ModMestreTarefa são definidos. ModFazerTarefa possui um objeto ProtTarefa prottarefa. O ModMestreTarefa possui o número de objetos de ProtTarefa que representam os computadores do supercomputador menos 1.

O programador envia o ModMestreTarefa para um dos dispositivos da rede e para os outros o ModFazerTarefa. Os ModFazerTarefa tentam criar uma conexão com uma porta e o ID do ModMestreTarefa. O ModMestreTarefa executa um thread que em certo momento necessita realizar uma parte da tarefa em múltiplas PUs, então tenta criar um certo número de conexões para realizar a tarefa indicando a porta usando seus objetos do vetor prottarefa. Em seguida envia certa mensagem (definida em ProtTarefa) para cada um dos objetos da prottarefa conectados. Quando todos enviarem de volta uma mensagem com um resultado, o ModMestreTarefa continua sua execução. Eventualmente o ModMestreTarefa destrói conexões, cria novas conexões e envia novos pedidos para as PUs conectadas.

Exemplo 3. Robôs

Consideremos o caso em que o robô é constituído de um controlador (ligado a um motor e um sensor) nas rede RS232, um microcontrolador (ligado a um motor e um sensor) em uma outra rede RS232 e um computador embarcado ligado às rede RS232 e à rede TCP/IP. Um computador pode se conectar ao robô por TCP/IP para controlá-lo.

Os Modules ModMicrocontrolador, ModMotor, ModMotorMicro, ModMotorContro, ModRobo, ModUsuarioRobo são definidos. O ModMicrocontrolador é enviado para o microcontrolador. O ModMotorMicro, ModMotorContro e ModRobo são enviados para o computador embarcado. O computador do usuário possui um ModUsuarioRobo.

O ModMotor define algumas variáveis para representar o motor e algumas funções para iniciar a atualização dos dados ou para enviar os comandos de controle de um motor. ModMotorMicro e ModMotorContro são classes derivadas do ModMotor. O ModRobo possui os objetos ModMotor modmotormicro e ModMotor modmotorcontro. No início da execução o ModRobo inicia os dois objetos com do tipo ModMotorMicro e ModMotorContro, mas ainda usará apenas as funções e variáveis de ModMotor. O Protocol ProtMotorMicro é definido. O ModMicrocontrolador e o ModMotorMicro tem um objeto deste Protocol. O ModMotorMicro é configurado para ser o cliente e o ModMicrocontrolador o servidor por uma porta na MOS Network (configurada em cima de uma das redes RS232).

O Protocol ProtMotorContro é definido e o ModMotorContro tem um objeto deste Protocol. O ModMotorContro usa o objeto da ProtMotorContro na rede RS232 restante. O ModRobo e o ModUsuarioRobo possuem um objeto do Protocol ProtRobo. Comandos são definidos para que o usuário controle o robô, ou seja, para que o ModRobo use as funções e variáveis de seus objetos ModMotor que por sua vez se comunicam pelas redes RS232.

1.4 Organização do Documento

O documento está organizado da seguinte forma:

- Capítulo 1 Introdução: Introduz o MOS.
- Capítulo 2 Funcionamento do MOS: Apresenta os principais elementos do MOS, explica como é a interação entre estes e como atuam
- Capítulo 3 MOSLanguage: Detalha o comportamento da MOSLanguage do ponto de vista do programador da linguagem, mostrando as capacidades e limitações.

- Capítulo 4 Arquitetura De Software: A arquitetura de software do MOS é apresentada para a linguagem C++.
- Capítulo 5 Conclusões: Conclui a dissertação e apresenta trabalhos futuros.

Capítulo 2

Funcionamento do MOS

Este capítulo têm o objetivo de explicar como o MOS funciona. Os assuntos serão tratados como uma sequência de ideias. Não é foco deste capítulo explicar o código por trás do MOS. A arquitetura em si (relacionado ao código em C++) será tratado no Capítulo 4.

2.1 Definições

Para melhor compreensão é crucial a definição dos termos que são usados neste trabalho. Os termos foram criados em inglês em prol da portabilidade do MOS. O plural dos termos segue o inglês, mas o gênero segue a tradução dos termos para o português.

- MOS (Module Operating System): pequeno Sistema Operacional baseado em módulos; desenvolvido em C++
- MOSProcess: um processo de um OS (Windows ou Linux por exemplo) que roda um executável do MOS
- MOSMicro: Microcontrolador que roda um executável do MOS
- MOSDevice: MOSProcess ou MOSMicro
- MOSDeviceType: tipo de dispositivo de um MOSDevice (Win, Win 64, Linux, Linux 64, AVR ATmega2560, etc.)
- MOSLanguage: linguagem orientada a objetos usada nos MOS
- ModClass (Module Class): código de uma classe na MOSLanguage representando um Module
- BModClass (Bounded Module Class): ModClass vinculada a um MOSDeviceType

- CMod (Compiled Module): código objeto e outras informações referentes a uma ModClass para um MOSDeviceType
- RMod (Ready Module): CMod pronto para execução; esta presente no MOSDevice em que será executado (o código objeto é transformado em instruções de máquina da ModClass e estas estão prontas para executar no MOSDevice)
- ModIn (Module Instance): instância de um RMod em um dispositivo (há uma reserva de memória)
- MOSBuilder: compila/liga ModClasses para MOSDeviceTypes, convertendo CMods em RMods
- NativeModule: Module desenvolvidos em C++ ao invés da MOSLanguage
- Module: usado para referenciar o conjunto de ModClass, CMod, RMod, ModIn e NativeModule
- MOSProject: repositório de um projeto contendo definições de classes e Modules na MOSLanguage
- PName: nome de um MOSProject
- MName: nome de um Module
- MVersion: número da versão de um Module
- MPubVersion: número da versão pública de um Module
- MOSLink: vínculo entre um par de Modules (só pode ser estabelecida entre ModIns dentro de um MOSDevice)
- LinkOwner: Module que se conecta a outro através de um MOSLink
- LinkEnd: Module que é conectado a outro através de um MOSLink
- LinkSlot: unidade de conexão nos LinkEnd para formar um MOSLink
- MOSLinkDirect: comunicação direta entre LinkOwner e LinkEnd (sentido: LinkOwner acessa LinkEnd)
- MOSLinkMP (MOSLink Message Passing): comunicação pela passagem de mensagens entre LinkOwner e LinkEnd (sentido: comunicação nos dois sentidos)
- MPThread (Message Passing Thread): Thread do MOS que trata todas as mensagens do MOSLinkMP

- Net (Network): rede de comunicação disponível em um MOSDevice
- NetType: tipo da Net (TCP/IP, RS232, etc.)
- Protocol: protocolo de mensagens definido na MOSLanguage; ModClasses usam em Nets
- BProtocol (Bounded Protocol): protocolo vinculado a um NetType
- Field: campo de dados em um Protocol
- MOSNetwork: Net que conecta MOSDevices (ModIns de MOSDevices diferentes podem se conectar); é aplicado sobre outros tipos de Net
- MOSNetProtocol: Protocol em C++ que representa a MOSNetwork; usado em outras Nets
- Route: rota de mensagens na MOSNetwork
- MOSMaster: MOSDevice mestre dentro de uma MOSNetwork
- System: objeto global na MOSLanguage que permite classes ou ModClasses acessarem o MOS
- MOSAPI: API (Application Programming Interface) do MOS que permite processos usarem redes definidas pelo MOS; estas redes criam uma abstração dos mecanismos de comunicação entre processos de SOs

2.2 Module

Até este momento, o termo Module foi usado de uma forma um tanto genérica. Um exemplo será apresentado para que a diferença em seus componentes fique clara. Considere a existência do Module **M** do MOSProject **P**, e dois MOSDevices **A** (Windows) e **B** (ATmega2560). **B** não possui um RMod de **M** mas deseja executá-lo. O projeto **P** reside em **A**. **B** faz o pedido de execução de **M** do projeto **P**. **A** verifica se já existe um CMod de **M** para ATmega2560, mas como não existe a ModClass de **M** é compilada em um CMod para ATmega2560. O CMod é transferido para **B**. **B** realiza um procedimento no CMod gerando um RMod, que pode ser instanciado como uma ModIn.

Na MOSLanguage um Module equivale a uma classe. Uma ModIn possui uma área de memória reservada para si que corresponde a um objeto de uma classe, ou seja, a área reservada equivale às variáveis definidas na ModClass.

Na MOSLanguage é possível identificar variáveis e funções de Modules como Public (públicas). Modules podem se conectar (MOSLink), e quando o fazem, são as

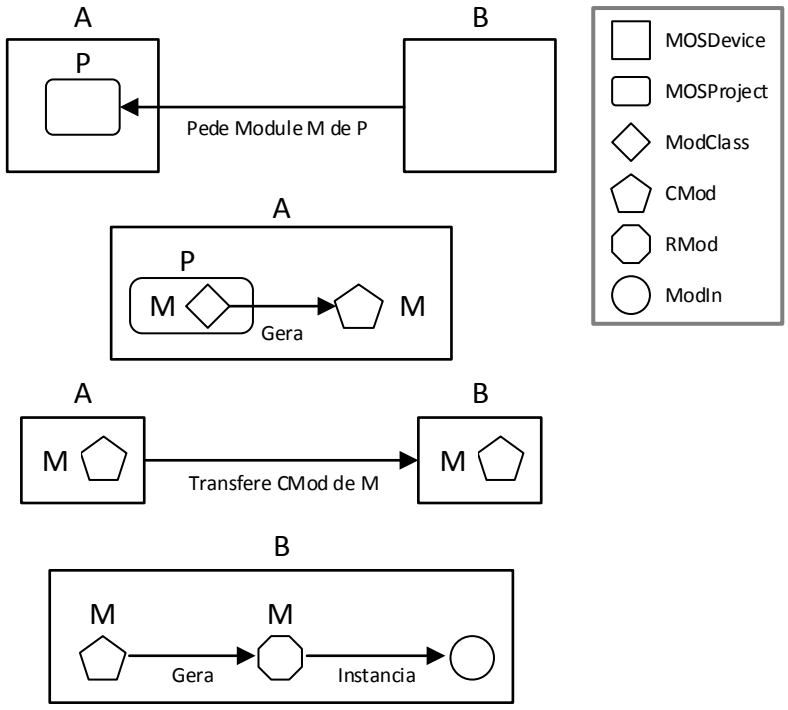

Figura 2.1: Exemplo da transferência de Modules entre MOSDevices.

variáveis públicas que um acessa do outro. A MOSLink será analisada posteriormente.

Threads executam funções dos Modules. As instruções de máquina das funções estão localizadas no RMod de um Module. Ou seja, quando um RMod é "instalado" em um MOSDevice uma área de memória também é reservada. Com isto, threads acessam a área reservada de uma ModIn para ler/escrever nas variáveis do Module, e acessam a memória reservada do RMod para executar as instruções de máquina.

Todo Module possui um conjunto fixo de informações: PName, MName, MVersion e MPublicVersion. A ModClass de um Module permanece em um MOSProject, e o PName guarda o nome deste. O MName identifica o Module. Toda vez que a ModClass for modificada e um CMod for gerado, o Module terá um valor acrescido de 1 em sua MVersion. A MPubVersion é similar à Version, mas o valor é acrescido apenas quando há uma alteração nas variáveis e funções públicas. Note que a alteração no código de uma função publica não muda a MPubVersion, mas a adição de uma função pública, ou a renomeação de variáveis e funções públicas alteram a MPubVersion.

Para demonstrar a utilidade destas informações, considere o seguinte exemplo: existem as ModIn X e Y dos Modules MX e MY em um MOSDevice. X deseja se conectar com um MY, então fornece ao MOS as 4 informações. O MOS encontra Y como uma instância compatível de MY. Se uma das informações fosse diferente, o MOS não encontraria Y como compatível.

Considere agora a existência dos ModIn *Y1* e *Y2* de *MY*, mas com MVersion diferentes, sendo *Y1* a versão mais antiga. Ao invés de *X* fornecer 4, fornece apenas 3 parâmetros, omitindo a MVersion. Ambas versões de *MY* são compatíveis com *X* pois a MPubVersion de *MY* que *X* precisa é a que *Y1* e *Y2* possuem (ambos possuem as mesmas variáveis e funcões públicas). O MOS criará o MOSLink de *X* com *Y2* por *Y2* ser uma versão mais nova do Module.

O controle de versão no MOS é um sistema simples que visa ajudar o desenvolvimento das aplicações. Por exemplo algumas versões podem ser salvas como versões estáveis de um Module. A MPubVersion de Modules é um diferencial por marcar quais versões são compatíveis para o MOSLink ser criado. Os marcadores de versão tornam-se ferramentas para a manutenção fácil de um MOSDevice, permitindo a troca de ModIns por ModIns mais recentes.

2.3 Memória Desprotegida

A proteção de memória é uma forma de controlar os direitos de acesso à memória de um computador e faz parte dos Sistemas Operacionais. Seu principal objetivo é impedir que um processo acesse uma memória que não foi alocada para si, impedindo que um bug em um processo afete outros processos ou o próprio sistema operacional.

Tradicionalmente, a MMU (Memory Management Unit) [14] e o OS gerenciam a memória do computador, separando pedaços de memória para processos e impede que um processo acesse a memória do outro. Soluções atuais para a computação com múltiplos computadores funcionam com a transferência de programas entre os computadores e a execução destes, iniciando processos remotamente. Para que os processos interajam entre si, ou a API da solução é usada, ou os programas estão preparados para usar uma rede (como a TCP), ou estão preparados para usar ferramentas do sistema operacional (como a memória compartilhada). Estas APIs acabam simplificando a comunicação na perspectiva do programador, mas são desenvolvidas em cima das redes/ ferramentas do sistema operacional.

Os processadores possuem uma MMU que os sistemas operacionais utilizam. No entanto, em geral, microcontroladores não possuem esta unidade e possuem uma memória pequena, tornando a proteção sobre a memória inviável de ser reproduzida em microcontroladores. Desta forma, com um OS simples em um microcontrolador, um processo executando poderia acessar indevidamente a memória de outro processo.

O uso da memória desprotegida dentro do MOS é um dos itens que distingue o MOS de outras soluções para computação com múltiplos processadores. É graças a este item que o conceito de Modules surge. A princípio a ModIn teria a mesma funcionalidade de um processo dentro de um OS, mas a ModIn utiliza-se da memó-

ria desprotegida para acessar o espaço de memória de outra ModIn. As interações comentadas anteriormente entre os Modules dentro de um MOSDevice (acesso mestre/escravo e a passagem de mensagens entre Modules de um MOSLink) surge da memória desprotegida. Estas interações seriam complexas de ser reproduzir caso Modules fossem processos (memória isolada) como são os nodes de ROS. Artifícios como a memória compartilhada ou IPC (Inter-Process Communication) teriam de ser extensivamente usados.

A ausência da MMU tem seu lado negativo no MOS:

- A linguagem MOSLanguage não permite o uso de ponteiros para que o programador não tenha acesso integral à memória. Nos OSs, quando um processo tenta acessar uma memória de outro, o processo é destruído, mas isto não pode ser reproduzido pelas ModIn em um MOSMicro pela ausência da MMU.

- Nos microcontroladores (MMU ausente) é preciso dividir a memória em seções fixas para armazenamento de informações no MOS e para as Pilhas De Chamada (ou Call Stacks) [41] de threads (assunto melhor detalhado no capítulo 4.3. A alocação dinâmica padrão do compilador C++ de microcontroladores poderiam reservar espaços que o MOS pretende usar como Pilha de threads. Por este motivo opta-se pela implementação do MOS para microcontroladores (em C++) sem o uso de alocação dinâmica. A linguagem MOSLanguage também não permite alocação de variáveis de forma dinâmica.

- Sem a MMU (MOSMicro), é possível que chamadas aninhadas de funções usem mais memória do que o tamanho da Pilha de Chamada do thread, acessando a Pilha de outro thread e ocasionando em um bug na execução (apenas no MOSMicro). Para resolver este problema seria necessário inserir um overhead de verificação de memória em toda chamada de função e cancelar a execução do thread se a Pilha chegar no limite (computacionalmente caro). Isto deve acontecer tanto nas instruções de máquina do MOS quanto nas instruções das ModIns. O compilador que gera o executável do MOS e o MOSBuilder teriam de adicionar instruções de máquina em toda chamada de função.

2.4 Confiança e Segurança

Este fator é determinante no tipo de aplicação em que o MOS se encaixa.

O MOS deve ser implementado fornecendo segurança na execução, ou seja, o MOS não travará quando um de seus Modules apresentarem um problema de execução. Se uma ModIn não se comporta como planejado (comum no desenvolvimento de uma aplicação), a ModIn pode ser destruída por outra ModIn, ou remotamente

(MOSDevices conectados). O único bug que não pode ser contornado é o estouro da Pilha de um thread nos MOSMicros, como comentado anteriormente.

A MOSLanguage não permite o uso de ponteiros, logo, o programador não tem acesso direto à memória. Isto evita que o programador crie Modules que invadam o espaço de memória de um outro Module pela MOSLanguage.

A utilidade do MOS está em conectar MOSDevices, transferindo e executando Modules. Os MOSProjects podem armazenar os CMod no disco rígido por diversos motivos. Isto pode gerar um problema. É possível criar um CMod em C++ ou assembly que quando compilado tenha as mesmas características do CMod original, mas que comporte-se diferente quando executado. O CMod pode ser transformado em RMod e executado nos MOSDevices da MOSNetwork. Como não há proteção de memória, a ModIn deste RMod tem acesso irrestrito a todo o dispositivo MOS. A ModIn pode inclusive mandar o modulo mal intencionado ser executado nos outros dispositivos da MOSNetwork. Isto permite uma invasão de todos os dispositivos da rede.

Ainda existe uma outra forma para uma invasão no MOS. Dado um ponto de conexão na MOSNetwork livre e, dado que o protocolo da MOSNetwork (MOSNet-Protocol) é conhecido, é possível criar um MOSDevice falso, capaz de se passar por um real, e com isto ter acesso à todos os MOSDevices.

Nesta ótica, hackers veriam o MOS como um prato cheio. Apesar da habilidade de transferir "programas" com facilidade melhore o desenvolvimento e a flexibilidade de aplicações, elas passam a ser inseguras. É por este motivo que o MOS deve ser usado apenas em aplicações bem definidos, em que o intruso no sistema é descartado. Apesar do MOS não prever um sistema de usuários, a adição deste no MOS poderia solucionar esta falha permitindo que apenas MOSDevices conhecidos (usuário e senha) entrem na MOSNetwork ou consigam executar Modules remotamente.

No Windows ou no Linux um dispositivo MOS é um processo. Uma outra saída para aumentar a segurança seria implementar as ModIn como processos isolados no Windows e Linux, tornando o MOSDevice como um conjunto de processos. Desta forma, as ModIn espiãs não teriam acesso integral às outras ModIn. Vale lembrar que uma mudança destas alteraria completamente o funcionamento interno do MOS para estes OSs, além de tornar a comunicação entre Modules mais lenta (as comunicações entre processos é mais demorada do que as dentro de um processo).

2.5 Camada de isolamento

A programação da ModClass na MOSLanguage consiste em definir variáveis e métodos, como se definisse uma classe. Normalmente, métodos de uma classe podem: chamar seus próprios métodos, chamar funções externas ou acessar objetos externos.

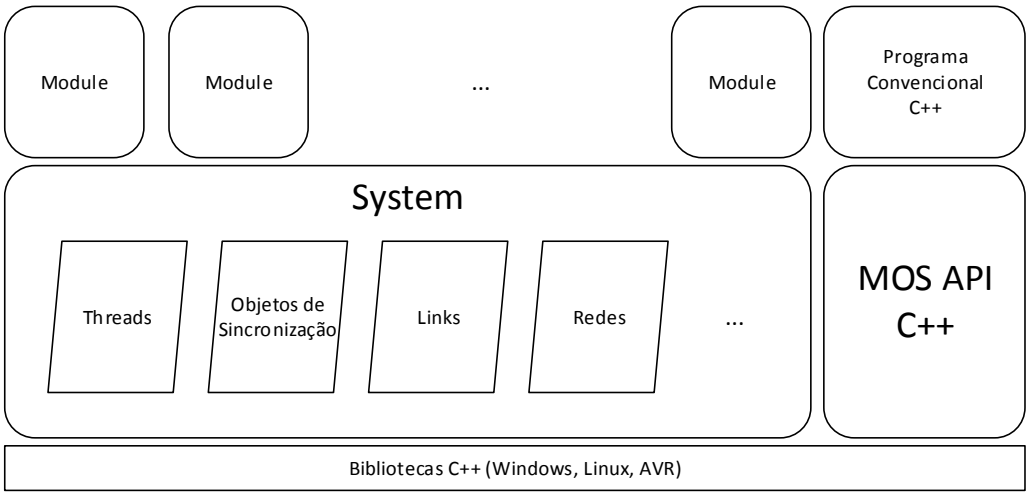

Figura 2.2: Camadas de funções no MOS (MOSLanguage).

Na MOSLanguage, não existem funções ou objetos externos declarados pelo programador. O único objeto externo que o programador pode acessar de uma ModClass é o System.

O objeto global System é o próprio MOS. Modules acessam este objeto para tudo que desejam realizar no MOS. O MOS pode ser representado como uma casca que os Modules acessam. Isto é equivalente ao papel do Sistema Operacional em relação aos processos. Por isto o MOS é chamado de Module Operating System mesmo sendo um Framework em cima de outros Sistemas Operacionais. Esta casca é mais parecida com o Windows do que com o Linux em relação às APIs do sistema pois trata-se de uma casca fechada e fixa. No Linux as mudanças nas APIs estão associadas às versões do Linux que são constantemente geradas.

A casca disponibiliza uma série de funções divididas em conjuntos, de acordo com suas funcionalidades. Por exemplo, um conjunto de funções esta relacionada à manipulação de threads. Vale ressaltar que classes normais definidas em um MOSProject tem acesso parcial à estas funções, enquanto os ModClass tem acesso integral (classes normais e as ModClass são escritas na MOSLanguage). A Figura 2.2 representa o funcionamento da casca. Os conjuntos de funções são estudados no Capítulo 4.

Muitas das funções do System chamam funções do OS nativo. Por exemplo, a criação de threads é feita no Windows com a função *CreateThread()* da Kernel32.dll da Win32 API (biblioteca nativa), ou no Linux com a *pthread_create()* da Pthread (biblioteca nativa). A implementação do MOS na realidade é feita com a biblioteca de thread do Boost [42], que por sua vez chama internamente as funções nativas. No caso da MOSMicro a função é definida no Kernel do MOS de microcontroladores seção 4.3.2). Isto é um exemplo de porque o MOS isola os Modules do resto do

dispositivo.

Um dos conjuntos de funções do System é o conjunto de Objetos de Sincronização (melhor apresentados na seção 2.9). Estes são elementos que ajudam o programador a lidar com múltiplos threads. Tradicionalmente, os objetos estão intimamente ligados ao Kernel (faz parte dos Sistemas Operacionais), pois é este que decide qual thread será executado.

Com isto, o MOS (o System) é uma camada de isolamento entre os Modules e outros Sistemas Operacionais (MOSProcess) ou os Kernels (MOSMicro). A abordagem de camada de isolamento foi adotada no MOS para que os Modules possam funcionar sem se preocupar em qual MOSDeviceType a ModIn será criada.

2.6 Multiplataforma

O aspecto multiplataforma do MOS possui duas vertentes. Uma consiste na arquitetura de software do MOS em C++, e a outra está ligado em como o MOS lida com múltiplos tipos de dispositivos.

A primeira vertente será brevemente mencionada. A arquitetura adotada consiste em uma classe principal, que representa o MOSDevice. É esta classe que possui as funções do System da MOSLanguage, ou seja, é esta classe que cria a camada de isolamento. A classe é derivada para cada MOSDeviceType para com isto adaptar o MOS ao tipo de dispositivo. O capítulo 4 explica melhor a arquitetura de software usada no MOS.

Analisemos agora a segunda vertente. Um dos objetivos dos MOS é permitir que o programador tenha acesso fácil aos recursos de um MOSDevice, mesmo com uma grande gama de MOSDeviceTypes. Toda ModClass pode ser vinculada a um único MOSDeviceType. Quando vinculada, esta ModClass é então chamada de BModClass (Bounded Module Class). A classe passa a ter acesso a um conjunto maior de funções relacionados ao tipo de dispositivo no objeto global System.

Considere o microcontrolador ATmega2560 da Atmel Corporation. Existe uma série de recursos a serem usados neste modelo como portas para I/O analógico e digital, timers, SPI, USART, I2C, etc, como pode ser visto no manual [35]. Alguns destes recursos não podem ser usados ao mesmo tempo. Todos as pinos das portas A até L do microcontrolador podem ser usados como Input/Output mas também possuem outras funcionalidades. A USART2 usa os pinos PH0, PH1 e PH2 (Porta H). Uma série de registradores é usado para configurar o uso destes recursos. A MOSLanguage cria uma camada de abstração em cima destes registradores com uma série de funções no System. Por exemplo, a função System.PortH.Pin1.SetInput() da MOSLanguage faria o PH1 ser um Input, enquanto System.CreateRS232(2) faria PH1 ser o TX (pino de transmissão) da USART2. As funções do System verificam

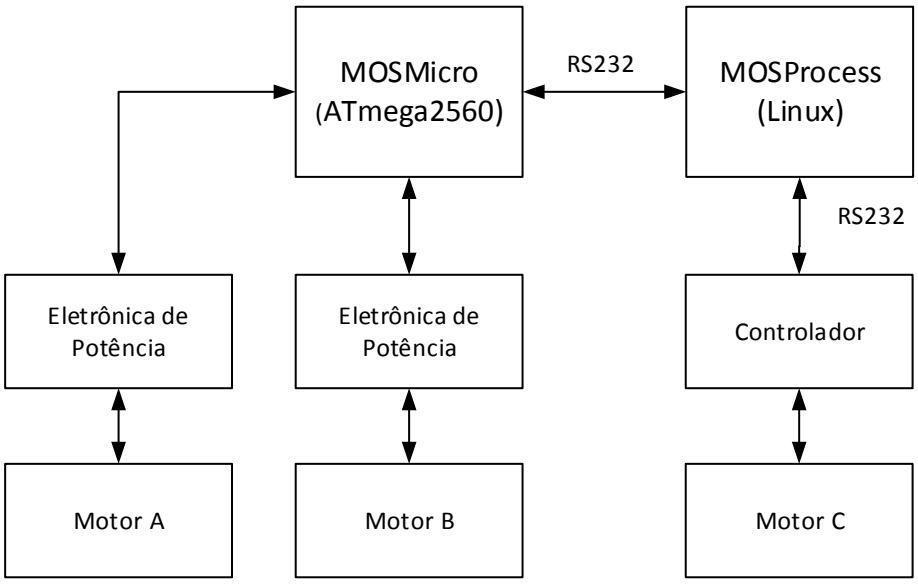

Figura 2.3: Exemplo de um MOSProcess e MOSMicros ligados a motores.

se o pino já está sendo usado em uma opções, e impede que o pino seja usado com os dois propósitos ao mesmo tempo.

Note que as duas funções mencionadas estão disponíveis apenas quando o vínculo de uma BModClass for com o ATmega2560. O mesmo aconteceria com outros MOSDeviceTypes. Por exemplo, o Windows/Linux teriam acesso à função System.OpenTCP() que os microcontroladores não teriam.

Uma ModClass derivada de uma BModClass é também uma BModClass, e é vinculada ao mesmo MOSDeviceType. Uma BModClass não pode ser derivada de outras BModClasses com MOSDeviceTypes diferentes. Isto representa uma condição que normalmente não seria imposta à derivação de classes, mas que é adotada na MOSLanguage.

Um exemplo, representado pela figura 2.3, mostra como Modules podem ser criados com a ideia de derivação. A aplicação consiste de múltiplos motores. Um motor possui um controlador que é ligado a um MOSProcess com uma rede RS232. Dois motores serão controlados diretamente a um MOSMicro utilizando suas portas analógicas e circuitos elétricos.

A figura 2.4 corresponde a uma implementação de ModClasses no exemplo acima atendo-se apenas ao vínculo de ModClasses a MOSDevices. A ModClass MotorControl é responsável por enviar dados de controle para motores. A função Init() deve inicializar mecanismos comuns relacionados ao hardware. A origem dos valores de controle não interessa neste exemplo. A função SendToMotor() deve enviar os valores de controle para os motores, mas nesta classe ela faz nada. A ModClass é então deri-

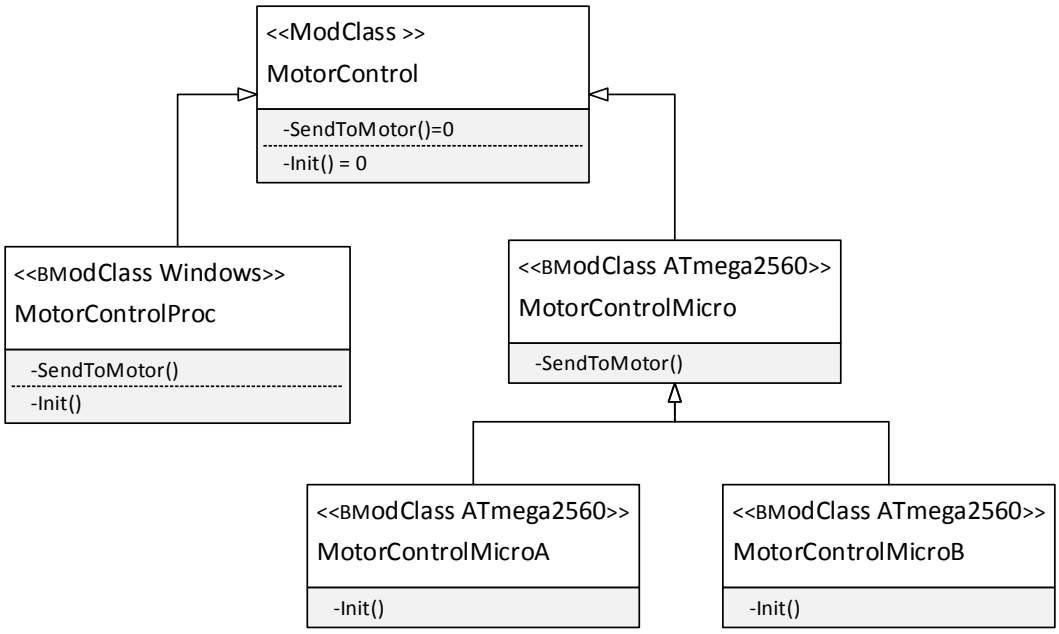

Figura 2.4: BModClasses do exemplo.

vada para as classes MotorControlProc, MotorControlMicro, MotorControlMicroA e MotorControlMicroB. MotorControlProc e MotorControlMicro são BModClasses vinculadas respectivamente ao Windows e ao ATmega2560, e nelas implementa-se tudo relacionado ao MOSDeviceType. No MotorControlProc, a Init() configura a rede RS232 a ser usada, e a SendToMotor() envia os dados para o controlador por essa rede. O MotorControlMicro especifica algo específico para ATmega2560, mas não indica quais portas serão usadas. Isto é identificado nas classes MotorControlMicroA e MotorControlMicroB, com a Init() indicando quais portas serão usadas e a SendToMotor() mudando o valor nestas portas.

O modelo de vínculo de ModClasses com MOSDeviceTypes torna o MOS versátil. Modules são multiplataforma por serem compiláveis para qualquer dispositivo MOS, mas também podem ser vinculados a um tipo de dispositivo para usar seus recursos únicos. O que é feito na MOSLanguage poderia ser feito com outras linguagens. Em C++, *defines*, *ifdef* e *ifndef* podem ser usados nos trechos de códigos que são exclusivos de certos dispositivos (ex: o *include* de uma biblioteca ou a definição de uma classe). A MOSLanguage tenta facilitar o trabalho do desenvolvedor neste quesito, ao vincular uma ModClass a um MOSDeviceType e acrescentando uma série de funções ao objeto System.

O programador da MOSLanguange não deve se preocupar se o MOSDevice é um MOSMicro ou MOSProcess. Esta distinção é utilizada na dissertação para indicar que existe diferenças na implementação do MOS em microcontroladores e Sistemas Operacionais. Para o programador na MOSLanguage o que importa é o MOSDeviceType e o que ele permite ser feito quando uma ModClass é vinculada.

2.6.1 MOSBuilder

O vínculo feito na MOSLanguage tem uma relação direta com a compilação da MOSLanguage feita pelo MOSBuilder. Faremos um paralelo do MOSBuilder com Cross Compiler e Just In Time Compiler[43]. A definição de um Cross Compiler é simples. Quando um compilador em um máquina é capaz de compilar um código para uma máquina de outra arquitetura de microprocessadores ou microcontroladores, este compilador é chamado de Cross Compiler.

Um JIT (Just In Time) Compiler compila um código fonte inicialmente para Byte Code (código em bytes) ao invés de Machine Code (código de máquina). Um programa executável é composto por Machine Code, um código que uma CPU consegue entender (instruções de quais operações devem ocorrer na CPU). Cada arquitetura de microprocessadores e microcontroladores define um certo conjunto de instruções, portanto Machine Code é específico para famílias de microprocessadores e microcontroladores. Byte Code é genérico e pode ser ou interpretado em uma máquina ou transformado em Machine Code para depois ser executado. JIT Compilers compilam o código fonte para byte code, transferem o código para um destino, e no destino transforma o Byte Code em Machine Code. Uma parte do JIT Compiler está na origem do código fonte, e outra parte está no destino. Existem várias vantagens/desvantagens sobre este compilador, mas o que interessa no contexto é que ele diminui o conteúdo que é transferido entre origem e destino, torna o código genérico, mas tem um custo extra de um processamento no destino (interpretar Byte Code ou compilar e executar Byte Code).

O MOSBuilder tem características de um JIT Compiler mas não é um. Pela forma como o MOS funciona faria sentido o MOSBuilder ser um JIT Compiler. No entanto, o MOS deve funcionar também com microcontroladores cujo poder de processamento é muito limitado, logo, fazê-lo interpretar Byte Code ou compilar Byte Code para código de máquina não é uma opção. O MOSBuilder é um Cross Compiler, ou seja, a ModClass pode ser compilada para um tipo de dispositivo diferente do dispositivo que compila. Para uma ModClass ser compilada deve-se indicar o MOSDeviceType. Se a ModClass for uma BModClass o MOSDeviceType já é evidente.

A característica que o MOSBuilder carrega do JIT Compiler é a transferência de um conjunto de bytes para o destino e a alteração destes bytes para gerar o executável. O MOS divide a preparação de um Module para execução em três etapas. A primeira transforma uma ModClass em um CMod, na linguagem de máquina de um MOSDeviceType. O CMod não é executável. A segunda etapa consiste na transferência do CMod da origem para o destino (caso os dois sejam MOSDevices diferentes) usando a MOSNetwork. A terceira consiste em tornar o

CMod executável no MOSDevice, gerando o RMod. As figura 2.5, 2.6 e 2.7 ilustram as 3 etapas etapas de operação do MOSBuilder.

Código objeto é um código de máquina não executável e que para tal deve ser ligado com outros códigos objeto por um Ligador (Linker). O MOSBuilder gera códigos objeto a partir de uma ModClass e de suas classes dependentes, depois o MOSBuilder liga os códigos objeto, mas nem tudo pode ser ligado corretamente. É comum compiladores usarem uma posição fixa para o endereço de funções, mas como as instruções de máquina podem ser carregadas em qualquer ponto da memória, inclusive de outro MOSDevice, o fim do ligamento deve ser feito no destino da transferência. Por isto o CMod não é executável. O CMod é um conjunto de bytes divididos em quatro setores, o primeiro contém o código de máquina (os códigos objetos ligados) e os outros três são tabelas.

Existe uma série de informações que o MOS precisa conhecer para criar uma instância de um RMod. Esta informação não está acessível no primeiro setor, e portanto ficam na primeira tabela (segundo setor). Exemplo de informações são a quantidade/tipo de objetos de sincronização, threads e etc., que o Module precisa. O MOS pode ter um número limitado dos objetos e threads (depende do MOSDeviceType), portanto adotou-se o seguinte procedimento: todo Module indica os recursos necessários, e uma instância pode ser criada apenas se o MOSDevice tiver estes recursos disponíveis. A ModIn precisa saber quais recursos foram alocados para si, e ela possui um pedaço de memória que representa a ModClass. Lá, existe um espaço que indica quais recursos serão usados, e esta posição é outro item da tabela. Outro dado importante na primeira tabela é o endereço da função de inicialização da ModClass dentro do primeiro setor.

Para um CMod ser transformado em um RMod é preciso corrigir o código de máquina que está no primeiro setor. Considere o caso da compilação para ATmega2560. A chamada de função é vista como uma subrotina, e usa-se instruções de 16 bits como a CALL. Isto faz com que o Program Counter (registrador que aponta para a próxima instrução a ser executada) mude para o endereço na memória que está no registrador Z. Esse registrador é preenchido com um valor constante em uma instrução anterior. É este valor que deve ser alterado. A segunda tabela (terceiro setor) indica a posição de todas instruções relacionadas a funções da ModClass. A terceira tabela (quarto setor) são de funções relacionadas ao System.

Para ajustar o endereçamento de funções de uma ModClass (segunda tabela) basta adicionar o endereço de onde o RMod ficará na memória pois no CMod os valores já apontam para as funções como se o CMod estivesse na posição 0 de memória. Nas funções do System (terceira tabela), ao invés das instruções conterem o endereço das funções do System, possuem um ID (identificador) que representa a função, e quando o CMod chegar no destino, o MOSBuilder irá substituir os IDs

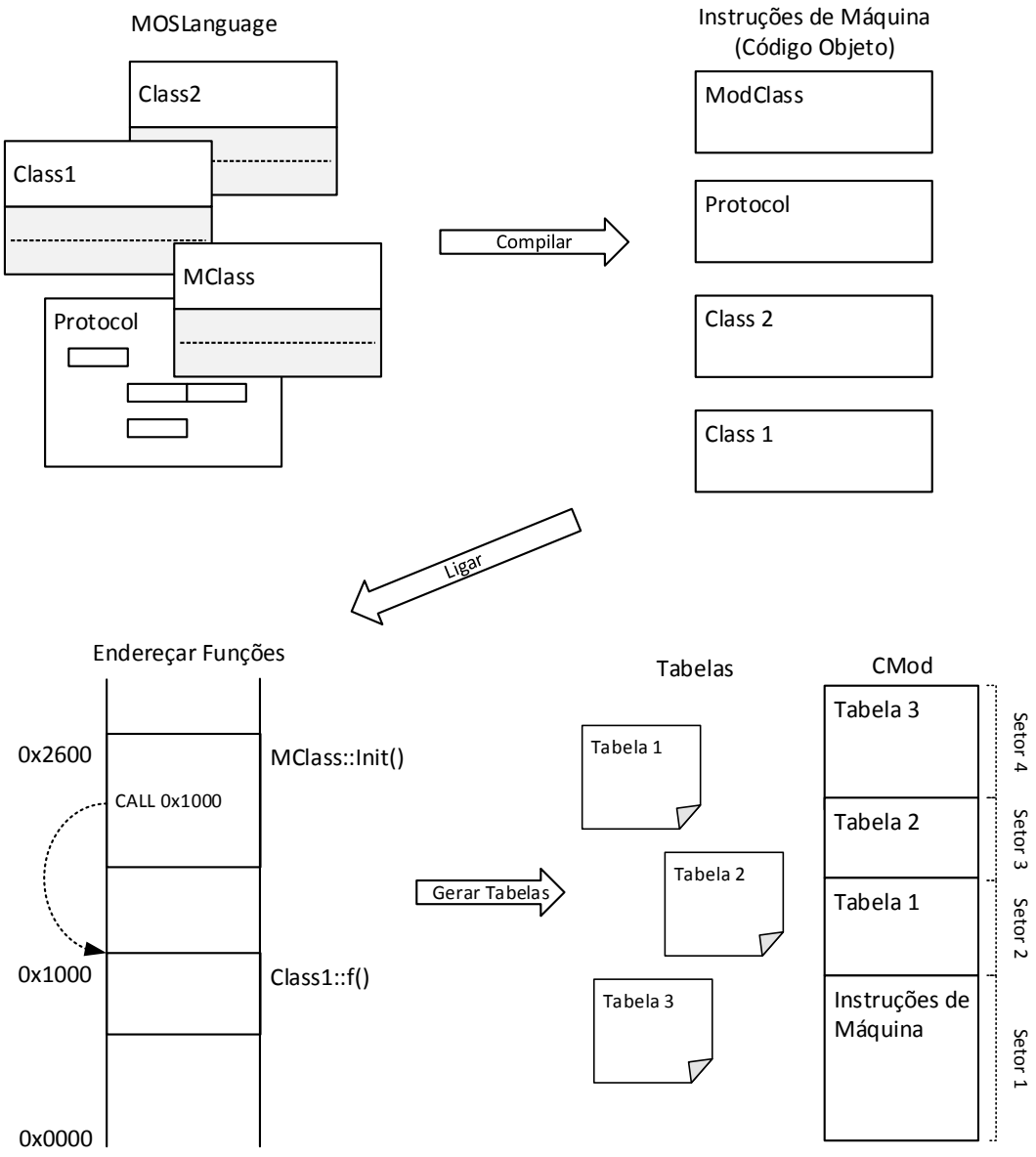

Figura 2.5: Etapa 1 do MOSBuilder.

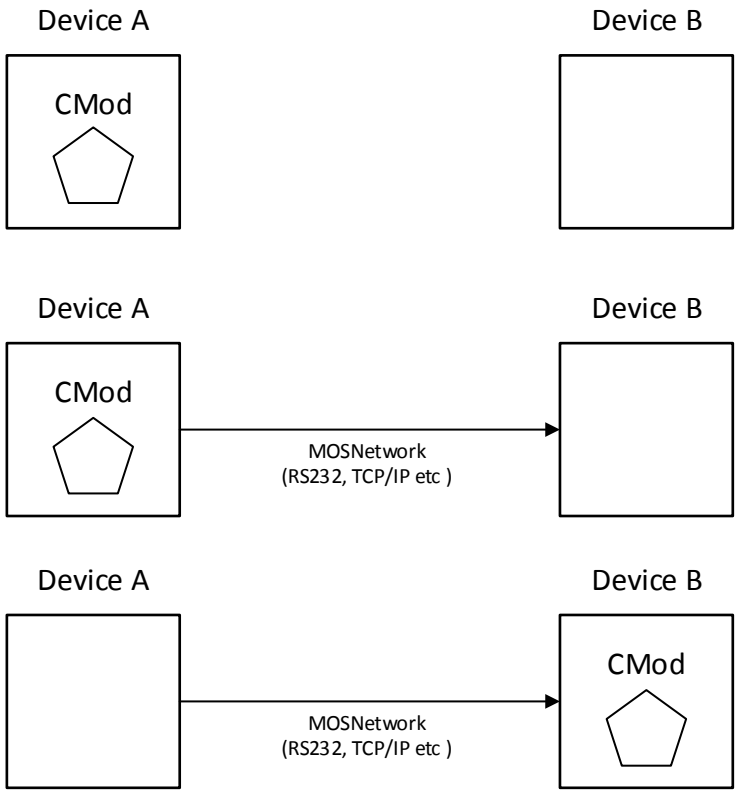

Figura 2.6: Etapa 2 do MOSBuilder.

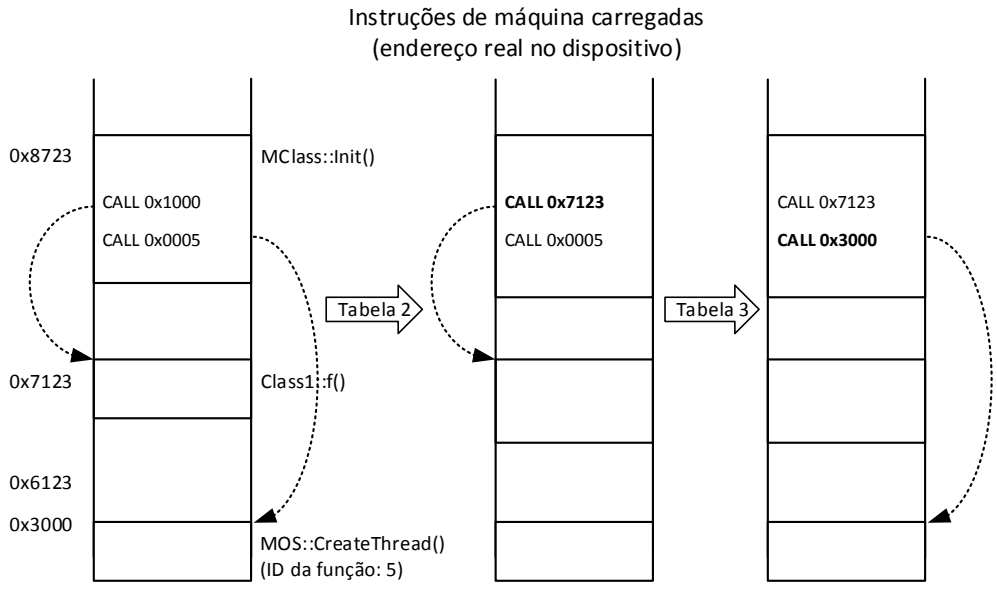

Figura 2.7: Etapa 3 do MOSBuilder.

pelo endereço real das funções do System. A figura 2.7 exemplifica como as tabelas são usadas pelo MOSBuilder para corrigir o endereçamento. A correção usando a tabela 3 é parecida com a Relocação[44], e a tabelas 3 parecida com o Dicionário de Relocação. O procedimento de corrigir o endereçamento com a tabela 2 é chamado de Carga[44].

Considera-se que os procedimento relacionados ao uso das tabelas fazem parte do MOSBuilder (dentro do procedimento de ligar um código objeto), ou seja, o MOSBuilder tem um comportamento semelhante ao JIT Compiler.

O MOSBuilder é um elemento chave dentro do MOS, mas, como dito anteriormente, uma parte do MOSBuilder (a parte que compila) existirá apenas nos MOSProcess pelos MOSMicros terem uma capacidade de processamento muito limitada.

Apesar do ATmega2560 ter sido usado como exemplo, o mesmo aconteceria para todos os outros MOSDeviceTypes mesmo tendo implementações diferentes do MOSBuilder. Uma das formas mais fáceis do MOSBuilder compilar a MOSLanguage seria transformá-lo em um código de C++ e usar um compiladores de C++ para cada MOSDeviceType, como o GCC [45] ou Microsoft Visual C++. O inconveniente neste caso é extrair de um compilador as tabelas que compõem o CMod, mas como as tabelas estão relacionadas às etapas de Ligação, Relocação, Carga e etc. da compilação e execução de programas. Estes compiladores teriam de ser modificados e separados para realizar as etapas 1 e 3. Outra alternativa muito mais trabalhosa seria criar o MOSBuilder do zero, traduzindo a MOSLanguage direto para o assembly do conjunto de instruções de máquina de cada MOSDeviceType. O MOSBuilder é um item importante no MOS mas sua implementação não é tão importante para explicar o funcionamento do MOS. Basta entender do que um CMod é composto e o que deve ser feito para transformá-lo em um RMod.

2.7 NativeModules e Meio Externo

O MOS não permite que programadores desenvolvam seus Modules em C++ para impedir que Modules tenham acesso irrestrito dentro do sistema. A MOSLanguage também não possui vários recursos de OSs quando comparada com o C++.

Uma das ideias por trás do MOS é fornecer os meios de comunicação de forma fácil ao programador. Isto poderia ser alcançado criando uma camada em cima das APIs de rede, sendo APIs do próprio OS ou não. No entanto, existem muitas bibliotecas/APIs (antigas e/ou extensivamente testadas) que não contém nenhuma relação com redes mas que possuem grande importância para certos tipos de aplicação. Uma possível dificuldade para a aceitação do MOS é justamente a falta deste tipo de recurso. Existem duas alternativas para este problema: Reescrever as bibliotecas/APIs para a MOSLanguage, ou criar uma casca sobre elas. O problema do

primeiro caso é a ausência de alocação dinâmica na MOSLanguage. O segundo caso é o que chamaremos de NativeModules.

NativeModules são Modules especiais, escritos em C++ que complementam as funcionalidades do MOS. Estes Modules em geral possuem restrições sobre para quais MOSDeviceTypes são compiláveis (por exemplo apenas para Windows e Linux). Também não podem ser transferidos para outros MOSDevices, e por isto recebem o nome de NativeModules. Podem existir várias ModIn do mesmo NativeModule. Uma vantagem é que por serem Modules no MOS, possuem os meios de comunicação (MOSNetwork) e conexão (MOSLink) com outros Modules. Isto gera a possibilidade de NativeModules serem "acessados" remotamente, ou seja, um ModIn de um NativeModule em um MOSDevice pode se conectar com um ModIn de outro MOSDevice.

Um ponto importante na criação de um NativeModule em C++ é a dificuldade em reproduzir as funcionalidades da MOSLanguage sem erros. Por exemplo, quando um Module acessa outro com o MOSLink, o programador na MOSLanguage não precisa se preocupar como isto ocorre já que esta é a responsabilidade do compilador da MOSLanguage. Esta conexão feito no C++ requer o conhecimento do procedimento de MOSLink, que deve ser reproduzido em C++ para evitar bugs.

O MOS disponibiliza um conjunto de classes em C++ que correspondem à redes do MOS específicas de OSs, no formato de uma API (chamada de MOSAPI). Estas redes são as mesmas usadas para conectar MOSDevices dentro de um computador. São redes que trabalham em cima dos mecanismos de comunicação entre processos presentes em OSs. Isto permite que uma ModIn se comunique diretamente com um processo que use a MOSAPI.

Com isto, existem duas formas para que Modules acessem um meio externo ao MOS mas dentro de um computador, com a MOSAPI ("redes" dentro de um computador) ou com os NativeModules.

NativeModules podem ser desenvolvidos de acordo com a necessidade, como por exemplo um Module para Qt. Outro exemplo seriam Modules que criam interfaces para redes não suportadas pelo MOS, ou interfaces para outros frameworks (ex: ROS). Isto possibilitaria um ModIn se comunicar com um processo criado fora do MOS.

O MOS não possui funções para escrever texto em um terminal. O motivo desta limitação é a impossibilidade de reproduzir estas funcionalidades nos MOSMicros. A princípio existe apenas um NativeModule definido no MOS, o MOSTerminal. Este Module é capaz de escrever ou receber informações de um terminal (Linux) ou console (Windows). Desta forma, Modules podem criar um MOSLink com um ModIn do MOSTerminal. Nos dois sistemas operacionais é possível associar um processo (no caso, o próprio MOSProcess) a uma série de terminais/consoles usando

APIs nativas. O MOSTerminal também deve ser capaz de enviar e receber caracteres pela MOSNetwork, tornando-o um terminal remoto.

2.8 Threads

No Windows ou no Linux um MOSDevice é um MOSProcess, portanto é possível ter vários MOSDevices dentro de um computador, mas um MOSDevice não tem acesso à memória do outro. No MOS, a ModIn possui um pedaço de memória reservado. ModIns em um MOSDevice são comparáveis a processos em um SOs, porém, uma ModIn tem acesso à memória de outra ModIn. Sendo mais preciso, os threads no MOS podem acessar a área reservada de qualquer ModIn dentro de um MOSDevice.

Para demonstrar este conceito focaremos primeiro na interação entre um thread e uma única ModIn. Threads executam uma função definida em uma ModClass. Esta função não é uma função global, e sim um método do Module descrito na ModClass, ou seja, o thread acessa o espaço reservado da ModIn. Vale lembrar que para o programador um Module é uma classe com variáveis e funções, e que portanto o thread de Module pode usar as variáveis e funções do Module sem ter de referenciar o Module.

Considere o pseudo-código abaixo como um exemplo:

Código 2.1: Exemplo de ModClass

```
Class C1 {
  double var = 3;
}

Module M1 {
  InitialThreads: ThreadFunction;

  C1 obj;

  [double ret] VarHalf() {
    ret = obj.var/2}
  [] ThreadFunction() {
    var += VarHalf()}
}
```

O código define a classe *C1* e o Modules *M1*. A classe *C1* possui apenas 1 objeto double que é iniciado com o valor 3.

O Module *M1* possui uma variável *obj*, uma função *VarHalf()* e uma função especial *ThreadFunction()*. O *InitialThreads: ThreadFunction* faz um thread ser criado

no início da execução do Module que roda a função *ThreadFunction()*. A função soma à *var* a metade de seu próprio valor. Quando o Module *M1* for instanciado (ModIn de *M1*), um thread será criado e rodará *ThreadFunction()*. Ao término da função o thread é destruído e, neste caso, o valor de *var* será 4,5. Note que a função é um método da classe *M1*, portanto *var* e *VarHalf()* podem ser acessados diretamente, sem usar referencias no thread.

Um fator expressivo na diferença entre processos e Modules é a relação destes com threads. Cada processo tem um conjunto de threads. Quando um processo termina, todos seus threads são destruídos. No MOS todo thread sabe qual ModIn o criou, mas o thread não pertence à ModIn. A natureza do MOS permite que um thread acessando uma ModIn (tanto a memória da ModIn quanto a execução das instruções no RMod associado) acesse outra ModIn. Para o programador na MOSLanguage isto equivale a chamar uma função ou uma variável de um Module conectado. Desta forma, o thread deve voltar a acessar o Module criador, no retorno da função.

A destruição de uma ModIn é complexa. Quando destruído, todos os threads que a ModIn criou deveriam ser destruídos. No entanto, existem threads que não pertencem à ModIn mas a estão acessando, e seus próprios threads podem estar acessando outros ModIn. O MOS possui duas versões para destruição de Modules, a normal, e a forçada.

Na destruição normal de uma ModIn, o MOS espera que todos seus threads voltem para seu criador para que sejam devidamente destruídos, e que threads de outras ModIns deixem de acessá-la. Note que este procedimento depende consideravelmente dos programadores, já que os threads podem acessar uma ModIn (chamando uma função) e nunca mais retornar à ModIn original (o programador define a função).

A destruição forçada consiste em imediatamente destruir todos os threads que estão acessando certa ModIn. Isto vale também para os threads que não foram criados pela ModIn. Threads que a ModIn criou mas que estão acessando outra ModIn permanecem intactos. Quando estes threads tentarem voltar à ModIn criadora (não existe mais), o thread será destruído.

Para representar esta relação, considere o Código 2.2 como uma expansão do Código 2.1:

Código 2.2: Exemplo de ModClasses

```
Module M3{
public:
        Function3()
}
```

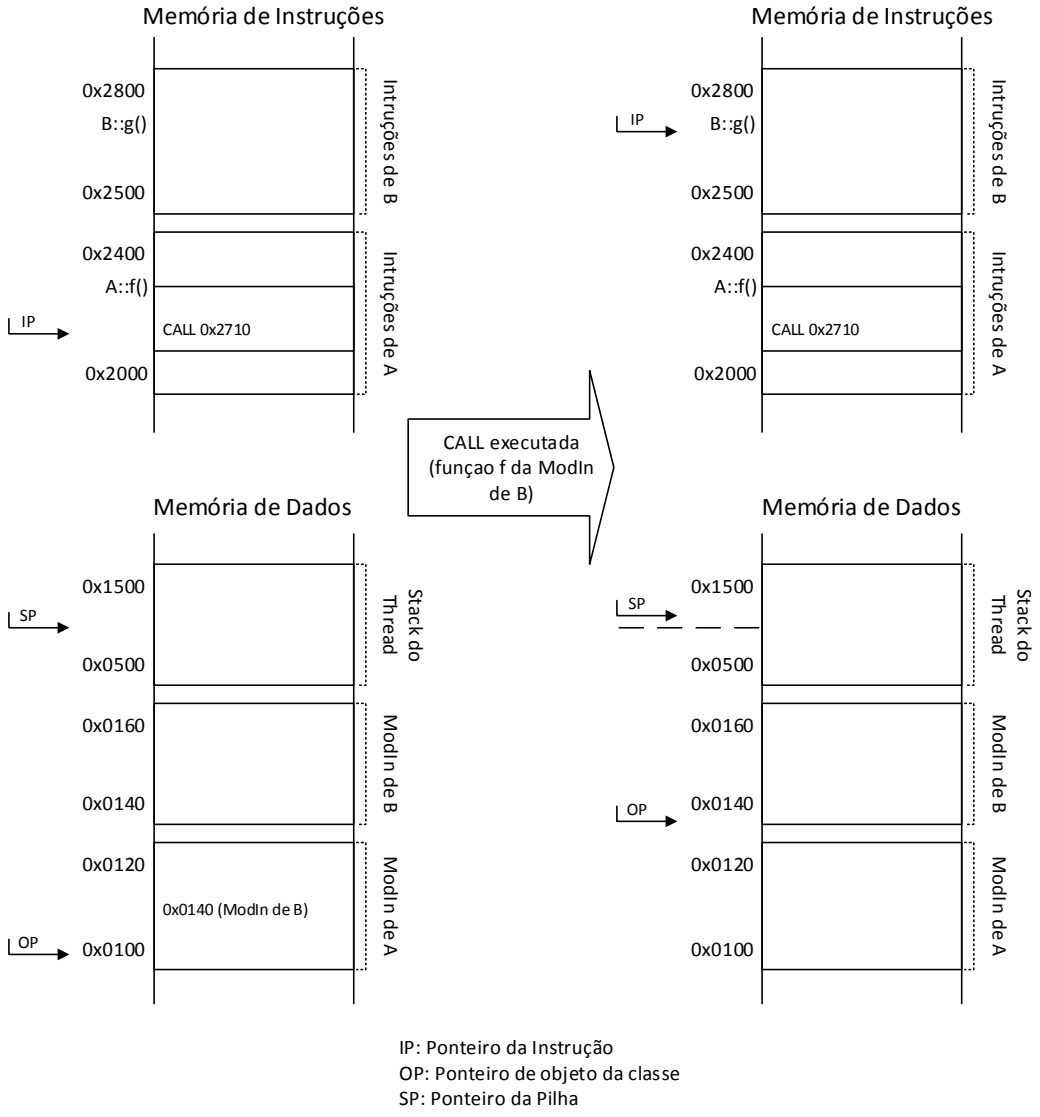

Figura 2.8: Thread acessando ModIns.

```
Module M2{
        M3 module3
public:
        Function2(){
                ...
               module3.Function3()
                ...
        }
}

Module M1{
         ...
         M2 module2
public:
         ThreadFunction2(){
                ...
                module2.Function2()
                ...
         }
         ...
}
```

O código mostra 2 outros Modules *M2* e *M3*, e uma modificações no Module *M1*. As relações entre *M1* com *M2*, e *M2* com *M3* no MOS equivalem a uma conexão entre estes Modules chamada no MOS de MOSLink, e será comentada em seguida. Aqui, um Module é um objeto de outro, e com isto um pode chamar funções e variáveis do outro. O *M1* possui uma outra função de thread *ThreadFunction2()*.

Considere que existe uma ModIn para cada Module, e que todos estão devidamente conectados (MOSLink), e que *ThreadFunction2()* foi executada em um thread. Eventualmente *Function2()* do *module2* é chamada, e o thread sai do *module1* e vai para o *module2*. Depois o thread sai do *module2* e entra no *module3* quando *Function3()* é chamada na *Function2()*.

Analisaremos agora a destruição forçada de um desses modules com o thread criado. Imagine que o thread está executando *Function3()*. Se o *module2* for destruído nada acontece de imediato com o thread, mas quando o thread tentar voltar à execução de *Function2()* o thread será destruído. Se o *module1* for destruído o thread deve ser destruído imediatamente, não importando em que Module ele está, pois o *module1* é o criador do thread. Se o *module3* for destruído, o thread também deve ser destruído de imediato por estar acessando o thread que está sendo destruído.

2.9 Objetos de Sincronização

Recurso em um Sistema Operacional é um componente do sistema com uma disponibilidade limitada. Os tipos mais comuns de recurso são o tempo de CPU (apenas um thread/processo pode ser executado por vez em um núcleo da CPU), memória (física ou virtual [14]), disco rígido, redes, dispositivos externos e operações de Input/Output. O MOS utiliza uma série de recursos, seja no MOSProcesses ou nos MOSMicros. O único dos citados que não é tratado no MOS são os dispositivos externos.

A maioria destes recursos já são tratados automaticamente pelo MOS. Nos MOSProcesses o MOS encapsula as bibliotecas dos SOs para usar o disco rígido ou redes. Os MOSMicros possuem entradas e saídas analógicas e digitais que são gerenciadas pelo MOS. O Kernel (OS ou Kernel dos do MOS) decide qual thread do MOS será executado. O mesmo vale para a memória com a diferença que as instruções de máquina do MOS em C++ e dos Modules na MOSLanguage determinam a alocação na RAM.

Apesar dos recursos estarem disponíveis para o programador muitas vezes eles não podem ser usados em paralelo. Um dos grandes enfoques do MOS é o uso de threads, especialmente por permitir o uso de threads em microcontroladores. É essencial acrescentar no MOS proteção para estes recursos.

O exemplo clássico é o de acesso simultâneo de uma região da memória RAM por threads. Considere que uma variável é vista por dois threads. A variável possui um valor atual. Um thread tem o novo valor da variável, então tentará atualizá-lo. Outro thread deve usar o valor atual para executar em um algoritmo. Se um escreve na posição de memória enquanto o outro faz a leitura é possível que a leitura seja uma mistura (em bytes) dos valores atual e novo. O mesmo problema aconteceria se dois threads tentassem escrever valores na variável. O novo valor da variável poderia ser a mistura dos valores que deveriam ser escritos.

A boa prática de programação diz que objetos de sincronização de threads são necessários nessa situação. Mas o que são estes objetos? Os recursos são vistos como regiões críticas [14]. Quando um recurso é usado ou acessado a região crítica é acessada. Nem todo thread pode/deve acessar uma região crítica ao mesmo tempo. Os objetos basicamente impedem que as regiões sejam acessadas quando não devem. Impedir o acesso em geral consiste em bloquear o thread até a região estar novamente acessível.

Um Mutex (Mutual Exclusion ou Exclusão Mútua) [14] é uma das técnicas mais simples de sincronização. Neste caso a região crítica só pode ser acessada por apenas um único thread por vez. Quando um thread acessa a região crítica e nenhum outro thread a está acessando, o primeiro thread toma a região para si. Diz-se então que o

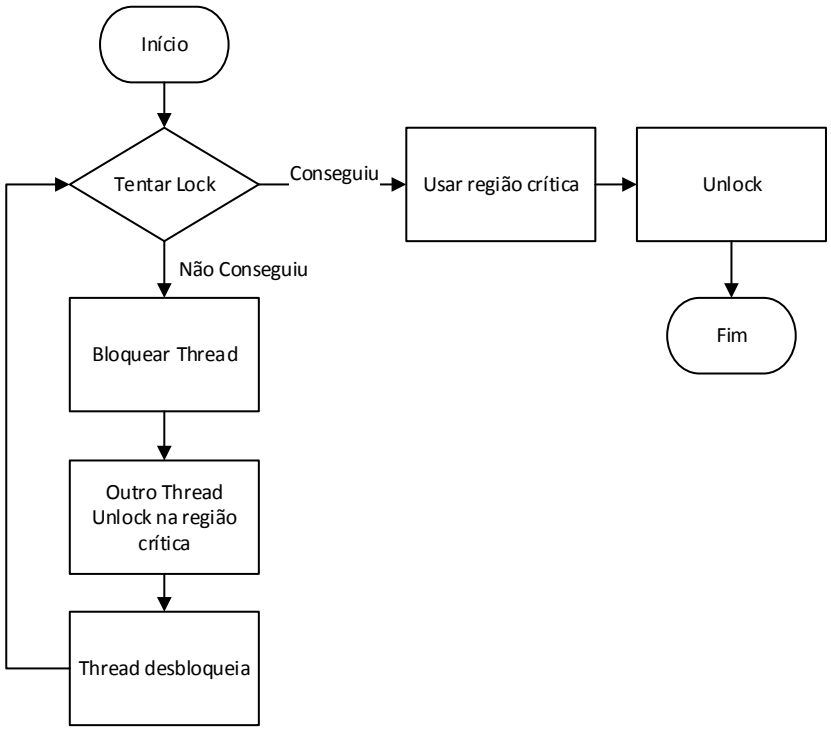

Figura 2.9: Procedimentos com um mutex.

thread travou a região (Lock). Se outro thread tentar acessar a região travada não conseguirá e ficará aguardando (thread bloqueado) o thread ser destravado. Quando o thread acessando destravar a região (Unlock) outro thread poderá acessar a região. Os threads bloqueados despertarão e um deles trava a região novamente. A região só volta a ficar livre após todos os threads destravarem a região. A figura 2.9 mostra as etapas de acesso em um Mutex.

Os objetos de sincronização são representados muitas vezes como uma série de funções de bibliotecasao invés de classes/objetos. Para o Mutex as funções associadas seriam as *Lock()* e *Unlock()* que receberiam o identificador do Mutex. As funções para travar e destravar controlam o fluxo de processamento do thread, e não os recursos em si.

Não vale a pena explicar os vários tipos de objetos que podem estar definidos na MOSLanguage. Citaremos os mais importantes:

Mutex (Mutual Exclusion) Apenas um thread acessa a região crítica por vez. Quando um thread tenta acessar um região que está sendo acessada por outro, o thread é bloqueado. O thread desbloqueia quando ganhar o acesso.

Semaphore Considere o caso em que uma região crítica possui um número máximo de acessos simultâneos. Um contador marca quando um thread entra e sai da região. Se o contador estiver em 0 os próximos threads que tentarem acessar

ficarão bloqueados, ou seja, impede o acesso como um semáforo [14].

Monitor Um Monitor [14] é constituído de um Mutex e uma variável condicional. Um thread acessando uma região com um Mutex pode renunciar ao acesso e se bloquear até que uma condição seja satisfeita. Neste momento outros travar a região com o Mutex. Quando a condição da variável condicional é satisfeita o thread tenta novamente acessar a região. Quando o thread travar novamente a região com o Mutex voltará a executar normalmente, eventualmente destravando a região.

Read And Write Mutex O acesso de uma região crítica é dividida entre os que leem a região e os que escrevem na região. A escrita funciona como um Mutex, apenas um pode escrever na região por vez. Como a leitura não modifica a região, múltiplos threads podem ler a região ao mesmo tempo. Com a região sendo lida nenhum thread que tente ler será bloqueado até algum thread tentar escrever. Quando a região está sendo escrita os threads que tentarem ler ou escrever serão bloqueados.

O Kernel é responsável por gerenciar a execução dos threads, então o Kernel e os objetos de sincronização estão interligados. SOs modernos possuem bibliotecas nativas para a manipulação de threads e uso dos objetos. Alguns dos objetos não são fornecidos diretamente, portanto alguns objetos seriam criados em cima dos básicos (ex: Monitor no Windows seria criado com Mutex e Conditional Variables da biblioteca nativa). Nos MOSMicros o Kernel do MOS deve ser implementado tendo em mente estes objetos. A própria implementação do MOS em C++ usa estes objetos pois é um sistema multithreads.

A MOSLanguage define classes que representam objetos de sincronização para todos os MOSDeviceTypes, como as classes *Mutex*, *Semaphore*, *Monitor* e *ReadWriteMutex*. A *Mutex* teria as funções *Lock()* e *Unlock*. Desta forma os objetos de sincronização ficam fáceis de serem usados na MOSLanguage. Classes definidas pelos programadores podem conter objetos das classes de sincronização, de forma que os threads (no MOS os threads estão vinculados à classes) podem acessar os objetos de sincronização para realizar seus algoritmos.

Os objetos de sincronização não podem ser alocados dinamicamente. Esta regra foi imposta pelo problema de alocação dinâmica nos MOSMicros. Por isto os objetos de sincronização são vistos como recursos dentro do MOS. As classes ou ModClasses podem conter estes objetos, mas não podem criá-los dentro de suas funções. Quando a ModClass for compilada o CMod conterá a quantidade e tipo de cada objeto que a ModClass usa. Para uma ModIn ser instanciada de um RMod verifica-se se existem objetos suficientes disponíveis, que então são alocados para a Instância.

Os objetos de sincronização são objetos privados, ou seja, uma ModIn não consegue acessar outra para usar o objeto. A ModIn deve entrar na outra (função é chamada) e esta sim usa o objeto. Isto é necessário para não gerar conflitos dentro do MOS relacionados à destruição de ModIns e liberação dos objetos (que são vistos como recursos).

2.10 Link

Modules são capazes de acessar o meio externo acessando uma rede no MOSDevice. Existe também a comunicação dentro de um MOSDevice, realizada entre dois Modules. Ela é chamada de MOSLink.

Uma das limitações do MOS é a falta de alocação de dinâmica, proveniente de MOSMicros. O MOSLink tenta suprir esta limitação ao permitir que as conexões sejam feitas em tempo de execução. Ao invés de um objeto ser alocado dinamicamente, uma ModIn pode pedir ao MOS a criação de um Module para estabelecer um MOSLink.

O melhor caminho para representar o MOSLink é identificar seu funcionamento na MOSLanguage. Quando o programador define uma ModClass, ele pode optar por usar objetos de outras ModClasses. O Código 2.2 ilustra esta situação. O Module *M1* contém um *M2*, e o *M2* contém um *M3*.

O nome escolhido para este tipo de conexão (MOSLink) foi inspirado no link (ou hyperlink) da linguagem de marcação HTML (HyperText Markup Language). Considere o exemplo: Um link em uma página A leva o browser para outra página B. Uma outra página C pode conter o mesmo endereço de link que leva o browser para B. Links representam uma conexão de apenas um sentido, de forma que B não conhece A ou C. Algo similar ocorre no MOSLink, e por esta razão existe o LinkOwner e o LinkEnd. Pelo exemplo, LinkOwner é o Module representado pelas páginas A e C, e o LinkEnd é o Module representado por B. É importante ressaltar que o modelo representa a etapa de conexão entre os Modules e não a de comunicação. Na conexão, o LinkOwner sabe quem é o LinkEnd, mas o LinkEnd nunca sabe quem é o LinkOwner. Na comunicação, mesmo que de forma limitada, o LinkEnd consegue se comunicar com o LinkOwner.

Com o MOSLink existe duas formas do LinkOwner e LinkEnd se comunicarem. A primeira é uma abordagem direta (MOSLinkDirect). O LinkOwner usa variáveis ou chama funções do LinkEnd, como comentado na seção 2.8. A segunda é um mecanismo de passagem de mensagens apelidado de MOSLinkMP (MOSLink Message Passing). O LinkEnd define uma série de mensagens. Tanto o LinkEnd quanto o LinkOwner podem enviar e/ou definir o tratamento destas mensagens.

Os MOSProjects definem ModClasses com a MOSLanguage. No protótipo de

uma ModClass o programador pode declarar objetos de outras ModClasses. Cada um destes objetos está associado a um MOSLink, com a ModClass sendo o LinkOwner e os objetos sendo os LinkEnd. Quando esta ModClass for instanciada em uma ModIn, nenhum MOSLink está usável. Não se aloca espaço para estes objetos. Eles são na realidade referências que iniciam vazias. Algum thread da ModIn deve requisitar ao MOS a conexão com um LinkEnd. Este procedimento foi apresentado em 2.2, e leva em consideração PName, MName, MVersion, MPubVersion. Se o sistema encontrar uma ModIn compatível, o MOSLink torna-se usável.

Antes de prosseguir, uma observação será feita sobre compiladores e linguagens orientada a objetos. Funções de classes podem acessar seus objetos (copiar ou modificar o valor destas). Um registrador armazena o endereço de memória do objeto. As funções de classes acessam os objetos da classe somando um valor a este endereço. O compilador decide o valor para cada objeto. O acesso de objetos de objetos utiliza o mesmo mecanismo. Quando tem-se a referência de um objeto (endereço na memória), o código de máquina somará um valor a esta referência ao invés do endereço no registrador. No entanto, quando deseja-se usar uma função de um objeto, o escopo deve mudar. Alguns registradores são armazenados na Pilha de Chamada, os registradores ganham novos valores, e o processador chama a função. Quando a função retorna os registradores são carregados com os valores na Pilha. Aquele registrador que guardava o endereço da classe teve seu endereço modificado até ser recuperado da Pilha.

O MOSLink tem uma similaridade com o procedimento apresentado. Existem termos relacionados ao MOSLink a serem mencionados: Acesso, Entrada e Retorno. Diz-se que um thread em um Module *MA* acessa um Module *MB* quando uma função de *MA* acessa um dos objetos de *MB*. Um thread em *MA* entra em *MB* quando uma função de *MA* chama uma função de *MB*. Quando esta função é finalizada, o *MB* retorna ao *MA*. Esta distinção é necessária pois o MOS e a MOSLanguage tratam os dois casos de maneiras diferentes.

O MOS possui um vetor de dados que representa todos os MOSLinks no MOSDevice. Cada item do vetor armazena o LinkOwner, o LinkEnd e outros dados. Quando uma ModIn é criada, uma parte do vetor é preenchido para representar os novos MOSLinks da ModIn (os LinkEnd não são preenchidos de início). A ModIn guarda os IDs do MOSLink e o item do vetor guarda a ModIn como LinkOwner. Quando um thread é criado ele é associado à ModIn criadora, e dois MOSLinks (um para Acesso e outro para Entrada). Se o thread acessar uma ModIn, o thread armazenará o ID do MOSLink sendo acessado enquanto faz o acesso. Se o thread entrar em um LinkEnd de um MOSLink, o thread armazenará o ID deste MOSLink, até retornar do MOSLink. Note que threads podem entrar consecutivamente em Modules. O thread armazena apenas o ID do último MOSLink que ocorreu a entrada,

no entanto, o ID anterior fica armazenado na Pilha, de forma que após o retorno da entrada o ID anterior é restaurado, permitindo que o thread volte eventualmente à ModIn que o criou (considerando que o programador não criou um loop infinito no meio do caminho).

A informação de qual ModIn o thread está acessando está sempre disponível. Isto é muito importantes na destruição de ModIns apresentada em 2.8. Sempre que um thread tentar retornar ao MOSOwner, o sistema irá verificar se o MOSOwner ainda existe, e se não existir este thread será terminado. Com isto não ocorrerá nenhum bug de acesso indevido de memória.

2.10.1 MOSLinkDirect

Na MOSLanguage, o MOSLinkDirect consiste no programador usar variáveis e funções públicas dos LinkEnd como se fossem objetos normais. Funções na MOSLanguage podem ter mais de um retorno. Quando uma função de um MOSLinkEnd é chamada, existe um retorno extra booleano que indica se a função foi executada com êxito ou não. A falha ocorre quando o MOSLink não foi estabelecido com um LinkEnd. O mesmo erro ocorre no acesso aos objetos do LinkEnd, mas o programador não sabe se o acesso ocorreu corretamente ou não. A MOSLanguage será estudada com detalhes no capítulo 3, portanto o Código 2.3 (em pseudo código) será apresentado aqui para mostrar esta diferença.

Código 2.3: Exemplo de acesso e entrada em um MOSLink

```
Module M1{
public:
      int var;
      [int, double] func();
}

Module M2{
      M1 m1
      int copyvar;
      int ret1;
      double ret2;
      void ThreadFunction(){
            copyvar = m1.var;
            if ([ ,ret1, ret2]m1.func()[0])
            {...}
      }
}
```

O código apresenta dois Modules *M1* e *M2*. *M2* tem um objeto de *M1* chamado *m1*, que deve ser conectado com um objeto real (MOSLink) para ser usado sem erros. A função *ThreadFunction()* de *M2* acessa *var* em *m1* (muda o valor) e depois entra em *m1* ao chamar a função *func()*. No acesso a *m1*, não se verifica se o objeto foi usado ou não. Se o MOSLink não estiver estabelecido, o valor de *copyvar* não mudará e o programador não saberá. Analisando a entrada em *m1*, se *M1* define *[int double] func();*, com dois retornos, os Modules que se conectarão com *M1* enxergarão *[bool, int, double] func();*, com um retorno extra booleano que indica se a função foi executada ou não. O *if ([,ret1, ret2]m1.func()[0])* indica que o primeiro retorno da função (booleano) será usado no if. Desta forma o programador sabe se a função falhou (indica se *ret1* e *ret2* foram preenchidos com o resultado da função ou não).

Pelo problema encontrado na sintaxe do acesso no MOSLink, a MOSLanguage propõe a criação de um bloco de conexão (LinkBlock), que só será executado se certo MOSLink estiver devidamente conectado. O Código 2.4 é uma modificação do Código 2.3 que utiliza este bloco. Dentro deste bloco, garante-se que a conexão estará estabelecida e que esta não será desfeita até o fim do bloco (o LinkEnd não será destruído nem desconectado do LinkOwner).

Código 2.4: Exemplo de acesso e entrada em um MOSLink usando um LinkBlock

```
void ThreadFunction(){
        LinkBlock(m1){
                copyvar = m1.var;
                [ ,ret1, ret2]m1.func()
                ....
        }
}
```

Para o usuário o mecanismo de acesso e entrada em LinkEnds é transparente, mas as instruções de máquina geradas da MOSLanguage invocam uma série de ifs e funções do sistema que verificam o estado do MOSLink para então acessar objetos ou chamar funções do LinkEnd. Dentro desta etapa objetos de sincronização são usados para impedir a desconexão do MOSLink ou a destruição de seus membros. Este procedimento gera um custo computacional adicional em relação a um simples acesso a objetos de uma classe. Os programadores devem tomar cuidado em como usam o acesso e entrada no MOSLink.

A interação entre Modules ocorre apenas entre LinkOwner e LinkEnd. Considere novamente o Código 2.2. Esta afirmação se traduz no *M1* não poder acessar a *Function3()* diretamente, já que *module2* não é pública. Esta limitação se encontrará na MOSLanguage, de forma que na definição de uma ModClass, a declaração de objetos de outras ModClasses não pode ser pública. Este mecanismo foi adotado

para garantir que um thread esteja em apenas um Module e no máximo acessando outro Module (LinkEnd). Caso contrário seria possível chamar a função *Function3()* do *module1* com *module2.module3.Function3()*, ou seja, o *module1* acessa o *module3* mesmo sem um MOSLink entre os dois. O MOSLink descrito neste trabalho não permite este tipo de acesso.

2.10.2 MOSLinkMP

Um outro aspecto no MOSLink é quantidade de MOSLinks. Quando uma ModIn é instanciada, um número fixo de MOSLinks é criado também. O LinkOwner é fixo nestes MOSLink, mas o LinkEnd é escolhido dinamicamente. Um Module pode ser LinkEnd de múltiplos MOSLinks, mas esta quantidade é definida pelo programador na ModClass. Um LinkSlot em uma ModIn representa a possibilidade desta ser um LinkEnd de um MOSLink. Na definição da ModClass o programador define o número de LinkSlots do Module, ou seja, define o número de vezes que a ModIn será um LinkEnd. Quando o MOSLink é desfeito, o LinkSlot é liberado, permitindo que o Module seja LinkEnd de um novo MOSLink. Cabe ao programador definir como o Module se comporta. Às vezes um Module nunca deve ser um LinkEnd, em outros casos um Module só pode ser LinkEnd de um único MOSLink, etc.

Os LinkSlots são preenchidos com o ID dos MOSLinks armazenados no MOS. Esta informação permitiria que um LinkEnd conhecesse o LinkOwner mas isto não fica disponível na MOSLanguage. No entanto, o MOSLinkMP funciona por isto. Esta informação permite que o LinkEnd "acesse" todos os seus LinkOwners, mesmo sem conhecê-los.

O MOSLinkMP consiste em uma ModClasses definir mensagens, e que tanto esta ModClasses quanto outras conectadas por MOSLinks possam tratar estas mensagens. O programador cria mensagens em ModClasses, indicando os dados a serem transferidos. Automaticamente são criadas funções para que estas mensagens possam ser enviadas (dentro do escopo da ModClass). Depois, é possível definir o tratamento destas mensagens, mas não é necessário. Supondo a existência das ModClasses *MA* e *MB*, sendo que *MB* possui um objeto de *MA*, *MB* herda todas as mensagens de *MA*. Isto significa que *MB* pode enviá-las e definir o tratamento destas.

Quando um thread em um Module chama a função para enviar uma mensagem, a mensagem não é tratada no mesmo thread porque o MOS possui um thread para realizar o tratamento de todas as mensagens do MOSLinkMP, o MPThread (Message Passing Thread). Existe um mecanismo de sincronização entre o thread que faz o pedido de envio e o MPThread. O thread do pedido trava até que o MPThread esteja livre para tratar a mensagem. O programador decide se o pedido irá travar

até a mensagem começar a ser tratada ou até o tratamento terminar.

Em geral, os thread no MOS são criados por uma ModIn e podem acessar/entrar outras ModIns, mas não o MPThread. Ele é criado pelo próprio MOS. Mas este thread ainda entra em outras ModIns pois o tratamento está definido nas ModIns. Isto significa que antes de iniciar o tratamento, o MOS deve criar um MOSLink com as ModIns em questão. Toda ModIn possui um LinkSlot separado para o tratamento de mensagens. O MPThread está sendo usado no MOSLinkMP pois os threads vão de um LinkOwner para um LinkEnd e depois retornam ao LinkOwner, e o contrário é "proibido", mas as mensagens do MOSLinkMP precisam ser tratadas pelos LinkOwners mesmo quando o LinkEnd envia a mensagem.

Considere novamente *MA* e *MB*, e que os dois definiram o tratamento de uma mensagem *msg* definida em *MB*. Existem duas formas para executar o tratamento desta mensagem, quando *MA* envia a mensagem e quando *MB* envia a mensagem.

Se *MA* enviar a *msg*, o MPThread cria um MOSLink, entra em *MA* e trata a mensagem. Depois de retornar, o thread acessa *MA* para ver o LinkOwner do seu primeiro LinkSlot. Em seguida, supondo que o primeiro é *MB*, o MPThread cria um MOSLink com *MB*, entra em *MB*, faz o tratamento, e retorna. O MPThread acessa novamente *MA*, busca o próximo LinkOwner, e repete o processo até que os LinkSlots terminem. Procedimento descrito na figura 2.10.

Se *MB* enviar a *msg*, o MPThread cria um MOSLink, entra em *MB* e trata a mensagem, depois retorna. Quando *MB* define o tratamento de *msg*, define para um certo objeto da ModClass de *MA*, então está claro qual MOSLink será usado no MOSLinkMP. O MPThread acessa *MB* para ver o LinkEnd do MOSLink (objeto de *MA*) e cria um MOSLink com este. O MPThread entra em *MA*, trata a mensagem e retorna.

Quando uma mensagem definida na LinkEnd (ModClass) é enviada pelo LinkOwner, apenas os dois tratam a mensagem. Quando é enviada pelo LinkEnd, todos os LinkOwners nos LinkSlots do LinkEnd tratam a mensagem.

Enquanto o MPThread trata uma mensagem, ela está em uma ModIn. Existe a possibilidade desta ModIn sofrer uma destruição forçada. Neste caso, o MPThread deixará de existir por estar acessando esta ModIn. O MOS recriará o MPThread para que o sistema volte a tratar mensagens, mas a mensagem que estava sendo tratado possivelmente não seria mais tratada (depende da implementação do MOSLinkMP). Esta é uma medida para que o sistema se recupere, mas o sistema ainda corre o risco de travar porque o tratamento das mensagens é feito pelo programador da MOSLanguage, que pode cometer erros. Se o MPThread ficar travado, nenhuma outra mensagem será tratada no MOS, e o MOS não conseguirá se recuperar automaticamente.

Obviamente, a lógica de mensagens falharia se fosse possível enviar uma mensa-

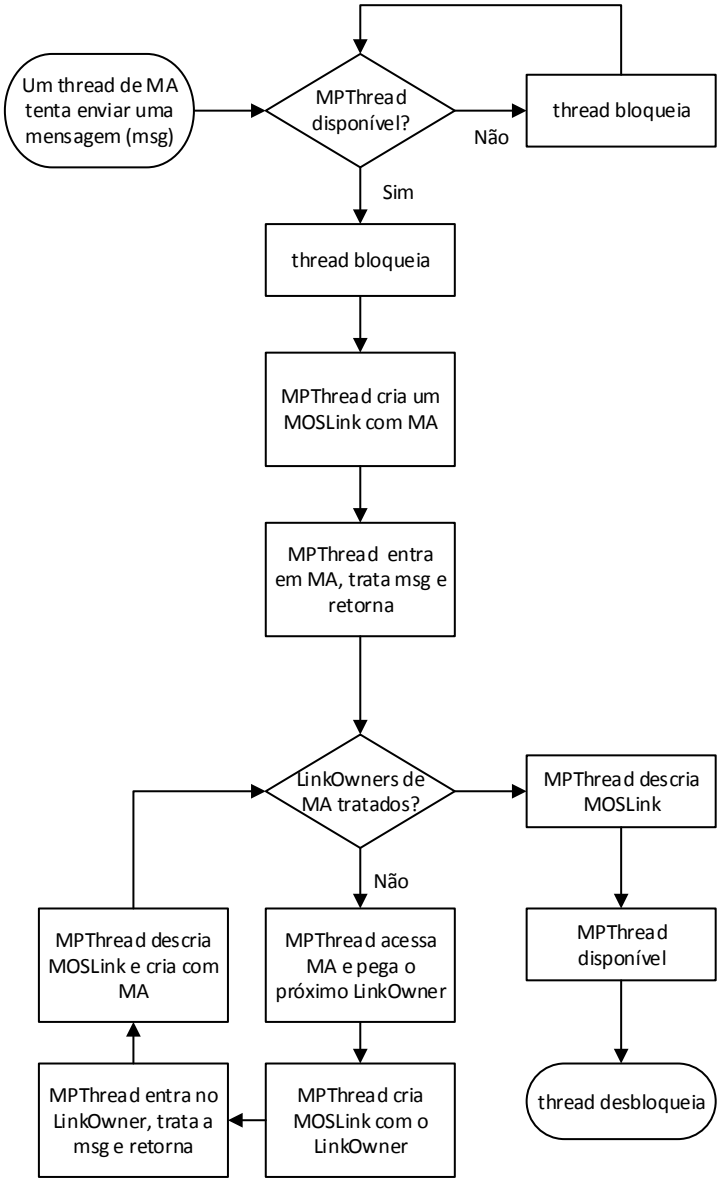

Figura 2.10: Procedimentos de MA enviando a mensagem.

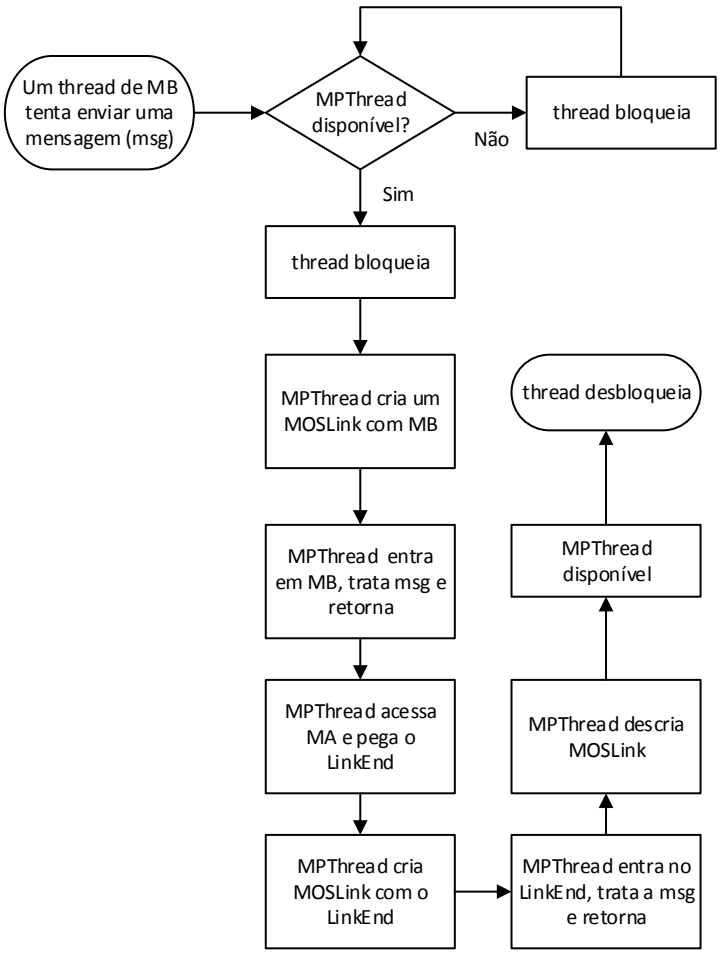

Figura 2.11: Procedimentos de MB enviando a mensagem.

gem dentro do tratamento de uma mensagem, portanto a MOSLanguage proíbe o envio de mensagens no tratamento de mensagens. O segundo envio poderia falhar e retornar imediatamente.

A implementação do tratamento de mensagens poderia ser feita de outra forma. Isto é verdade para muitas outras limitações que poderiam ser eliminadas do MOS e sua linguagem. No entanto, nesta dissertação não se propõe resolver todos os problemas do MOS, apenas especificar o sistema e em alguns casos apresentar a implementação.

2.11 Redes de Comunicação

O MOS tenta criar uma abstração para as redes de comunicação. O programador deve se preocupar com as mensagens que serão usadas ao invés dos pormenores da rede escolhida. O ideal seria o programador criar as mensagens sem sequer escolher a rede, mas infelizmente é impossível criar uma rede completamente genérica.

Uma análise de algumas redes de comunicação será feita. Considere as redes TCP/IP, RS232 e CAN focando nas mensagens do protocolo. A CAN é usada na análise mesmo não especificando a CAN no MOS. Na TCP/IP deve-se indicar quantos bytes de dados serão enviados. Na RS232 envia-se frames onde deve-se configurar o uso do parity bit e do stop bit. A CAN também possui um frame de dados em que deve-se indicar o número de bytes de dado, e um ID (identificador de 11 ou 29 bits). O tamanho máximo das mensagens também é variável nas redes. Na TCP/IP o pacote tem no máximo 65535 bytes, na RS232 cada frame tem até 9 bits de dado (mas normalmente usa-se 1 byte), e um frame de dados na CAN usa no máximo 8 bytes.

O foco da análise será alterado para o funcionamento das redes. Na TCP/IP uma conexão é criada entre dois pontos. É preciso indicar a porta, quem será o servidor e quem será o cliente da comunicação, e o cliente precisa do IP do servidor. A RS232 é uma rede serial que liga dois pontos, sem conexão e com várias opções de bitrate (taxa de bits enviados no tempo). A CAN é um barramento em que todos os dispositivos leem sempre e tentam escrever quando precisam. CAN permite comunicação multi-ponto sem conexão com várias outros bitrates.

Esta análise sequer leva em conta outras configurações relacionadas às outras camadas do Modelo OSI [6], as bibliotecas para usar estas redes ou o hardware. Qual a máscara IP que deve estar configurada no computador? Qual porta serial do computador (COM1, COM2 etc.) ou quais pinos/registradores do ATmega2560 serão usados? O que fazer quando o driver do dispositivo CAN não funciona?

Com redes tão diferentes como então facilitar o trabalho do programador? A abordagem adotada no MOS é separar as mensagens a serem enviadas/recebidas

das configurações de rede. Cada rede possui suas idiossincrasias, e cada dispositivo possui formas diferentes de tratá-los. No entanto, o que as redes possibilitam é a transferência de mensagens, e isto sim pode ser tratado de forma genérica. Ao invés do programador se preocupar como uma rede funciona, basta saber os limites das mensagens na rede.

2.11.1 Net

O MOS incorpora as configurações das redes dentro de sua implementação. Algumas opções serão disponibilizadas para cada MOSDeviceType, e então o programador pode criar uma rede de acordo com as opções. As redes tratadas no trabalho são as RS232 e a TCP/IP. O protocolo RS232 é a forma mais simples de comunicação entre um microcontrolador e um computador. O protocolo TCP/IP é o mais usado no mundo para conectar computadores (internet). Um tipo de rede pode ter opções diferentes em MOSDeviceTypes diferentes. Para criar uma rede RS232 no MOSDeviceType Windows é preciso indicar a porta serial (COM1, COM3, etc), no Linux o endereço do device, e no ATmega2560 o número da porta USART. Outras configurações como o bit de paridade estão em todos os MOSDeviceTypes. As opções na TCP/IP são iguais no Windows e Linux: cliente ou servidor, e se for cliente, deve-se indicar o IP do servidor.

Seguindo a lógica de nomenclatura no MOS usaremos o termo Net para designar uma rede disponível em um MOSDevice. A criação/configuração assim como a destruição de Nets é feita com funções do objeto System na MOSLanguage. Como as Nets dependem do MOSDeviceType, as funções só podem ser acessadas por BModClasses. Por exemplo, a função CreateTCPIP() é acessível no System do Windows e Linux com os mesmos argumentos. Já a CreateRS232() é acessível no System do Windows, Linux e ATmega2560, mas os três possuem argumentos diferentes (configurações diferentes). Outra função lista as Nets no MOS. Uma ModIn pode usar uma Net criada por outra ModIn com o item da lista.

A MOSLanguage não permite alocação dinâmica então, quando uma ModClass é definida, o número e tipo das Nets que serão usados no Module devem ser definidos. Isto é feito adicionando objetos privados nas ModClasses/BModClasses com classes que representam as Nets. O retorno das funções do System (criação, destruição ou item da lista de Nets) preenchem esses objetos. Os objetos na realidade representam o ID da Net a ser usada.

Informações são salvas dentro do MOS quando uma Net é criada. Cada rede de cada MOSDeviceType está associada a uma classe no C++ que contém estas informações. O MOS possui uma lista de objetos para cada classe. O ID de uma Net é o índice de uma das listas. A estrutura adotada nas redes é similar à estrutura

adotada no MOSLink. Assim como um MOS possui uma lista com os MOSLinks, o MOS possui uma lista para cada Net. Podemos dizer que os objetos privados dentro da ModClass que representam o ID da Net são os NetSlots (como os LinkSlots nos LinkEnd).

Uma observação importante deve ser feita em relação a este tipo de implementação (MOSLink e Net). A implementação do MOS para OS (MOSProcess) aloca dinamicamente objetos, portanto as listas têm tamanhos variáveis. Nos MOSMicros o tamanho é fixo. Isto significa que, ao ser compilado, o MOS deve ter todos estes valores máximos configurados. Nas Nets, deve-se por exemplo indicar o número máximo de redes RS232 que podem ser criadas, o tamanho do buffer da redes RS232 (assunto a ser visto mais a frente) e etc. O número de LinkSlots de um Module não é indicado na compilação do MOS, e sim na compilação do Module.

Um exemplo será criado para representar a etapa de criação de Nets. Um Module deve agir como um gateway entre as redes RS232 e a TCP/IP conectadas a um computador com Windows com um MOSProcess. A ModClass *Gateway* é criada com dois objetos privados *NetSlotRS232 serial* e *NetSlotTCPIP tcp*. A classe é então derivada na BModClass *Gatewaywin* vinculada ao Windows. A *Gatewaywin* é configurada para criar um thread assim que a ModIn for criada. No thread usa-se *CreateRS232()* e *CreateTCPIP()* com certa COM, porta e IP. Se as funções de criação falharem a ModClass se destrói, mas se funcionar os objetos *serial* e *tcp* são preenchidos. Isto resume a etapa de criação de Nets. Depois de criar e configurar as Nets o thread se destrói. Voltaremos ao exemplo após o funcionamento das mensagens ser descrito.

2.11.2 Protocol

Agora que foi-se visto como uma rede é criada no MOS, pode-se ver como uma rede é usada. O elemento chave é o Protocol definido na MOSLanguage. No Protocol o programador define um protocolo de mensagens. Os Protocols são definidos dentro de um MOSProject junto das ModClasses. Um protocolo consiste em uma sequência de campos (Field) cujo valor de cada campo pode determinar o próximo campo. Um Field é similar a uma variável de uma classe, e possui um tipo de dado ou classe.

A melhor forma de compreender o Protocol é através de figuras. A figura 2.12.a mostra uma única mensagem definida no Protocol. Esta mensagem poderia ser enviada ou recebida em uma Net. A figura 2.12.b modifica a 2.12.a. O valor do primeiro Field (booleano) determina quais serão os outros Fields, e com isto o Protocol define duas mensagens. As condições de um Field são o que determinam qual será o próximo Field. Uma condição representa a chamada de uma função do Field (Fields são como variáveis de uma classe) que retorna *true* ou *false*. A ordem que as condi-

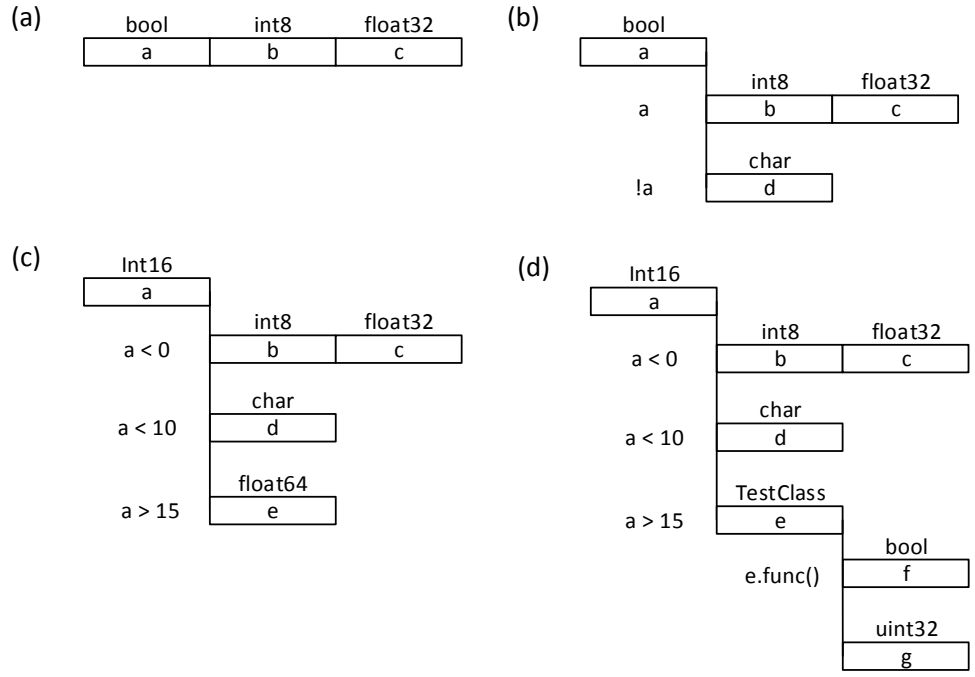

Figura 2.12: Exemplos de Protocol.

ções são feitas no Protocol importa, já que uma condição só é checada se a anterior falhar. A figura 2.12.c modifica 2.12.b mudando o tipo do primeiro Field para um inteiro. Agora 3 mensagens são definidas dependendo do valor do primeiro Field. Note que existem valores no primeiro Field que não constituem nenhuma mensagem (de 10 e 15), neste caso a mensagem pode ser descartada para o Protocol.

O itens da figura mencionados possuem condições feitas apenas no primeiro Field, mas qualquer Field pode ter condições que levam a outros Fields ou que descarte a mensagem. A figura 2.12.d mostra esse caso. O Field **e** pertence à classe *Test* que possui a função *f()*. Ao invés de usarmos uma condição do tipo maior, menor ou igual, usaremos a função definida na classe *Test*. Note que não é necessário definir uma condição (última mensagem formada na figura), e neste considera-se que a condição está satisfeita (*true*). Ou seja, Protocols possibilitam a construção ágil de protocolos elaborados de mensagens.

Todas as sequências de Fields formam mensagens. As mensagens devem ser nomeadas para identificá-las na MOSLanguage (figura 2.13), e os nomes devem ser únicos. Se não for definido não poderá ser usado pelos Modules. Não confundir com os nome dos Fields das mensagens, pois estes podem ser repetidos (Fields com mesmo nome possuem as mesmas características). Note que as condições são usadas para verificar se uma mensagem recebida é consistente com o Protocol. No entanto o envio de mensagens nada tem a ver com as condições. A mensagem a ser enviada tem os Fields preenchidos pelo programador. Os Fields a serem preenchidos ficam evidentes quando o nome da mensagem é escolhido. Na mesma figura um Field tem

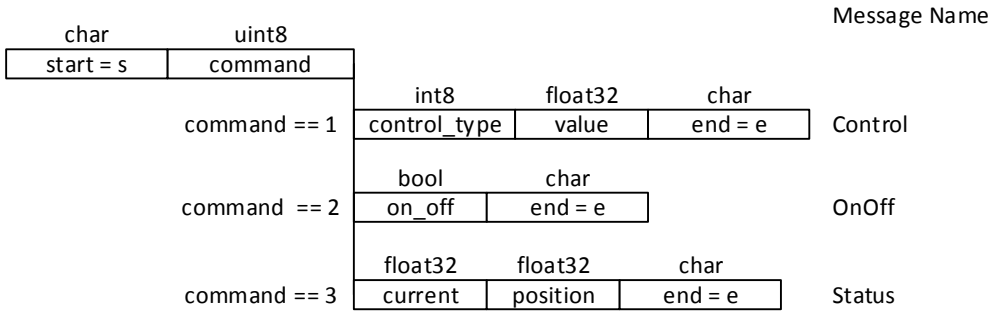

Figura 2.13: Exemplo de nome de mensagens no Protocol.

um valor constante. O programador não pode alterar o valor desse Field quando for enviar a mensagem. O valor constante também serve como uma condição para detectar a mensagem recebida.

Protocols seguem parcialmente o modelo de orientação a objetos. Os Fields podem ser vistos como suas variáveis. Protocols podem ser derivados como classes. Na derivação novos Fields podem ser adicionados e novas condições adicionadas ou definidas (condição não foi definida na Protocol base). No entanto, Protocols não definem funções na MOSLanguage. A figura 2.14 exemplifica a derivação.

Os Fields definidos em um Protocol são na realidade os dados de uma mensagem na Net. Um BProtocol é um Protocol vinculado a uma Net. Neste momento surgem outros Fields no Protocol relacionados à Net. Condições podem ser adicionadas nestes novos Fields. As redes RS232 e TCP/IP não são bons exemplos por não adicionarem Fields. A CAN por outro lado adicionaria o CAN ID da mensagem que é um elemento chave no CAN. Enquanto o Protocol permite a criação de um protocolo genérico, o BProtocol adapta o protocolo para uma rede. Isto ajuda no desenvolvimento de uma aplicação já que a comunicação pode ser planejada e testada de antemão, sem nenhum hardware estar disponível ou um hardware estar sendo usado temporariamente. Bastaria derivar o Protocol para BProtocol do hardware final quando este estiver disponível.

Um MOSProject define classes, ModClasses, BModClasses, Protocols e BProtocols. Todos são compilados pelo MOSBuilder. Quando uma classe possui um objeto de uma Net (como os usados no exemplo do gateway), ou seja, um NetSlot, o programador deve indicar o Protocol que será associado ao objeto. Um NetSlot de uma Net (RS232, TCP/IP etc.) pode se associar a qualquer Protocol, no entanto o NetSlot só pode se associar aos BProtocols da mesma Net. A partir da associação a classe pode definir uma função para cada mensagem definida no Protocol. São as funções que tratam as mensagem de Protocol quando os Net/Protocol as identificam. Ao mesmo tempo, a classe tem acesso às funções do NetSlot que, quando invocadas, enviam mensagens pela Net. Ou seja, a conexão de um NetSlot com um Protocol permite que a classe receba e envie mensagens de um Protocol.

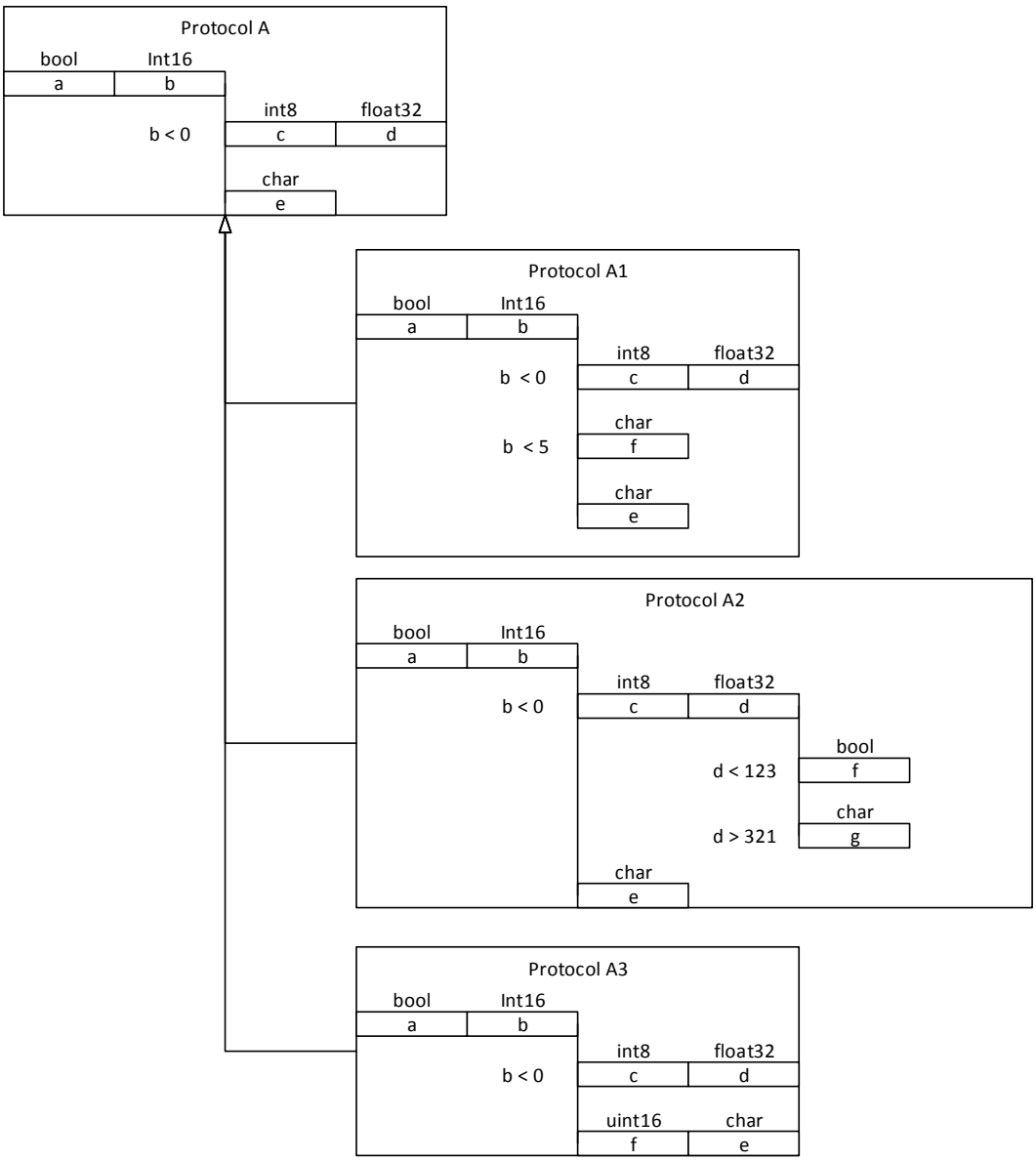

Figura 2.14: Exemplo de derivação no Protocol.

Existem duas formas de associar uma Net a um NetSlot (e, consequentemente, a um Protocol) na MOSLanguage: quando a rede é criada ou adicionando a uma Net já existente. Um exemplo na MOSLanguage da criação é *rs232 = System.CreateRS232()*, sendo *rs232* um *NetSlotRS232* de uma classe. Para adicionar um NetSlot em uma Net basta escrever no NetSlot o id da Net. Por exemplo, se três Nets de rede RS232 forem criadas no MOSDevice seus IDs serão 1, 2 e 3, então basta escrever *rs232 = 2* para que o NetSlot seja adicionado à segunda Net. Note que o operador igual não é uma simples escrita de valor no NetSlot (apesar de parecer para o programador da MOSLanguage), um procedimento complexo é feito dentro do MOS.

Múltiplos Protocols podem ser adicionados em uma Net. Isto significa que um mesmo canal de comunicação pode ser usado para enviar e receber mensagens completamente diferentes, e com destinos diferentes dentro do MOS (ModIns usando a mesma Net). Uma Net pode receber vários Protocols, mas o número máximo pode variar. Este máximo pode ser definido quando o MOS for compilado, definido quando a Net é criada ou indefinido. Isto dependerá da implementação do MOS e, portanto, não entraremos em detalhes. Pode parecer perigoso a princípio mas na realidade é apenas uma ferramenta para uso na aplicação e cabe ao programador usá-la.

A seguir um exemplo de aplicação será mostrado. Um MOSDevice deve acessar um driver de controle de motor por uma rede RS232 e, ao invés de concentrar toda a lógica em um único Module, foram criados dois, o *DriverManager* e o *DriverControl*. O *DriverManager* monitora o driver e controla quando ele deve ser ligado/desligado. O *DriverControl* realiza uma malha de controle, periodicamente recebendo dados dos motores e enviando sinais de controle. O Protocol *ProtDriver* foi criado para conter o protocolo de mensagem do driver, que equivale a um byte de início, um byte representando um comando e um byte de fim da mensagem. Os outros bytes da mensagem devem ficar entre os bytes de comando e fim da mensagem. Dois Protocols foram derivados, o Protocol *ProtDriverManager* e *ProtDriverControl* que serão usados, respectivamente, nos *DriverManager* e *DriverControl*. A figura 2.15 descreve os Protocols na aplicação. Algumas mensagens definidas serão usadas no envio e outras no recebimento. Note que não haverá conflito de mensagens a serem tratadas nos dois Protocols porque os comandos tratados são diferentes.

Mas como uma Net pode lidar com vários Protocols ao mesmo tempo? Algo muito importante para esta compreensão é o buffer usado no recebimento de mensagens na Net. Cada mensagem em um Protocol tem um tamanho, e o tamanho do Protocol representa o tamanho de sua maior mensagem. Um Protocol só pode ser adicionado em uma Net se o tamanho do Protocol for menor ou igual ao buffer da Net. Este é um problema na definição de Protocols que serão usados nos MOSMicros

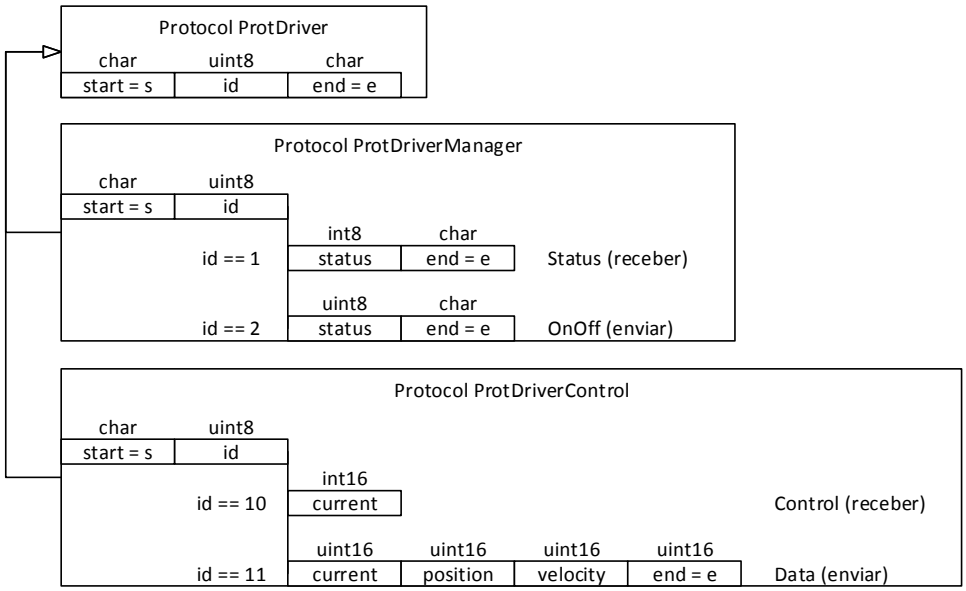

Figura 2.15: Exemplo do driver: Protocols.

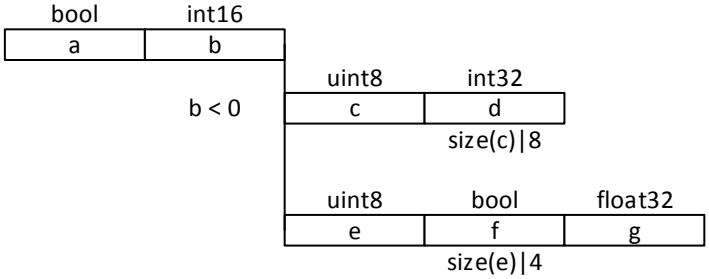

Figura 2.16: Exemplo de vetor no Protocol.

porque estes não possuem muita memória RAM. O tamanho do buffer é escolhido de forma diferente em MOSDeviceTypes diferentes. Os MOSProcess podem usar alocação dinâmica internamente, portanto pode ser determinado pelo programador na criação das Nets. OS MOSMicros por outro lado tem um número limitado de redes de cada tipo, que é determinado na compilação do MOS. O buffer neste caso faz parte do objeto relacionado à Net em C++ que por sua vez pertence ao objeto global do MOS (System na MOSLanguage).

Algo que não foi comentado antes é o uso de vetores de dados no Protocol. Isto se traduz como um Field do Protocol sendo um vetor com tamanho definido ou tamanho máximo definido. A figura 2.16 mostra um Protocol em que uma mensagem contém um vetor de *int* de 8 bytes, e outra mensagem de *bool* com até 4 bytes. Quando o tamanho é variável, um Field anterior é usado para indicar o tamanho do vetor. Saber o tamanho máximo do vetor permite determinar o tamanho do Protocol e com isto se ele poderá ser usado ou não em um Net.

O funcionamento do envio de mensagens será analisado. O Protocol define nomes para as mensagens. O NetSlot (que está vinculado a um Protocol) possui uma função de envio para cada mensagem. Por exemplo, se a mensagem ControlMsg for definida no Protocol de um NetSlot (*NetSlotRS232 rs232*), o programador pode chamar *rs232.SendControlMsg(...)* com os argumentos adequados (valores dos Fields da mensagem). As funções de envio bloqueiam até a mensagem ser enviada. Os NetSlots estão associados a uma Net, então a função irá travar enquanto alguma outra mensagem estiver sendo enviada pela Net. Não importa de qual ModIn, NetSlot/Protocol ou thread está tentando enviar uma mensagem por uma Net, a função trava se alguma outra mensagem já está sendo enviada pela Net porque a Net usa um objeto de sincronização.

Ao enviar uma mensagem pela Net não há restrição em relação ao conteúdo da mensagem, mas no recebimento há. Esta é forma que o MOS adotou para descobrir se uma mensagem se encaixa ou não em um dos Protocols, e usa as condições do Protocol para identificar as mensagens. A comparação é feita em um thread especial criado junto da Net, o NetThread. O thread não pertence a nenhuma ModIn assim como o MPThread. Após o thread ser criado ele dorme até que um byte ou conjunto de bytes cheguem. A implementação do NetThread deve variar muito já que cada Net em cada MOSDeviceType deve receber dados de forma diferente. Os MOSProcess usariam bibliotecas e armazenariam os novos bytes no buffer. Os MOSMicros devem trabalhar com interrupções. Quando a interrupção ocorre o novo byte é armazenado no buffer e libera o NetThread se este estiver travado.

O uso de um thread para o tratamento de mensagens é uma vantagem. Quem escolhe qual thread irá executar é o Escalonador. Se um thread tem uma prioridade maior do que o tratamento de mensagens ele será executado antes do tratamento. Por exemplo, se o MOSDevice é um driver de potência de um motor e roda um algoritmo de controle em tempo real, a execução do algoritmo (que envia sinais para o motor) tem prioridade sobre a comunicação no MOS. Enquanto uma interrupção está sendo executada no MOSMicro os outros threads não executam nem podem parar a interrupção (não importa a prioridade dos threads). A vantagem do uso de threads no tratamento é maior nos MOSMicros porque se o tratamento fosse dentro de uma interrupção e o tratamento demorasse, o deadline de threads não seria atendido. Na implementação do MOS deve-se tomar cuidado para que as interrupções sejam sempre rápidas para evitar estes tipos de problemas.

Uma observação deve ser feita sobre a criação de uma Net. Toda Net usa objetos de sincronização, ao menos um thread e outros itens (ex: RS232 usa uma das USARTs). Se algum destes objetos não estiver disponível a Net não pode ser criada. Novamente, isto é um problema nos MOSMicros pelo fato dos objetos de sincronização/threads serem definidos na compilação do MOS para um MOSDevice e também

por serem em pouca quantidade (RAM pequena).

Dependendo da implementação um byte ou vários bytes podem ser recebidos de uma vez, mas o thread trata cada byte individualmente. Quando um byte é recebido o thread cria um MOSLinkDirect com cada ModIn (Protocols pertencem às ModIns) em sequência. Quando o thread entra na ModIn, uma função é chamada para verificar se uma mensagem foi identificada. Se não for identificada o thread retorna da ModIn e repete o procedimento até identificar a mensagem de algum Protocol. Quando identificada, a mensagem é tratada dentro da própria ModIn e o buffer é esvaziado. O thread retorna e volta a esperar o próximo byte, que será o primeiro byte de uma nova mensagem. Isto implica em uma mensagem ser tratada em apenas um Protocol. Este comportamento foi escolhido mas pode mudar se as aplicações dos MOS exigirem.

A figura 2.17 resume os procedimentos relacionados a uma Net.

A figura 2.18: ilustra as etapas para a identificação de uma mensagem dentro de um Protocol. Os bytes são verificados um a um. Quando os bytes formam um Field e o Field possui condições, verifica-se as condições. Na primeira condição satisfeita escolhe-se o ramo do Protocol relacionado à condição para tratar os próximos bytes. Se nenhuma condição for satisfeita então os bytes não constituem uma mensagem do Protocol, então um novo byte só será testado no Protocol quando o buffer for reiniciado. Se o novo byte terminar o último Field de uma mensagem, a mensagem será identificada e pode ser tratada. A figura 2.18 não considera que outro Protocol pode identificar a mensagem e esvaziar o buffer. Nesta caso quando o novo byte chegar a identificação voltará a ser feita a partir do primeiro byte do Protocol.

Para finalizar as explicações do Protocol falta indicar o que pode ser feito quando um Protocol é associado a um NetSlot. O programador pode redefinir as condições no Protocol. Diferente da derivação de Protocols, é possível invocar funções ou usar variáveis da ModClass. A figura 2.19 exemplifica esta habilidade. O programador também pode definir uma função de tratamento na MOSLanguage para cada mensagem. A vantagem deste poder é usar os Fields da mensagem como variáveis. Voltemos ao exemplo do driver de controle. O código 2.5 mostra como seria DriverManager e explica como o ProtDriverManager trata a mensagem que recebe. A *StatusMessage(char status)* é a função executada quando a mensagem de status é recebida. Podemos ver que a variável *status* é usada como se fosse uma variável. No tratamento a função *StopControl()* é chamada, que por sua vez envia a mensagem *DriverOffMessage*. O envio da mensagem *DriverOffMessage* não precisa ser definido pelo programador porque o MOSBuilder cuida disto. Ou seja, a MOSLanguage facilita o envio, recebimento e tratamento de mensagens em protocolos diferentes.

Código 2.5: Pseudo-código da ModClass DriverManager do exemplo de Driver

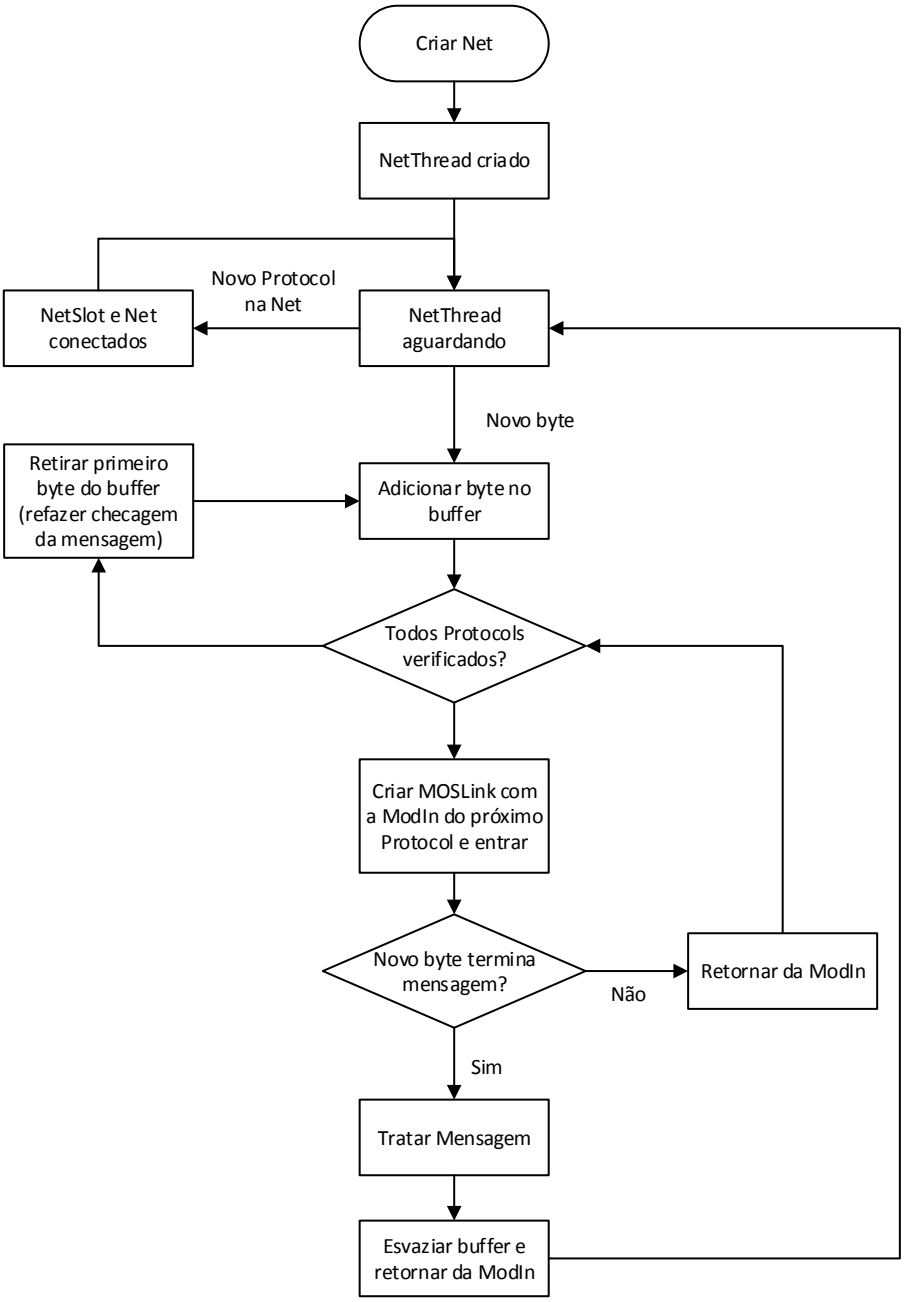

Figura 2.17: Etapas na Net.

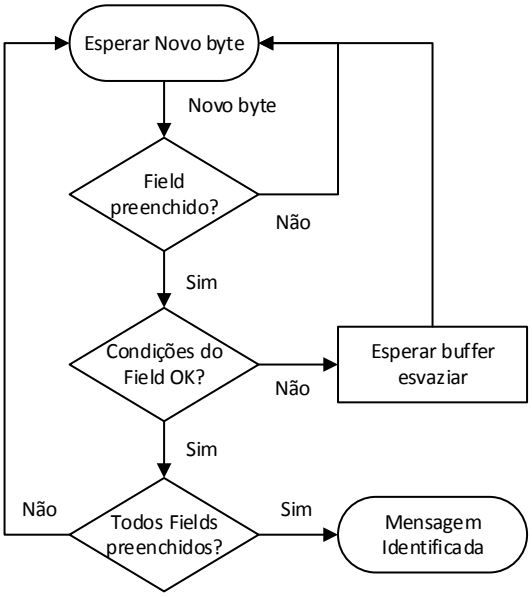

Figura 2.18: Etapas na identificação de mensagens.

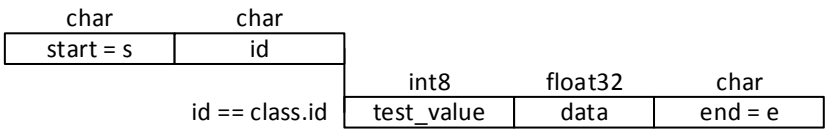

Figura 2.19: Exemplo de Protocol em uma classe.

```
ModClass DriverManager {
  NetSlotRS232 rs232(ProtDriverManager);

  char last_status;
  bool erro;

public:
  [] StartControl();
  [] StopControl();
}

[] DriverManager::rs232::Recv::StatusMessage(char status) {
  if (status == ERRO)
  {
    bool erro = true;
    StopControl();
  }
  last_status = status;
}

[] DriverManager::StopControl() {
  rs232::Send::DriverOffMessage();
}
```

Com o Protocol bem definido, voltemos ao exemplo do gateway. Os NetSlots já foram configurados no thread da ModIn do *gateway*, ou seja, as Nets RS232 e TCP/IP já estão com um Protocol do *gateway*. Basta um Protocol para os dois NetSlots e que será definido agora. Considere que os dispositivos conectados no MOSDevice usam o seguinte protocolo: 1 byte de início, 1 byte indicando o número de bytes do vetor de dados, o vetor de dados e 1 byte de fim. A figura 2.20 mostra o Protocol que segue o protocolo especificado. O gateway ignorará mensagens fora do protocolo. O Código 2.6 em pseudo-código mostra como o tratamento das mensagens é feito em cada NetSlot. Basicamente, no tratamento da mensagem de uma Net envia-se a mesma mensagem pela outra Net, criando um Gateway.

Código 2.6: Código para definir o tratamento de mensagens no exemplo do Gateway

```
Gateway::serial::TransportMessage(char vsize, char[] v)
{
  tcp.TransportMessage(vsize, v);
}
```

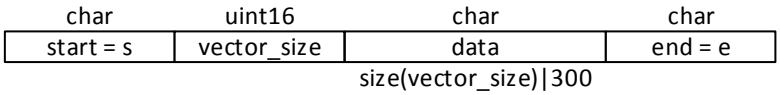

Figura 2.20: Protocol do exemplo de Gateway.

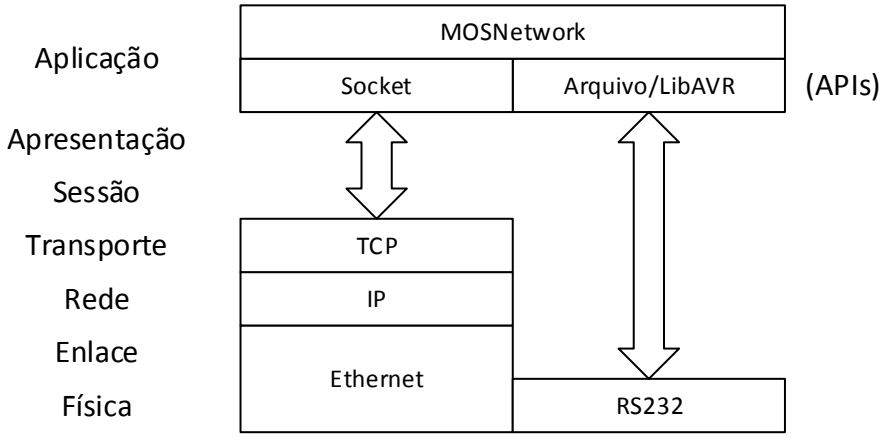

Figura 2.21: Modelo OSI da MOSNetwork.

```
Gateway::tcp::TransportMessage(char vsize, char[] v)
{
  serial.TransportMessage(vsize, v);
}
```

2.11.3 MOSNetwork

A MOSNetwork é uma rede de comunicação definida pelo MOS. Seguindo o Modelo OSI, a MOSNetwork faz parte da Camada de Aplicação. Nas camadas abaixo estão os protocolos de comunicação usados dentro do MOS como a RS232 e a TCP/IP (figura 2.21). A rede têm múltiplos objetivos dentro do MOS. Várias funções do System usam essa rede. A MOSNetwork é uma Net assim como a RS232 e a TCP/IP dentro do MOS e da MOSLanguage. Isto significa que as ModClasses podem conter NetSlots da MOSNetwork e o System possui uma função para criar uma conexão na MOSLanguage.

O foco mudará para como a MOSNetwork é montada dentro de um MOSDevice. Existe um Protocol especial, escrito em C++, que pode ser inserido em cima de outras Nets. Este é o MOSNetProtocol. Existem funções do System que o insere nas Nets, mas o programador não lida com este protocolo (basta indicar a

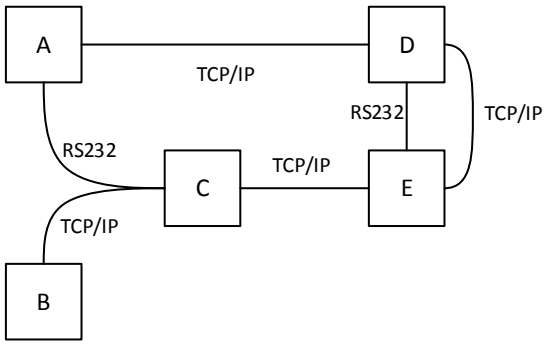

Figura 2.22: Exemplo de conexões com MOSDevices.

Net). Por exemplo, no ATmega2560 o System possui a função *AddMOSNetworkOnRS232(NetSlotRS232 slot)*. Se esta função for chamada com um objeto da *NetSlotRS232* configurado, o protocolo em C++ será inserido na Net RS232 associada. A rede MOSNetwork é única dentro do MOSDevice, ou seja, os Protocols executando nestas Nets acessam os mesmos dados.

A MOSNetwork só pode ser inserida em outras Nets que realizam comunicação ponto a ponto. A MOSNetwork é especificada para ligar dois MOSDevices. Ou seja, se o protocolo CAN for implementado no MOS, a MOSNetwork não pode ser montada sobre a CAN por ser um barramento, ao menos não diretamente.

A MOSNetwork e outras Nets podem ser definidas na compilação do MOS para que o MOSDevice crie as Nets quando iniciado. Por exemplo, deseja-se carregar o MOS em um microcontrolador (ATmega2560) e assim que este comece a executar, o MOS deve se conectar com o PC através de uma rede RS232 (USART2). O microcontrolador ficará isolado quando receber o MOS padrão, então, antes de enviar o MOS para o micro deve-se recompilá-lo adicionando algumas instruções iniciais. Primeiro indica-se que uma nova Net RS232 deve ser criada usando a USART2, e depois que a MOSNetwork deve ser montada em cima desta rede. Desta forma será possível desenvolver ou testar a aplicação imediatamente, já que o microcontrolador se comunicará imediatamente com o MOSProcess no computador.

O objetivo da MOSNetwork é conectar MOSDevices em uma rede sem topologia definida. Seria possível criar conexões como as na figura 2.22. A MOSNetwork é semelhante à Rede Mesh (topologia) [46], embora esta última esteja muito ligado ao padrão 802.11 (Wi-Fi). A MOSNetwork permite que uma comunicação seja repassada por redes diferentes até chegar ao destino, que é um dos MOSDevices. A mesma figura mostra os MOSDevices conectados mesmo com conexões de Nets diferentes.

O funcionamento do MOSNetProtocol é similar aos Protocols. O tratamento é

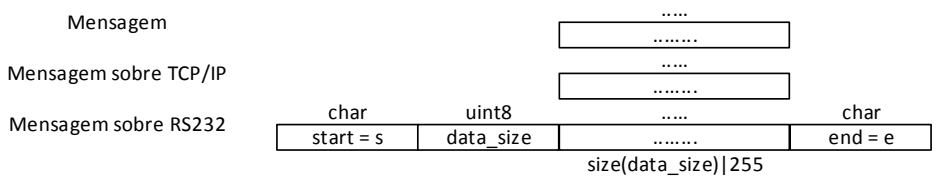

Figura 2.23: MOSNetProtocol sobre outras Nets.

feito no NetThread da Net que está executando o MOSNetProtocol. Isto ainda é feito byte a byte. A diferença é que, quando os Fields são recebidos, não ocorre uma simples verificação de condições. O NetThread pode executar uma função do System, enviar uma mensagem por qualquer Net, ou até mesmo entrar em ModIns nos próximos bytes para continuar o tratamento da mensagem. A MOSNetwork não precisa de um NetThread próprio pois pode usar o dos outros.

O MOSNetProtocol é adaptado para o uso adequado em cada Net. Por exemplo, em redes RS232 é comum o uso de bytes para indicar o início, o fim, e o número de bytes de uma mensagem. Na TCP/IP isto é desnecessário porque isto já é feito no protocolo TCP. Então mensagens da MOSNetwork na RS232 contém 3 bytes a mais, enquanto na TCP/IP contém nenhum byte extra (Figura 2.23).

Como os Modules usam a MOSNetwork? Esta Net permite a criação de uma conexão entre dois MOSDevices. Isto envolve o MOSMaster e o roteamento de mensagens, mas isto será visto mais tarde. O modelo adotado para a conexão é o de cliente/servidor (ou mestre/escravo). O servidor irá aguardar um pedido de conexão por um tempo. Se um cliente fizer o pedido neste período a conexão entre os dois é estabelecida. Com a conexão estabelecida, os dois podem trocar mensagens livremente (não há mais diferença entre cliente e servidor). Todo dispositivo na MOSNetwork possui um identificador (ID) único escolhido pelo MOSMaster. O cliente precisa indicar o ID com que deseja encontrar o servidor. A conexão utiliza um número (porta) para que não ocorrem conexões inesperadas. Os mecanismos adotados são similares ao IP e porta na criação de uma conexão TCP/IP.

O programador pode invocar duas funções do System na MOSLanguage, a *CreateMOSNetClient(...)* e *CreateMOSNetServer(...)* para iniciar a conexão na MOSNetwork como cliente ou servidor. A função do servidor suspende por certo tempo ou até um cliente tentar se conectar pela porta. A função do cliente envia uma mensagem pela rede que trafega pelos dispositivos da MOSNetwork até chegar no MOSDevice correto e então verificar se existe alguma conexão esperando para ser criada com a porta.

O retorno destas funções é do tipo *NetSlotMOSNetwork*. ModClasses podem ter objetos do tipo *NetSlotMOSNetwork* que devem ser associados a Protocols. Uma

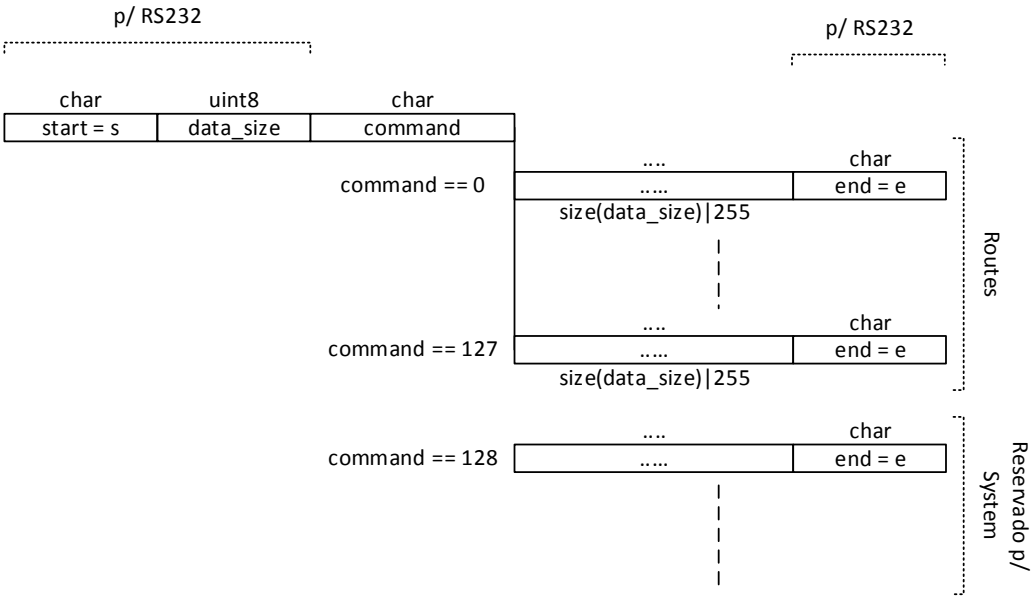

Figura 2.24: MOSNetProtocol sobre RS232.

função do System lista todas as *NetSlotMOSNetwork* que foram criadas no MOSDevice, então qualquer ModIn pode se ligar a qualquer conexão da MOSNetwork feita no MOSDevice. Ou seja, para o programador, o uso da conexão é idêntico ao das outras Nets.

Existe outra forma das ModClasses usarem a MOSNetwork. Diversas funções do System enviam mensagens pela MOSNetwork como as *CreateMOSNetClient()* e *CreateMOSNetServer()*. O uso não é transparente para o programador. As vezes uma função pode enviar várias mensagens pela MOSNetwork e pode aguardar a chegada de outras. O MOSProtocol foi mencionado mas não foi explicado. Não é possível descrever o MOSProtocol com figuras iguais aos dos Protocols porque o MOSProtocol é feito em C++ ao invés da MOSLanguage. A mensagem a ser enviada na MOSLanguage possui ao menos um byte, o Command. Os valores de 0 a 127 são usados para identificar rotas de mensagens (Routes), e os valores de 128 a 255 são reservados para o próprio MOS. As funções do System usam os dois tipos, mas as mensagens entre MOSDevices (*NetSlotMOSNetwork*) usa os identificadores de Routes. Depois do primeiro byte estão os bytes da mensagem a ser enviada. A figura 2.24 mostra como a mensagem de um Protocol na MOSNetwork é enviado quando sai do MOSDevice pela RS232. Os bytes start, size e end são usados para garantir a continuidade dos dados na RS232, e não existiriam se a mensagem fosse enviada pela rede TCP/IP. O Command possui um identificador de Route que está relacionado à conexão entre dois MOSDevices. Os outros bytes são os que de fato contém a mensagem definida no Protocol, e que serão tratado na ModIn do outro MOSDevice.

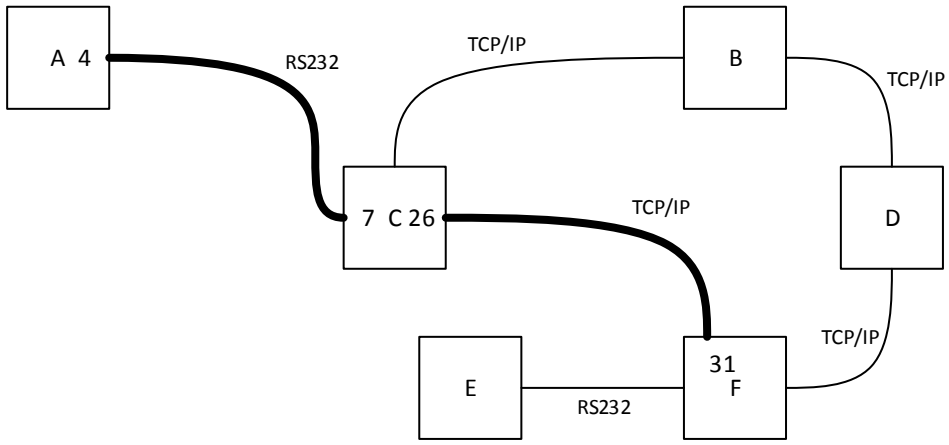

Figura 2.25: Exemplo de Routes do MOSNetProtocol.

Uma Route na MOSNetwork é um caminho que mensagens fazem para ir de um dispositivo ao outro na MOSNetwork. A figura 2.25 contém um exemplo de uma Route feita entre os MOSDevices **A** e **F** passando por **C**. Repare nos números usados na Rotue. A Route será usada para conectar duas ModIns, uma em cada MOSDevice. Quando **A** enviar a mensagem para **F** irá inserir o valor 7 no Command. A ModIn envia uma mensagem pelo seu *NetSlotMOSNetwork*, que então é repassado para a Net RS232. Quando **C** receber a mensagem e verificar o Command saberá que deve retransmitir a mensagem por outra Net mas mudando o valor 7 para 31. Quando **F** receber a mensagem com valor 31 sabe que deve entrar na ModIn do **F** para chamar a função de Tratamento da mensagem. Note que o caminho contrário não usa os mesmos valores de Route no Command. Cada MOSDevice possui uma tabela virtual de roteamento para indicar o que deve ser feito quando a mensagem chega. A mensagem ou é retransmitida substituindo o Command, ou é tratada por uma *NetSlotMOSNetwork* (nos Protocols dos Modules) ou é tratada pelo próprio System. A tabela é preenchida de acordo com a necessidade.

Os valores reservados do Command estão relacionados ao mestre da rede MOSNetwork, o MOSMaster. Existe apenas um MOSMaster na MOSNetwork. Ele é responsável por agrupar informações como na lista abaixo. Isto permite que qualquer MosDevice encontre informações de outros MOSDevices sem se comunicar com todos eles.

- Nome (string) e MOSDeviceType (Windows, Linux e ATmega2560) de todos MOSDevices na MOSNetwork
- Nome de todos MOSProjects e o ID dos MOSDevices que contêm o MOSProjects

- Conexões que ligam os MOSDevices (ex: RS232 liga os MOSDevices A e B)
- Todas as Routes na MOSNetwork (conhece o número de Route em cada MOSDevice)

O MOSMaster conhece todas as conexões da MOSNetwork. É ele quem determina o ID único do MOSDevice quando entra na MOSNetwork. Quando alguma conexão do *NetSlotMOSNetwork* for criada, é o MOSMaster que determina por quais MOSDevices a Route irá passar e se comunica com os mesmos para cadastrar os números de Route.

Devido à grande carga de informações, o MOSMaster deveria ser sempre um MOSProcess, mas isto só pode ser decidido quando o MOS for implementado. Por exemplo, a EEPROM do ATmega2560 pode ser o suficiente para guardar todas essas informações. Não é o foco do trabalho discutir como o MOSMaster é escolhido entre os MOSDevices conectados ou o que acontece quando o MOSMaster cai da rede porque estes aspectos também devem ser analisados na implementação do MOS. No entanto, iremos descrever o procedimento para que um MOSDevice encontre o MOSMaster pela primeira vez.

Os Commands reservados estão relacionados à comunicação entre dois MOSDevices. Vejamos novamente o caso da MOSNetwork sobre a RS232. Quando *AddMOSNetworkOnRS232(NetSlotRS232 slot)* é chamado no MOSDevice **A** (está fora de uma MOSNetwork) uma mensagem é enviada pela rede RS232 para o MOSDevice **B**. Se não houver resposta dentro de um período de tempo o MOSProtocol continua na Net RS232 mas nenhum cadastro é feito com o MOSMaster. Não há reposta se a Net RS232 no **B** não tem o MOSNetProtocol. Com o MOSNetProtocol a resposta depende se o **B** está dentro de uma MOSNetwork. Se não estiver o **B** encontra-se na mesma situação do **A**, e talvez um dos dois passa ser o novo MOSMaster na rede de dois dispositivos (os dois podem trocar mensagens até decidir quem é o novo MOSMaster). Se **B** já está conectado a um MOSMaster ele pedirá uma nova Route para o MOSMaster e adicionará o Module **A** no final da Route. Com isto, **A** pode se comunicar com o MOSMaster. Então **A** envia uma série de mensagens para o MOSMaster para cadastrar suas informações. Depois disto **A** faz parte da MOSNetwork. Este procedimento é descrito na figura 2.26.

Ou seja, quando uma MOSNetwork é aplicada com sucesso em cima de outra Net do MOSDevice, uma Route é criada entre o MOSMaster e o MOSDevice. Isto significa que pode existir mais de um caminho entre o MOSDevice e o MOSMaster tornando a rede mais robusta. A Route com o MOSMaster é usado em várias funções do System, sendo que podem usar a Route diversas vezes. A lista abaixo representa alguma das funcionalidades alcançadas quando uma Route é usada.

- Pedido de uma nova Route com outro MOSDevice.

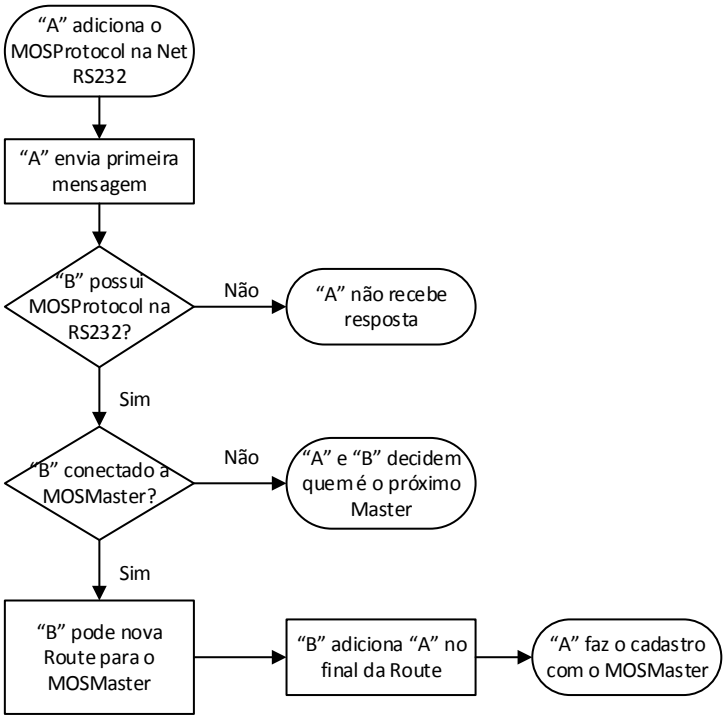

Figura 2.26: Primeira comunicação na MOSNetwork.

- Gerar um CMod de um MOSProject para um MOSDeviceType.
- Enviar um CMod de um MOSProject para um MOSDevice (transformar em RMod).
- Instanciar uma ModIn de um RMod em um MOSDevice.
- Destruir uma ModIn em um MOSDevice.
- Enviar lista com todos as ModIns executando (com id e nome da ModIn) em um MOSDevice para outro MOSDevice.

Nos MOSMicros o número de Nets em que a MOSNetwork pode ser adicionada é fixo e determinado na compilação do MOS. O mesmo vale para o número de Routes e conexões com outros MOSDevices. Apesar do protocolo definir um número máximo de Routes, os dispositivos podem ter muito menos do que 127 Routes porque as Routes representam um consumo razoável de memória.

2.12 Aplicações

Com as principais características do MOS descritas, é possível fazer um filtro de onde o MOS pode ser aplicado.

As principais habilidades do MOS são a de conectar dispositivos MOS na rede MOSNetwork, de permitir que "programas" (Modules) sejam transferidos entre estes, e de permitir a comunicação entre os "programas". A princípio, o MOS atenderia? em qualquer aplicação que lide com múltiplos dispositivos.

O MOS poderia ser implementado em múltiplos tipos de dispositivos. Inclusive, o MOS permitiria que dispositivos de tipos diferentes se conectassem sem se importar com a topologia. Um grande diferencial surge com a implementação do MOS em microcontroladores, já que as soluções semelhantes não têm essa possibilidade.

Ao ser aplicável em microcontroladores o número de aplicações aumenta, podendo ser usado até mesmo em aplicações de um único dispositivo, sendo um OS para microcontroladores, e que utiliza MOSLanguage (talvez facilite a programação do dispositivo). Por exemplo, um driver para motor que deve se comunicar pela rede RS232 pode ser construído inteiramente com o MOS. Alternativamente, o driver poderia usar a RS232 para entrar na MOSNetwork e permitir que o driver seja acessado remotamente.

A implementação em microcontroladores permite que o MOS seja usado em sistemas embarcados, sistemas cuja manutenção é complexa. Por exemplo, robôs podem conter vários microcontroladores soldados em placas, tornando a alteração do software destes complicada. Se estes estivessem conectados na MOSNetwork, basta um ponto de entrada no sistema (ponto onde outro MOSDevice pode ser conectado) para acessar todos os microcontroladores remotamente.

Obviamente, sistemas embarcados poderiam utilizar processadores x86, x64 ou ARM com vários sistemas operacionais. Esta dissertação foca no desenvolvimento do MOS no Windows e no Linux em x86, logo, a recompilação do MOS para estes OSs em ARM (Windows 8 RT e Linux ARM) ou 64 bits não é problemática. A dificuldade está em adaptar o MOSBuilder para ARM, já que o conjunto de instruções de máquina do x86 e do ARM são diferentes.

A capacidade de comunicação do MOS o torna vantajoso na Automação Residencial. Uma casa inteligente possui dispositivos/sensores conectados por redes (com ou sem fiação). A casa pode ter sistemas automáticos que decidem como dispositivos se comportam. Os usuários também podem controlar os dispositivos remotamente. É comum existir um computador para centralizar informações, tomada de decisões e interação com usuários. Este trabalho trata apenas das redes TCP/IP e RS232, que são ponto a ponto, e isto já seria o suficiente para torná-lo aceitável para a aplicação. Redes que utilizam barramento (RS485, CAN) podem ser adicionados aos MOSDevices para diminuir a quantidade de fios. Gateways poderiam ser adicionados nos dispositivos inteligentes (microcontrolador conectado ao Gateway) em outro casos. Estivemos considerando o uso do MOS apenas nos dispositivos inteligentes, mas MOSNetwork também poderia ser usada para ligá-los ao computador central.

O uso do MOS em Redes Industriais é mais complicada do que nas outras aplicações devido à padronização/especificação destas redes. Redes Industriais referem-se a protocolos de comunicação industriais como FOUNDATION Fieldbus, Modbus, DeviceNet, etc. Nos outros a aplicação poderia começar do zero adotando a filosofia do MOS, mas adaptar uma planta para usar o MOS é muito complexo. A solução mais prática seria integrar os protocolos industriais nas redes do MOS (neste trabalho foram abordadas somente a TCP/IP e RS232) e criar novos dispositivos MOS que usem estes protocolos. A vantagem está na redução do tempo de desenvolvimento destes dispositivos. Uma solução audaciosa seria criar uma nova Rede Industrial que adotasse a MOSNetwork para a comunicação.

Aplicações em tempo real são outro enfoque do MOS. O MOSProcess trabalha em cima do Kernel de OSs, e não é capaz de modificá-los. Tipicamente, o tempo mínimo na troca de processos é de 15ms, ou seja, o Linux e o Windows não estão preparados para aplicações tipo hard real-time. No entanto, o MOSMicro não tem um Kernel fixo. A ideia é cada MOSDeviceType ter um conjunto de algoritmos de escalonamento a disposição do programador permitindo, inclusive, que o algoritmo mude enquanto o MOSDevice está funcionando. Algoritmos em tempo real entram neste conjunto.

Uma das vantagens do MOS é ele ser implementado em C++ e usar recursos padronizados dos sistemas operacionais, portanto a adaptação do MOS para outros OSs não é complicado. Não é objetivo desta dissertação tratar da implementação do MOS em todos os OSs, mas o MOS poderia ser implementado em cima de Sistema Operacional em Tempo Real. Existem alguns OSs em tempo real baseados em Unix. Como o Linux é baseado em Unix, a implementação do MOS nestes OSs torna-se trivial.

O MOS também se encaixa bem em supercomputadores, onde cada supercomputador teria ao menos um MOSDevice. Parallella é um projeto fundado no Kickstarter em 2012. O objetivo do projeto é produzir supercomputadores portáteis que consistem de um chip de vários núcleos (16 ou 64 núcleos) e um processador (rodando Ubuntu) capaz de se comunicar com este chip. Cada núcleo possui uma CPU RISC e uma memória própria (e pequena). A arquitetura permite que cada núcleo acesse a memória do outro (não há proteção de memória entre os núcleos). As ferramentas de desenvolvimento do Parallella consistem em enviar um pequeno programa (escrito em C++ e compilado) para cada núcleo.

A plataforma Parallella possui um design diferente, mas que é ótima para a computação paralela. O MOS se encaixa muito bem nesta aplicação. Os núcleos na Parallella são muito parecidos com os microcontroladores AVR que a dissertação trata (implementação do MOS), mas com frequência e memória um pouco maiores. O MOS poderia ser implementado para rodar nestes núcleos. A MOSNetwork

pode ser montada com cada núcleo sendo um MOSDevice, mais um processo no processador rodando Ubuntu. Nesta implementação o MOS não usaria a memória compartilhada entre os núcleos diretamente, e sim para criar um protocolo de comunicação.

Como foi comentando anteriormente, o MOS possui algumas limitações, o que acaba por restringir onde o MOS pode ser aplicado. Devido à memoria desprotegida dentro do MOS, este só deve ser usado em aplicações confiáveis, onde não está prevista a entrada de intrusos no sistema.

As limitações estão concentradas no MOS em microcontroladores. Primeiramente, o MOS é desenvolvido em C++, logo, só pode ser implementado em microcontroladores com compilador para C++. Microcontroladores possuem uma memória muito pequena para os padrões atuais e é dividida para os threads, então aplicações que lidam com muitos dados não devem usar MOS. O número de threads e o tamanho de suas Pilhas são pequenos. O restante da memória é usado dentro do núcleo do MOS. Com isto, o número máximo de objetos de sincronização, threads, CMod, RMod e etc., que o MOSDevice terá devem ser configurados antes da compilação do MOS, e devem ser pequenos.

A MOSLanguage possui dois fatores limitantes para o programador. A primeiro é a falta de ponteiros para que o programador não tenha acesso direto à memória. A segunda é falta de alocação dinâmica. Este são os principais pontos negativos que atingem os MOSProcesses. Isto impede que bibliotecas de outras linguagens sejam refeitas na MOSLanguage, tornando-se MOSProjects e Modules. Ponteiros são ferramentas poderosas mas o uso vai contra a filosofia de programação do MOS. No entanto, existe uma solução para o segundo problema. A MOSLanguage deve ser modificada para que as BModClasses de MOSDeviceType Windows, Linux e etc., possam usar alocação dinâmica. Nos MOSProcesses, algo como o procedimento que threads fazem (acesso/entrada seguros em Modules) teriam de ser implementado no acesso de variáveis alocadas dinamicamente. Na MOSLanguage o usuário não lidaria com ponteiros na alocação dinâmica, mas o que fosse alocado seria acessado de forma segura, como na linguagem Java.

Capítulo 3

MOSLanguage

3.1 Necessidade da MOSLanguage

O MOS poderia ser um Sitema Operacional/Framework que utilizasse linguagens de programação conhecidas. Entre elas estão C++, Java, Phyton, Ruby, C# etc. No entanto o autor optou por definir uma nova linguagem de programação MOS por acreditar que as linguagens mencionadas não permitiram que as funcionalidades e restrições do MOS fossem reproduzidas. Abaixo listamos algumas das características desejadas para a linguagem de programação ligada ao MOS:

Compilável A linguagem não pode ser interpretada porque microcontroladores não tem muita potência computacional.

Orientação a Objeto O paradigma de orientação a objetos é usado na maioria das linguagens modernas, e representa uma grande ferramenta para programador.

Multiplataforma O MOS deve funcionar em diversos MOSDeviceTypes, portanto a linguagem deve ser compilável para os mesmos.

Proteção de memória Os MOSMicros criam um kernel para o microcontrolador, fracionando a memória do dispositivo, então o programador não pode ter acesso integral à memória. Como o MOS deve funcionar em microcontroladores que não possuem proteção de memória, a restrição de acesso estará na linguagem que o programador usa.

Alocação Dinâmica A memória de microcontroladores é pequena. Os MOSMicros criam um kernel para o microcontrolador permitindo a criação de threads. Os threads são criados dividindo o restante da memória do microcontrolador pelo número máximo de threads. Permitir a alocação dinâmica neste caso faria com que os threads tenham uma área reservada muita pequena, o que tornaria

impraticável o uso de threads. Portanto a linguagem não pode alocar variáveis de forma dinâmica. Isto é melhor estudado no capítulo 4.3.

System O programador deve ter acesso a uma variável global chamada System que representa o próprio MOS. Com isto um programa é capaz de alterar o funcionamento ou configurar o MOS de acordo com as necessidades da aplicação.

Modules Modules são pequenos programas desenvolvidos pelo programador. Instâncias de Modules podem se comunicar desde que um conheça o outro.

Recursos de Dispositivos Modules podem ser vinculados a dispositivos e com isto ter acesso aos recursos específicos do dispositivo.

Redes de Comunicação e Protocols O programador pode definir protocolos de comunicação para redes (define a sequência de bytes que trafega pela rede) e associá-los a um Module para que este envie ou trate mensagens de um rede.

MOSNetwork A linguagem deve permitir o acesso a uma rede especial do MOS, a MOSNetwork. A rede é capaz de conectar MOSDevices. O MOS usa a rede de várias formas, mas para o programador a rede deve ser usada como um protocolo.

Transferência de Modules O MOS permite que Modules sejam transferidos para outros dispositivos na rede MOS. A linguagem e o seu compilador devem considerar que um Module pode ser compilado para vários tipos de dispositivos e transferidos para eles. Como deve funcionar em microcontroladores, código compilado deve ser associado a tabelas como descrito no capítulo 2.

Fácil uso A linguagem deve permitir que o programador não tenha dificuldades para desenvolver suas aplicações. Para o MOS isto se traduz no programador usar os MOSProject, Module, Protocol antes e durante testes na aplicação.

Pode-se ver que os aspectos necessários existem muitas linguagens, mas não ao mesmo tempo ou da mesma forma. Considere o caso do objeto System que MOS deve ter. O C++ não tem um objeto System. O número de funções e classes que este objeto lidaria seria grandes demais. Ao invés disto o C++ possui as C++ Standard Library que define funções e classes em partes. O programador decide quais bibliotecas irá adicionar em seu código. Java por outro lado define pacotes para organizar as classes.

No entanto, se analisarmos com cuidado veremos que existe apenas uma linguagem de programação chega perto de atender estes requisitos, a linguagem C++. Ela é compilável, possui orientação a objetos e é multiplataforma. Ela é a linguagem

mais difundida em microcontroladores depois de C. Muitos das funcionalidades discutidas poderiam ser implementados em cima do C++ como o System, Modules, Redes e etc (seriam vistos como bibliotecas ou API).

O C++ é uma linguagem muito poderosa. Sendo assim seria necessário impor limitações em cima da linguagem C++. Isto está relacionado à proteção de memória e à alocação dinâmica que são assuntos correlacionados. O operador *new* não poderia ser usado para alocar variáveis, ponteiros não podem ser usados para impedir o acesso indevido a áreas do MOS e o acesso de elementos de arrays nunca poderia passar o tamanho do array. A alocação sem proteção (como no caso de microcontroladores) fará com que a memória de um colida com a do outro, ocasionando um bug. Isto será visto melhor no capítulo 4. A solução adotada no MOS para falta de alocação é a criação de Modules e a conexão que pode ser feita entre eles.

Um outro aspecto importante na adaptação do C++ é a compilação de Module. Os compiladores de C++ não geram as tabelas necessárias para compilar, transferir e instanciar um Module em um dispositivo (MOSBuilder na seção 2.6.1). Portanto seria preciso modificar os compiladores para extrair as tabelas e ajustar o processo de ligação de um programa (os endereços devem ser diferentes).

A linguagem C++ não é a melhor linguagem para iniciantes devido ao seu grau de complexidade. Uma das formas de melhorar este quadro no MOS é utilizar assistentes de programação. São ferramentas que agilizariam a programação no MOS. O exemplo mais contundente é o Protocol. É muito melhor que o programador desenvolva um Protocol visualmente como na figura 2.12 do que com código. Na realidade é preferível que o programador não possa desenvolver os Protocols escrevendo, apenas com a ferramenta. As ferramentas estão relacionadas a IDEs (Integrated development environment) e portanto não são o foco do trabalho. Obviamente isto poderia ser feito com muitas outras linguagens além da C++.

A questão é que no final nenhuma linguagem contém todas as características desejadas. Ao impormos limitações na linguagem C++ (a que mais atende as necessidades) na realidade teríamos uma outra linguagem de programação. É por isto que ao invés de trabalhar a C++ o trabalho propõe uma nova linguagem de programação mesmo explicado-a superficialmente, a MOSLanguage. Isto permite que os conceitos desenvolvidos no MOS sejam aplicados de forma direta.

3.2 Descrição

Uma característica da MOSLanguage é ser uma linguagem orientada a objetos. Uma linguagem desta natureza permite que os programadores não definam apenas os tipos de dados (*Data Type*) de uma estrutura de dados (*Class*), mas também os tipos de operação (*Function* ou *Method*) que podem ser usados nestas estruturas.

A MOSLanguage permite a criação de classes (*Classes*), ou seja, estruturas que possuem atributos (*Attributes*) de *Data Types/Classes*, e possuem métodos (*Methods*). Atributos são objetos de outras classes ou dos tipos de dado (data types) básicos do MOSLanguage. Métodos são funções que possuem uma certa quantidade de entradas (argumentos), outra quantidade de saídas (retornos) e acesso a variáveis internas (atributos) da classe.

O código 3.1 representa uma classe de exemplo com um atributo e dois métodos. O segundo método é desenvolvido para indicar como as variáveis podem ser usadas.

Código 3.1: Exemplo de classe na MOSLanguage com um atributo e duas funções

```
Class ExampleClass {
  AttributeType attribute;
  [] Method1();
  [ReturnType return1, ReturnType return2] Method2(
     ArgumentType argument1, ArgumentType argument2);
};

[ReturnType return1, ReturnType return2] Method2(
   ArgumentType argument1, ArgumentType argument2)
{
  return1 = attribute;
  return2 = argument1;
}
```

Existem muitas propriedades que compõe a orientação a objetos. A MOSLanguage não possui todas estas propriedades. Dentre as que possui estão o Polimorfismo, a Herança e a Abstração [47]. As classes possuem os tradicionais especificadores de acesso a atributos: *public*, *private* e *protected*. O especificador *public* permite que variáveis e funções sejam usadas fora da classe. No *private* apenas métodos da classe podem usar os atributos/métodos privados. O *protected* permite que classes derivadas tenham acesso aos atributos/métodos. O código mostra um exemplo de derivação de classes. Três classes A, B e C são definidas. B é uma classe derivada de A portanto tem acesso aos atributos/funções públicos e protegidos. C possui um objeto de A então tem acesso apenas aos atributo/função público de A.

Código 3.2: Exemplo de derivação de classes

```
Class A
{
private:
  AtributeType private_attribute;
```

```
  [] private_function();
protected:
  AtributeType protected_attribute;
  [] protected_function();
public:
  AtributeType public_attribute;
  [] public_function();
};

Class B : Class A
{
  [] Function()
  {
    protected_attribute;
    public_attribute;
    protected_function();
    public_function();
  }
};

Class C
{
  A a;
  [] Function()
  {
    a.public_attribute;
    a.public_attribute;
  }
}
```

A MOSLanguage define alguns tipos de dados básicos como os definidos em outras linguagens de programação. Os tipos básicos são mostrados na tabela 3.1.

Os valores dos tipos na tabela 3.1 seguem padrões de dados. O *char* segue o padrão UTF-8[48]. Os inteiros com sinal seguem o padrão Complemento a 2[49]. Os *float32* e *float64* seguem o padrão binário de dados de ponto flutuante descrito no documento IEEE 754[50].

A MOSLanguage não restringe a ordenação dos data types (Endianness) como sendo sempre de um tipo, o Little Endian ou o Big Endian. Cada MOSDeviceType trabalha com um dos tipos. Como ModClasses são compiladas para certo MOSDe-

Data Type	Descrição	Valor
bool	Booleano, tipo de dado que possui dois valores (verdadeiro ou falso)	true/false
char	Caractere, uma unidade de informação que representa um símbolo como uma letra do alfabeto	'a', 'b', etc
int8	Inteiro de 8 bits	-128 a 127
int16	Número inteiro de 16 bits	-2^{15} a $2^{15}-1$
int32	Número inteiro de 32 bits	-2^{31} a $2^{31}-1$
int64	Número inteiro de 64 bits	-2^{63} a $2^{63}-1$
uint8	Número inteiro de 8 bits sem sinal	0 a $2^{8}-1$
uint16	Número inteiro de 16 bits sem sinal	0 a $2^{16}-1$
uint32	Número inteiro de 32 bits sem sinal	0 a $2^{32}-1$
uint64	Número inteiro de 64 bits sem sinal	0 a $2^{64}-1$
float32	Número ponto flutuante de 32 bits	dígitos: 24; expoente: -126 a 127
float64	Número ponto flutuante de 64 bits	dígitos: 53; expoente: -1022 a 1023

Tabela 3.1: *Data Types* na MOSLanguage

viceType, é o MOSDeviceType que vai indicar se o o tipo será Little Endian ou Big Endian.

A MOSLanguage permite a criação de arrays (arranjo sistemático) de tipos de dados básicos ou classes. Eles sempre possuem tamanho fixo (só é permitido alocação estática na MOSLanguage). Strings na linguagem são vistos como array de *char* e continuam tendo a restrição de tamanho de arrays. Na MOSLanguage é possível definir operadores para classes assim como em outras linguagens como *[] {} + - = < >*. Os *Data Types* de números possuem operadores *+*, *-*, *=* etc. que permitem operações matemáticas simples.

No acesso a um elemento de um array no C++ não se verifica se o acesso está ocorrendo fora do vetor, ou seja, é um acesso não seguro. A MOSLanguage faz um acesso seguro de elementos de um array, então quando um elemento de um array é acessado verifica-se o tamanho do array. A verificação a todo elemento traz segurança ao MOS em detrimento de um tempo maior de acesso. Não é conveniente discutir as manobras que a MOSLanguage e o seu compilador usariam para realizar este precedimento (como a otimização para acessos em sequência), portanto o foco estará em uma sintaxe simples. O código 3.3 exemplifica o uso de arrays na MOSLanguage. Repare que existem duas formas de acessar o mesmo dado, uma ignorando que o acesso por dar errado e outra que sabe-se se deu certo ou errado. O protótipo de acesso de um array *a* do tipo *Type a[ArraySize]* é *[bool accessok, Type arrayelementvalue] operator[](ArrayPosition position)*. O valor do *arrayelementva-*

lue quando o acesso falha é de lixo (o que quer que esteja na memória). Desta forma tanto o programador pode verificar se o acesso falhou quanto o MOS não será acessado indevidamente. A escrita funciona de forma similar com o operador *[bool writeok] operator[](ArrayPosition position, Type arrayelementvalue)*

Código 3.3: Exemplo de uso de arrays

```
int8 data[10];
int8 dataelement;
int8 newdataelement = 82;
bool accessok;
bool writeok;

// acesso não seguro:
dataelement = dado[5];

// acesso seguro
[accessok, dataelement] = data[5];
// equivalente a [accessok, dataelement] = data.operator
   [](5);

// escrita não segura
data[5, newdataelement];
// escrita segura
[writeok] = data[5, newdataelement];
// equivalente a [writeok] = data.operator[](5,
   newdataelement);
```

O *char* pode ser usado como array para que os valores de char sejam escritos/-lidos individualmente. No entanto o *char* segue o UTF-8, então todos os caracteres fora do ASCII só podem ser representados com mais de um char. No UTF-8 um caractere pode ter até 4 bytes. A MOSLanguage define o operador *[bool accessok, char character[4]] operator[[]](CharacterPosition position)* e *[bool writeok] operator[[]](CharacterPosition position, char character[4])* de vetores de *char* para escrever ou ler um caractere de um vetor de *char*. Note que este operador não é uma simples escrita e leitura de um byte. Uma busca é feitas em cada byte para verificar caracteres no UTF-8 até que encontre-se o caractere desejado. A alteração do caractere pode alterar vários elementos do vetor. O código 3.4 mostra como os operadores são usados.

Código 3.4: Exemplo de uso de string

```
// a variável string tem 3 caracteres: 'a' de 1 byte, '®' de
    2 bytes e 'b' de 1 byte
char string[15] = "a®b";

// leitura do segundo carectere (secondchar fica com valor ®
   )
char secondchar[4] = string[[2]];

// escrita no segundo caractere faz com que a variável
   string tenha apenas 3 bytes sendo usados porque o
   caracter 'x' utiliza apenas 1 byte
string[[2, 'x']];
```

Devido a natureza das aplicações do MOS, a possibilidade da MOSLanguage permitir cálculos complexos deve existir. Portanto seria bom que a MOSLanguage tivesse suporte a matrizes, de forma similar ao Matlab[51] para que mais tarde a comunidade do MOS crie e disponibilize "bibliotecas" no MOS relacionados a cálculos com matrizes. Esta dissertação não define o uso de matrizes mas isto não significa que não podem ser adicionadas a posteriori. Elas deveriam funcionar como os arrays que verificam o acesso de seus elementos.

Funções das classes podem criar variáveis locais. Estas variáveis são alocadas automaticamente na Pilha de Chamada do thread que executa a função. O código 3.5 mostra uma classe *A* com um único método que cria e usa uma variável local (*temp*).

Código 3.5: Exemplo de variáveis locais

```
Class A
{
  [float16 ret] Half(float16 value)
  {
    float16 temp;
    temp = value/2;
    ret = temp;
  }
};
```

O programador não pode definir variáveis globais na MOSLanguage. Existe apenas um único objeto global que é definido pela própria linguagem, o objeto *Sys* da classe *System*. A classe não possui atributos públicos, apenas métodos públicos. Toda classe tem acesso a este objeto mas partes do *System* são visíveis apenas

se determinadas condições forem satisfeitas. A classe representa o próprio MOS e portanto permite que classes normais interajam com o MOS. Discutiremos mais sobre suas peculiaridades mais a frente.

A MOSLanguage define um *Module*, que nada mais é do que a extensão de uma *Class* (possui atributos e métodos). Quando um programador define um *Module* na verdade define uma ModClass. Os *Modules* seguem as mesmas regras de orientação a objetos das *Classes*. *Modules* podem ser derivados de *Classes* ou de outros *Modules*, mas *Classes* não podem ser derivadas de *Modules*. O código 3.6 exemplifica a derivação de *Modules*.

Código 3.6: Exemplo de derivação de *Modules*

```
Class A
{
public:
  bool atr;
  [] Func() {...};
};

Module A : Class A
{
public:
  bool atr2;
  [] Func2()
  {
    ...
    atr = true;
    Func();
  }
};

Module B : Module A
{
  Func3()
  {
    ...
    atr = false;
    atr2 = false;
    Func();
    Func2();
```

```
  }
};
```

O que o *Module* tem de especial é a sua interação com o MOS. O *Module* representa um pequeno programa que pode ser compilado, transferido e instanciado em um MOSDevice. Os *Modules* podem conter atributos que são outros *Modules*. Isto é similar a classes terem atributos de outras classes. Este atributos não estão usáveis quando o *Module* inicia. O *System* deve ser usado para conectar instâncias de modules. Quando conectados, o Module pode ser acessado como um atributo da classe (código 3.7). Note que todo *Module* deve indicar o número de LinkSlots de conexões se não quiser aceitar apenas uma conexão. A interação entre *Module* e *System* será vista mais tarde. O *Module* é um paradigma interessante para o programador que deve ver sua aplicação como um conjunto de pequenas classes que devem interagir.

Código 3.7: Exemplo de um Module usando outro

```
Module A <LinkSlots:3>
{
public:
  bool atr;
  [] Func() {...};
};

Module B
{
  A a;
  Func()
  {
    a.atr = true;
    a.Func();
  }
};
```

Modules podem utilizar o sistema de passagem de mensagens no MOS. Os *Modules* podem definir mensagens e o o tratamento das mesmas. Ao definir uma mensagem o *Module* automaticamente é capaz de enviar a mensagem com uma função. Se o tratamento da mensagem não for definido. Considere que um *Module B* possui um objeto de outro *Module A*. O *B* pode definir o tratamento das mensagens definidas em *A* e pode enviar as mesmas mensagens. Quando dois *Modules* conectados enviam mensagens os dois recebem as mensagens e as tratam se o tratamento estiver definido. O código 3.8 mostra que as mensagens definidas em *A* podem ser enviadas

por *B*. Note que quando *TestMessage1* for enviada apenas *B* irá tratar, e quando *TestMessage2* for enviada apenas *A* irá tratar. Vale lembrar que o tratamento das mensagens é feita em um thread especial do MOS, o MPThread.

Código 3.8: Exemplo de uso de mensagens em Modules

```
Module A
{
Message:
  TestMessage1;
  TestMessage2 {...};
private:
  Func()
  {
    SendMessage(TestMessage1);
  }
};

Module B
{
Message:
  a.TestMessage1() {...};
private:
  Module a;
  Func()
  {
    a.SendTestMessage1();
    a.SendTestMessage2();
  }
};
```

Modules podem ser vinculados a um MOSDeviceType. Quando isto acontece o *Module* passa a ter acesso a um conjunto maior de funções no objeto *Sys*. Da mesma forma que uma *Class* não pode ser derivada de um *Module*, um *Module* vinculado não pode ser derivado de um *Module* não vinculado. Mais ainda, um *Module* vinculado só pode ser derivado de outro *Module* com o vínculo do mesmo MOSDeviceType. O código 3.9 mostra esta derivação.

Código 3.9: Exemplo de derivação de Modules vinculados

```
Class A {...};
Module B : Class A <Windows> {...};
```

```
Module C {...};
Module D : Module C <ATmega2560> {...};
Module E : Module D {...}; // continua vinculado ao
   ATmega2560

Module F <Linux>{...};
```

Além das *Classes* e *Modules* a MOSLanguage permite a criação dos *Protocols*. O *Protocol* está relacionado à definição de um protocolo de mensagem a ser usado em uma rede. O protocolo consiste em um arranjo de Fields (campos). Todo Field possui um nome e um tipo (*Data Type* ou *Class*). O *Field* pode conter um valor fixo. É possível adicionar condições em um *Field* que retornam *true* ou *false*. Quando tenta-se verificar se uma mensagem está de acordo com um protocolo na realidade verifaca-se estas condições. As condições em um *Field* possuem uma ordem e são verificadas até que uma seja *true*, indicando quais são os próximos *Fields*. O programador deve indicar um nome para as sequências de *Fields* para transformá-los em mensagens. Um *Protocol* deve ter o formato da figura 2.12 apresentada no capítulo 2.

Neste capítulo são usados códigos para exemplificar a sintaxe da MOSLanguage no MOS. Para ajudar os programadores a escrever um *Protocol* seria ideal o auxílio de uma ferramenta gráfica. As figuras do Protocol na seção 2.11.2 valem de ilustração para o *Protocol*, incluindo as sobre a derivação de Protocols. A ferramenta deve permitir que o programador crie *Fields*, adicione condições e defina as mensagens do *Protocol* como nas figuras. Caso não exista uma ferramenta, o programador ainda pode gerar um código na MOSLanguage para criar *Protocols*. O código 3.10 exemplifica como *Protocols* podem ser criados e derivados, e a figura 3.1 representa estes *Protocols*. Note que a definição de *Protocols* por código não é muito amigável.

Código 3.10: Exemplo de Protocols definidos na MOSLanguage. O Protocol P2 é derivado do P

```
Protocol P
{
  char start = 's'
  > char command
    <command == 0>
    > bool value1
    > int value2
    > char end = 'e'
    : MsgA
    <<<
```

```
    <command == 1>
    > float value3
    > char end = 'e'
    : MsgB
};

Protocol P2 : Protocol P
{
// Uma nova mensagem (MsgC) é adicionada entre <command ==
   0> e <command == 1>
  > : MsgA <
  < < <
  <command == 2>
  > int16 value4
  > char end = 'e'
  : MsgC

// O field value4 do tipo int8 é adicionado depois do value3
  > float value3 <
  > int8 value5

// A MsgD é adicionada acrescentando uma condição em value1
  > bool value1 <
  <value1 == true>

  > : MsgA <
  < <
  <value1 == false>
  > char end = 'e'
  : MsgD
};
```

Para que o *Protocol* seja usado em um *Module* deve-se usar um objeto especial chamado *NetSlot*. Analisaremos-no melhor posteriormente, mas basta compreender que o *NetSlot* pode se associar a uma rede no MOSDevice ao ser usado no *Sys*, e que ele sempre deve ser associado a um *Protocol* para o *Module* ser compilado. Quando o *NetSlot* se liga a uma rede as mensagens definidas no Protocol podem ser enviadas e recebidas. O programador pode enviar mensagens do *Protocol* e define como as mensagens do Protocol devem ser tratados no *Module* usando o *Net* como

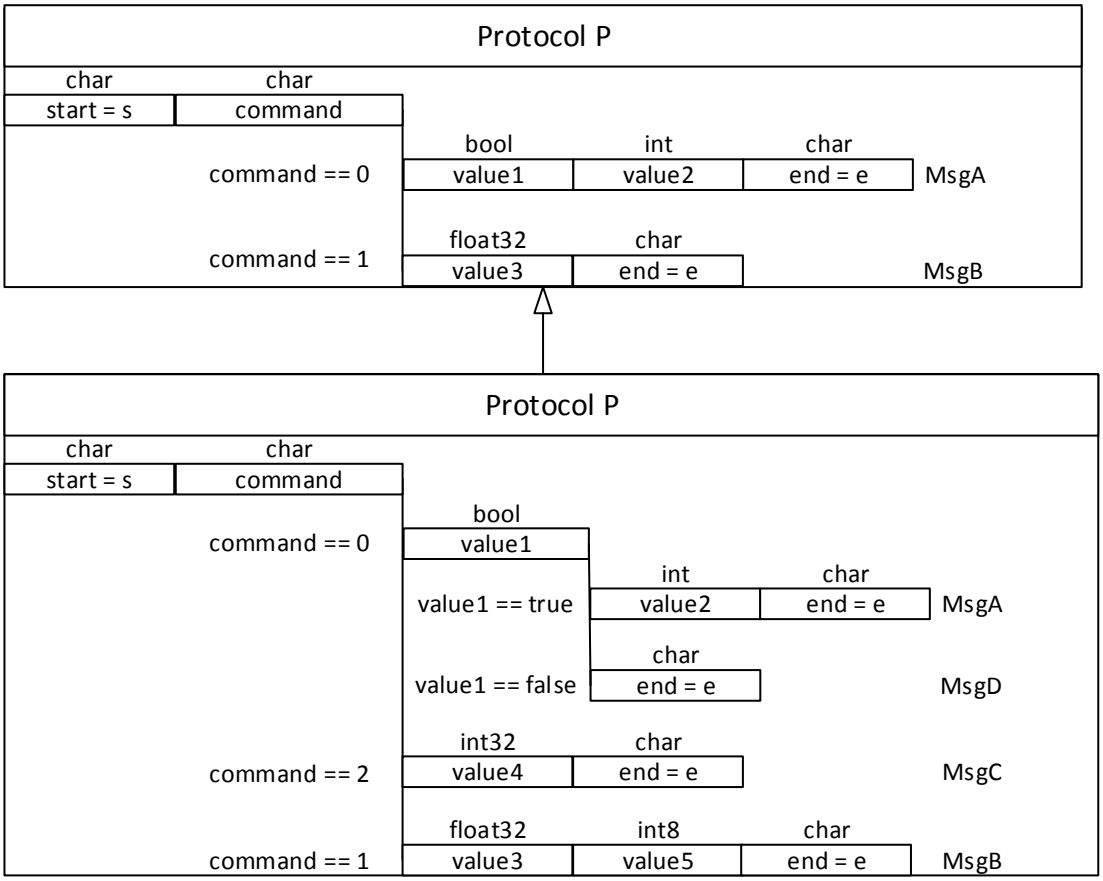

Figura 3.1: Protocolo P com duas mensagens (MsgA e MsgB).

intermediário. O código 3.11 mostra um *Module* que usa o *Protocol P* definido no código 3.10.

Código 3.11: Exemplo de Protocol em um Module usando uma rede RS232

```
Module M
{
  NetSlotRS232 rs232 <Protocol P>;

  rs232.MsgA() {...};
  rs232.MsgB() {...};

  [] Func()
  {
    rs232.SendMsgA(true, 40); // value1 = true | value2 = 40
    rs232.SendMsgB(3.85); // value3 = 3.85
  }
};
```

Por último, o programador pode criar um *MOSProject*, um projeto da linguagem para agrupar as *Classes*, *Modules* e *Protocols* definidos pelo programador. Um *MOSProject* pode ser dependente de outro MOSProject. Neste caso o projeto dependente tem acesso a todas as *Classes*, *Modules* e *Protocols* definidas pelo programador. Isto é similar a incorporar o código fonte de um projeto em outro projeto. O *MOSProject* está correlacionado ao MOSBuilder.

O *MOSProject* faz um pequeno controle de versão nos *Modules*. Sempre que uma mudança é feita em um *Module* e ele é compilado o número de sua MVersion incrementa. Sempre que uma mudança na parte pública de um *Module* é feita sua MPublicVersion é incrementada. Os números são usados para criar *Modules* de certa versão e ter certeza que *Modules* são compatíveis (*Modules* incompatíveis não podem ser conectados). Um *Module* pode acessar o *Sys* para que este inicie outro *Module*. Neste momento o MOSDevice pode passar o pedido para o que contém o *MOSProject* do outro. O MOSBuilder compila o *Module* do *MOSProject* e o transfere para o primeiro *Module* onde será instanciado. A abordagem de tornar o *MOSProject* mais do que um simples repositório de código distingue a MOSLanguage de outras linguagens.

3.3 Classes Especiais

As *Especial Classes* são classes definidas pela própria MOSLanguage assim como os *Data Types*. A diferença é que estas classes tem funcionalidades diferentes dos *Data Types*, talvez tenham restrições de onde podem ser criadas ou usadas e estão ligadas ao *System*.

Time

A classe *Time* representa um intervalo de tempo. Esta classe não tem restrição de onde pode ser criada, ou seja, um objeto da classe pode ser criado como uma variável local ou um atributo de uma *Class/Module*. O código 3.12 mostra o protótipo da classe (mesmo não sendo feita em C++). Ela possui apenas duas variáveis para indicar o tempo passado em segundos e nanosegundos desde um instante de referência. Esta referência depende da lógica do programador.

Código 3.12: Protótipo da classe Time

```
Class Time
{
public:
  int64 seconds;
  int32 nanoseconds;
};
```

A classe System possui apenas duas funções relacionada à classe Time (código 3.12). A primeira (*[Time timenow] TimeNow()*) indica o tempo passado desde uma referência como o Epoch time (tempo desde 1/1/1970 em Greenwich). O programador deve tomar cuidado pois muitos dispositivos não possuem um referencial de data, então neste caso o tempo de referência seria o instante em que o dispositivo foi iniciado. A segunda função ([] Sleep(Time sleeptime)) faz com que o o thread que chama a função seja suspenso pelo intervalo de tempo indicado.

Código 3.13: Funções relacionadas ao Time no System

```
Class System
{
public:
  [Time timenow] TimeNow();
  [] Sleep(Time sleeptime);
};
```

Note que com estas simples primitivas o programador tem a liberdade para criar seu próprios algoritmos envolvendo o tempo. Existe a possibilidade de incorporar as funções do *System* para dentro do *Time*, mas opta-se por mantê-las separadas para manter a similaridade com outras linguagens.

Thread

No MOS o escopo inicial dos threads é o escopo da classe que a cria. O código 3.14 indica como um thread é gerado na MOSLanguage. O atributo da classe *Thread* é associado a uma função da classe, e é esta função que será executada no thread (por isto possui o escopo da classe). O especificador de acesso no *Thread* independe da função que o thread executará. O código em 3.15 mostra que a classe tem apenas um método, o *Start()* que inicia o thread, e falha se o thread já está sendo executado. A *Start()* também pode fazer um thread ser reiniciado após ter terminado sua execução (função associada termina).

Código 3.14: Definindo um thread na MOSLanguage

```
Class A
{
  [] ThreadFunction1();
  [] ThreadFunction2();

  Thread t1 <ThreadFunction1>;

public:
  [] ThreadFunction3();
```

```
  Thread t2 <ThreadFunction2>;
  Thread t3 <ThreadFunction3>;
};
```

Código 3.15: Protótipo da classe Thread

```
Class Thread
{
public:
  [bool started] Start();
};
```

Os threads são definidos em *Classes* que está dentro de outra *Class* ou de um *Module*, e por isto dizemos no MOS que o dono do thread é uma ModIn. No MOS os threads são alocados para as ModIns quando estas são instanciadas. Desta forma o programador não precisa se preocupar com o thread estar disponível ou não no sistema (problema dos MOSMicros). Se não houverem threads disponíveis a ModIn simplesmente não é criada. Por isto objetos de uma classe que utilize threads não podem ser criado como uma variável local (dentro de uma função), já que isto necessitaria que um novo thread fosse alocado para a ModIn.

Objetos de Sincronização

Existem muitos objetos de sincronização que poderíamos tratar mas focaremos apenas em um, o que foi o foco no capítulo 2. O Mutex era associado ao Lock e Unlock. A classe *Mutex* é representada no código 3.16 e de fato possui apenas função para estas operações. A diferença entre o *Lock()* e o *TryLock()* é que o primeiro pode suspender um thread indefinidamente até ganhar o direito de acesso à região crítica, enquanto o segundo nunca trava e indica se conseguiu ou não o acesso exclusivo.

Código 3.16: Protótipo da classe Mutex

```
Class Mutex
{
public:
  [] Lock();
  [bool locked] TryLock();
  [] Unlock();
};
```

O código 3.17 mostra como os objetos são criados e usados em classes. Os objetos de sincronização são muito parecidos com os threads nas restrições de uso. Os objetos só podem ser criados dentro de classes e estas classes não podem ser

criadas como variáveis locais. Os donos dos objetos são as ModIns. Se os objetos de sincronização não estão disponíveis a ModIn não pode ser criada.

Código 3.17: Usando Mutexes na MOSLanguage

```
Class A
{
  Mutex a;
public:
  Mutex b;
  [] Func()
  {
    a.Lock();
    a.Unlock();
    b.TryLock();
    b.Unlock();
  };
};
```

Module

A classe *Module* já foi parcialmente tratada neste capítulo. Existem três coisas a mais que devem ser consideradas, quem pode criar objetos Module, o procedimento para criar/conectar *Modules* e a segurança na comunicação (um *Module* chama funções ou acessa variáveis do outro).

Um objeto de *Module* só pode existir dentro de outro como um atributo privado/protegido. Existem informações extras relacionados à conexão de um Module que não são armazenados em uma classe simples (ex: nome do MOSProject e ID do module). O atributo não pode ser público porque o acesso de um Module não é uma simples implementação, então proíbe-se o acesso aninhado de Modules.

O código 3.18 mostra as funções relacionadas ao controle de *Modules* dentro do *System*. A classe *ModuleID* identifica um Module dentro do MOSDevice representa o número das ModIns dentro do MOS. Quando o *ModuleID* tem o valor 0 significa que não identifica nenhuma ModIn. Esta classe não tem qualquer restrição sobre onde pode ser instanciada.

O *System* fornece as seguintes opções em relação a ModIns: buscar uma instância compatível dentro do MOSDevice, criar uma instância de um *Module* conhecido com certa versão (normal ou pública), deletar uma instância de um Module, retirar informações básicas de ModIns e conectar uma instância a um objeto *Module*. Quando o retorno é um *Module* deve-se usar o objeto *Module* que deseja-se conectar a um de verdade.

Código 3.18: Funções relacionadas ao Module no System

```
Class ModuleID
{
  uint16 modulenumber = 0;
public:
  [uint16 ret] Operator= (uint16 arg);
  [bool equal] Operator== (uint16 value);
  [bool equal] Operator== (ModuleID id);
};

Class System
{
public:
  [ModuleID id] SearchModule(ModuleID searchstart, char
     mosproject[20], char module[20], uint16 version, uint16
      pubversion);
  [bool created, ModuleID id, Module module] CreateModule(
     char mosproject[20], char module[20], uint16 version,
     uint16 pubversion);
  [bool created, ModuleID id, Module module] CreateModule(
     Module moduleinfo); // o moduleinfo é usado apenas para
      forncer as informações de nome e versão
  [bool connected] ConnectModule(Module module, ModuleID id)
     ;
  [] DestroyMdule(ModuleID id);
  [] KillModule(ModuleID id); // destrói um Module a força
};
```

Quando um código chama funções ou acessa atributos de um objeto da classe *Module* pode ser usado sem estar conectado uma ModIn. O código 3.3 mostra a função de um *Module* sendo alterada e ganhando um novo retorno que indica se a função foi usada com sucesso ou não. Algo similar é feito com o atributo. No final as variáveis *accessok* e *enterok* indicam se o acesso e a entrada em *a* foi feito com sucesso. Com isto o programador sabe se o *Module* foi usado com sucesso e suas variáveis conterão lixo.

```
Module A
{
  char hiddenchar;
public:
```

```
  bool var;
  [char ret] GetChar()
  {
    ret = hiddenchar;
  };
};

Module B
{
  A a;
public:
  [] Fun()
  {uma
    bool test;
    char letter;

    // uso normal
    test = a.var;
    letter = a.GetChar();



    // uso seguro
    bool accessok;
    bool enterok;

    [accessok, test] = a.var
    [enterok, letter] = a.GetChar();
    if (accessok && enterok) {...}
  };
};
```

NetSlot

A *NetSlot* é na realidade um conjunto de classes usadas para associar *Protocols* a redes, e com sito enviar/receber mensagens pelas redes. No MOS tratamos apenas as redes TCP/IP, RS232 e MOSNetwork, e então existirão as *NetSlotTCPIP*, *NetSlotRS232* e *NetSlotMOSNetwork*. Estas classes não tem restrição de onde podem ser criados, no entanto, quando são criados como variáveis locais não será possível associá-los a um *Protocol*, então os objetos servirão apenas como referência para um Net.

A associação de uma *Net* já foi demonstrada portanto o foco será em como o *System* lida com esses objetos. O modelo adotado é muito semelhante ao da interação do *System* com o *Module*. O código 3.19 mostra as principais funções. A classe *NetID* tem a mesma função do *ModuleID* identificando todas as Nets em um MOSDevice. O código utiliza uma nova sintaxe para indicar as funções que são acessíveis apenas para Modules vinculados a MOSDeviceTypes, e estas funções estão relacionadas a criação de redes. Isto é justo já que cada rede é configurada de forma diferente em cada dispositivo. A MOSNetwork é um caso especial. O programador não precisa criar uma MOSNetwork como faz com as outras, mas precisa adicioná-la em outras (o MOSNetProtocol é adicionado em outras Nets).

Código 3.19: Funções relacionadas ao NetSlot no System

```
Class NetID
{
  uint16 modulenumber = 0;
public:
  [uint16 ret] Operator= (uint16 arg);
  [bool equal] Operator== (uint16 value);
  [bool equal] Operator== (ModuleID id);
};

Class System
{
public:
  [NetID id] SearchNet(NetID searchstart, uint8 nettype);
  [bool added] AddNetSlotToNet(NetSlot slot, NetID id);
  [bool removed] RemoveNetSlotFromNet(NetSlot slot);
  [] DestroyNet(NetID id);

  [bool added] AddMOSNetworkToNet(NetID id);

<<Windows>>
  [bool created, NetID id] CreateTCPIPServer(NetSlotTCPIP
    slot, uint16 port);
  [bool created, NetID id] CreateTCPIPClient(NetSlotTCPIP
    slot, uint16 port, int8 ip[4]);
  [bool created, NetID id] CreateRS232(NetSlotRS232 slot,
    uint8 comport);
```

```
<<Linux>>
  [bool created, NetID id] CreateTCPIPServer(NetSlotTCPIP
    slot, uint16 port);
  [bool created, NetID id] CreateTCPIPClient(NetSlotTCPIP
    slot, uint16 port, int8 ip[4]);
  [bool created, NetID id] CreateRS232(NetSlotRS232 slot,
    char deviceaddress[20]);

<<ATmega2560>>
  [bool created, NetID id] CreateRS232(NetSlotRS232 slot,
    uint8 usartnumber);
};
```

Uma observação deve ser feita em relação aos tipos Little e Big Endian em redes. É comum redes não definirem qual dos tipos deve ser usado na sequência de bytes que trafegam na rede. Fica a cargo da aplicação decidir qual tipo será usado. Existe uma convenção de uso ddo Big Endian no TCP/IP mas não é uma obrigação. Na RS232 é mais comum o uso do tipo Little Endian do que do Big Endian. A MOSLanguage lida com isto nos NetSlots. Em todo NetSlot pode-se especificar qual o tipo de ordenação será usada no envio e recebimento de mensagens, caso contrário será usado o tipo de ordenação do MOSDeviceType. Desta forma ainda cabe ao programador especificar qual tipo será usado na comunicação de acordo com a aplicação.

System

A classe System é uma classe especial que não pode ser criada pelo programador. Boa parte da classe *System* já foi descrita. O que falta para terminar sua análise explanar mais sobre o comportamento da classe nos Modules vinculados a MOSDevices. As funções de vínculo que ainda não foram mencionadas estão relacionadas às configurações de Escalonamento e uso do hardware específico de MOSDevices.

O Escalonador dos Sistemas Operacionais Windows e Linux assim como o do próprio MOS para microcontroladores será explicado no capítulo 4. Para a MOSLanguage basta saber por alto como funcionam os Escalonadores. Em geral não há muita liberdade para alterar o Escalonador de SOs, todavia, estes em geral permitem a alteração da prioridade de seus threads. Por outro lado o Escalonador/threads do MOSMicro dependem exclusivamente da implementação do MOS nestes dispositivos.

No Windows a prioridade do thread é montada com o Priority Level e o Priority Class, sendo o primeiro específico do thread e o segundo do processo. Como o MOSProcess é um processo, o Priority Class é visto como uma configuração do

Escalonador no MOS, e o Priority Level do thread. No Linux existe uma prioridade e uma política de escalonamento a serem selecionadas para um thread. No MOSMicro do ATmega2560 é possível escolher o uso de dois Escalonadores, o Round Robin e a FIFO, mas os threads em si não tem nenhuma configuração de prioridade.

Quando a classe *Thread* foi apresentada ignorou-se o escalonamento. Apesar do de um thread ter um dono, o thread pode estar acessando qualquer ModIn e em qualquer classe. Obviamente o objeto dessa classe (usado para definir o thread) pode estar inacessível, portanto optou-se por passar a configuração dos threads para o *System*. Não problema para identificar o thread que será configurado (está sendo usado) porque o System sabe qual é o thread que está executando.

O código 3.20 contém as funções do System relacionados aos threads e Escalonadores que são acessíveis apenas em Modules vinculados.

Código 3.20: Funções relacionadas ao Escalonador e Threads no System

```
Class System
{
public:
  [] SetScheduler(uint8 schedulernumber);
  [uint8 schedulernumber] SetScheduler();

<<Windows>>
  [] SetSchedulerPriorityClass(uint8 priorityclass);
  [uint8 priorityclass] GetSchedulerPriorityClass();
  [] SetThreadPriorityLevel(uint8 prioritylevel);
  [uint8 prioritylevel] GetThreadPriorityLevel();

<<Linux>>
  [] SetThreadPolicy(uint8 policy);
  [uint8 policy] GetThreadPolicy();
  [] SetThreadPriority(uint8 priority);
  [uint8 priority] GetThreadPriority();

<<ATmega2560>>
  [] SetSchedulerRoundRobinTimeSlice(uint32 microseconds);
  [uint32 microseconds] GetSchedulerRoundRobinTimeSlice();
};
```

No trabalho não se discutiu sobre o uso de hardware específico de SOs no MOS, e sim de microcontroladores, portanto será discutido apenas as mudanças de microcontroladores. Vale ressaltar que mesmo não indicando funções sobre hardware nos

SOs, elas devem existir.

Algumas das funções já foram mencionadas, as relacionadas à criação e destruição de redes. Além de periféricos de rede (do qual mencionamos apenas a USART para a RS232) o ATmega2560 possui temporizadores/contadores, saídas PWM, conversores analógico para digital, comparador analógico etc. Os periféricos usam as portas do chip que podem ser controlados diretamente pelo programador. A MOSLanguage deve evitar o conflito de uso dessas portas. Os Timers também não deveriam ser usados pois estes poderiam interferir com o algoritmo de escalonamento do MOSMicro. O código 3.21 mostra algumas das funções que a MOSLanguage forneceria para o ATmega2560.

Código 3.21: Funções relacionadas ao hardware de MOSDevices

```
Class System
{
public:
<<ATmega2560>>
  [bool setok] SetPortA1(bool onoff);
  [bool onoff] GetPortA1();
  ...
  [bool setok] SetPortL8(bool onoff);
  [bool onoff] GetPortL8();
  ...
  [bool setok] SetPWM1(uint32 frequency, uint8 percentage);
  ...
  [uint16 analog1] GetAnalog1();
};
```

Capítulo 4

Arquitetura De Software

Nos capítulos anteriores, foram apresentadas as funcionalidades do MOS e a sua linguagem MOSLanguage, deixando claro as capacidades do MOS. Com esta base, este capítulo discute como a arquitetura de software do MOS pode ser construída. Para tanto, são apresentadas as classes e funções organizadas em grupos. É importante ressaltar que a linguagem de programação usada dentro do MOS é a C++ por ser a única linguagem orientada a objetos compatível em microcontroladores AVR (ATmega2560 usado como modelo de microcontrolador nesta dissertação), Windows e Linux.

4.1 Apresentação das classes

O capítulo 2 foi responsável por apresentar os conceitos do MOS e indicar como deve funcionar, já este capítulo é responsável por mostrar como o MOS pode ser implementado. O MOS não foi implementado de verdade. Um código foi desenvolvido para verificar como o MOS pode ser implementado mas não é um código funcional. Este código é representado pelas classes a serem discutidas nesta seção, mostrando a arquitetura de software do MOS.

Para facilitar a compreensão da arquitetura, as classes serão divididas nos grupos, de acordo com suas funcionalidades: GScheduler, GThreads, GSyncObjects, GLink, GNetwork, GModule e GModuleOS. Em geral, os grupos possuem classes para representar uma funcionalidade considerando MOSDeviceType diferentes. Seguindo a orientação a objetos, tem-se em geral uma classe genérica e depois suas derivações para MOSDeviceTypes. A arquitetura proposta não será extensivamente detalhada. O objetivo é mostrar quais dados e funções são mais relevantes e deixar a implementação das funções em aberto. Serão considerados apenas os MOSDeviceTypes Windows, Linux e ATmega2560.

O GModuleOS não será tratado isoladamente. Parte de suas classes serão apresentadas juntamente com as dos outros grupos. As classes desse grupo são o System

na MOSLanguage. Quando as funções das classes tiverem um "*", isto indica que as funções estarão disponíveis no System.

4.1.1 GScheduler e GThreads

Todo Sistema Operacional tem um Scheduler ou, em português, Escalonador. O Escalonador faz parte do Kernel de Sistemas Operacionais. Ele executa algoritmos de escalonamento que permitem processos e threads acessarem recursos do sistema (comunicação, tempo de processamento, etc). O MOS deve permitir que o programador tenha algum acesso às configurações básicas do Escalonador, processo ou thread.

Os grupos GScheduler e GThreads serão apresentados juntos por estarem muito correlacionados.

Schedulers de MOSDeviceTypes

Para que os grupos GScheduler e GThreads sejam entendidos, será feita uma breve analise dos Escalonadores dos MOSDeviceTypes considerados nesta dissertação.

Windows

De acordo com [4], no Windows, os threads são escalonados para execução de acordo com a sua prioridade, indo de 0 (menor prioridade) a 31 (maior prioridade). O Escalonador do Windows trata todos os threads com a mesma prioridade como iguais. O sistema atribui fatias de tempo (de tamanho fixo) em um modelo Round-Robin [14] para todos os threads com a maior prioridade. Se nenhum destes thread está pronto para execução, o Escalonador atribui as fatias de tempo para todos os threads da prioridade seguinte. Se um thread com prioridade maior estiver pronto para ser executado, o Escalonador deixa de executar o thread de prioridade menor (sem terminar sua fatia de tempo) e atribui uma fatia de tempo para este thread de maior prioridade.

O valor que representa a prioridade dos threads não é atribuída diretamente pelo programador. A prioridade é gerada a partir do Priority Class e do Priority Level. A Priority Class é um valor atribuído a um processo, enquanto o Priority Level é atribuído a um thread. A tabela 4.1 mostra a relação entre os dois e a prioridade resultante.

Linux

O Kernel no Linux evolui constantemente. Novas versões são periodicamente liberadas para os usuários. Isto envolve mudanças no Escalonador. Não convém discutir suas versões antigas, portanto será considerada a versão 2.6 do Kernel, onde as últimas grandes mudanças do Escalonador ocorreram [52][32].

Process priority class	Thread priority level	Base priority
IDLE	IDLE	1
	LOWEST	2
	BELOW_NORMAL	3
	NORMAL	4
	ABOVE_NORMAL	5
	HIGHEST	6
	TIME_CRITICAL	15
BELOW_NORMAL	IDLE	1
	LOWEST	4
	BELOW_NORMAL	5
	NORMAL	6
	ABOVE_NORMAL	7
	HIGHEST	8
	TIME_CRITICAL	15
NORMAL	IDLE	1
	LOWEST	6
	BELOW_NORMAL	7
	NORMAL	8
	ABOVE_NORMAL	9
	HIGHEST	10
	TIME_CRITICAL	15
ABOVE_NORMAL	IDLE	1
	LOWEST	8
	BELOW_NORMAL	9
	NORMAL	10
	ABOVE_NORMAL	11
	HIGHEST	12
	TIME_CRITICAL	15
HIGH	IDLE	1
	LOWEST	11
	BELOW_NORMAL	12
	NORMAL	13
	ABOVE_NORMAL	14
	HIGHEST	15
	TIME_CRITICAL	15
REALTIME	IDLE	16
	LOWEST	22
	BELOW_NORMAL	23
	NORMAL	24
	ABOVE_NORMAL	25
	HIGHEST	26
	TIME_CRITICAL	31

Tabela 4.1: Relação entre Priority Level e Priority Class na prioridade de threads no Windows. Tabela obtida de [4].

Uma observação importante é que o Escalonador é capaz de gerenciar threads de processos, ou seja, é possível ter threads de um mesmo processo com configurações de escalonamento diferentes.

Uma das principais mudanças foi a introdução de preempção para o Kernel, ou seja, códigos do Kernel sendo executado passaram a ser interrompíveis, permitindo que threads com prioridade maior sejam executados.

Outra mudança importante ocorreu no algoritmo de atribuição da fatia de tempo para a execução de cada thread. Antes o tempo era calculado proporcionalmente ao o número de threads sendo executados. Quanto mais threads, maior o tempo gasto na troca de threads/processos . Depois o tempo passou a ser fixo, tornando-o mais eficiente.

Estas mudanças tornaram o Linux atraente para aplicações do tipo *soft-realtime*. O Escalonador possui Policys (ou políticas) [14][52][53] para escolher os processos/threads a serem executados. Elas estão divididas em duas classes, a Real Time Schedule e a Completely Fair Schedule [52]. No final, independente da política, todo thread possuirá uma prioridade. O Escalonador possui uma lista de threads prontos para execução em cada prioridade.

A política Real Time Schedule (introduzida em versões anteriores) é dividida em SCHED_RR (Round Robin) e SCHED_FIFO (First In First Out). São estes que dão o aspecto *soft-realtime* para o Linux. O programador pode escolher a prioridade estática destes threads, entre 1 e 99, sendo 1 a maior prioridade. O Escalonador irá executar os threads da lista com a maior prioridade (seguindo a ordem da lista) até que um thread com maior prioridade esteja pronto para execução, mudando a lista a ser executada. O patch PREEMPT_RT muda o kernel do Linux tornando-o hard real time. Esta mudança não muda em nada as opções de escalonamento, apenas muda o comportamento do SO nos threads de tempo real SCHED_RR e SCHED_FIFO.

Dentro de uma prioridade, se o thread for SCHED_FIFO, o thread será executado até que ele se suspenda. No SCHED_RR o thread possui uma fatia de tempo limite, que quando expira faz o thread ir para o final da fila de sua prioridade.

É importante ressaltar que threads só podem ter a política de Real Time se o processo possuir privilégios administrativos. Este mecanismo é o que permite dar prioridade à execução do sistema operacional sobre a execução de processos comuns, evitando que os últimos travem o sistema.

As políticas da classe Completely Fair Schedule (CFS) são divididas em SCHED_NORMAL, SCHED_BATCH e SCHED_IDLE. Todo processo é criado com SCHED_NORMAL. A prioridade dos threads neste caso é dinâmica, e apenas no SCHED_NORMAL o usuário pode atribuir um impulso na prioridade (chamado de "nice value"). Nos 3 casos, quando um thread fica muito tempo sem executar, sua pri-

oridade aumenta até ser executado. O SCHED_BATCH acaba ficando com uma prioridade um pouco menor do que com a SCHED_NORMAL, e é destinada a processos de execução intensa, mas que possuem pouca interação com o sistema. O SCHED_IDLE possui uma prioridade ainda menor e é destinada a processos de background.

Microcontroladores

Existem Sistemas Operacionais para microcontroladores. Suas capacidades são sempre limitadas como é o caso do FreeRTOS. Em geral implementam um micro kernel capaz de gerenciar tarefas (não necessariamente threads) e objetos de sincronização. Para o MOS ser implementado é preciso um acesso aos recursos básicos de uma forma padronizada. O FreeRTOS por exemplo não é capaz de prover o que o MOS precisa. Para evitar estes problema de compatibilidade o MOS propõe o desenvolvimento de um pequeno Kernel para cada dispositivo em que for implementado, consequentemente existe uma grande liberdade de como o Escalonador do Kernel pode ser desenvolvido. Vale lembrar que este capítulo trata de como o MOS pode ser implementado e não como será de fato, ou seja, o MOS poderia ser implementado em cima de outras SOs de microcontroladores ao invés do Kernel proposto.

O relatório [54] é a inspiração para a construção de um Kernel para microcontroladores AVR. O relatório explica como o Escalonador do FreeRTOS é implementado em microcontroladores AVR usando interrupções disponíveis no microcontrolador. Nele, um algoritmo round robin é implementado, mas sem prioridade.

Esta dissertação propõe um Kernel do MOS para ATmega2560 baseado em [54]. Nesta proposta, o Kernel tem um Escalonador com duas opções, uma Round Robin e outra FIFO. No Round Robin os threads são executados em uma ordem e têm a sua fatia de tempo de execução. A única diferença entre a Round Robin e a FIFO é que a FIFO não possui fatia de tempo (o próprio thread deve se bloquear).

Note que as opções Round Robin e FIFO do escalonador não são como no Linux. No Linux são políticas para os threads, sendo possível ter um thread SCHED_RR e outro SCHED_FIFO a serem executados no computador. Na proposta do Escalonador do ATmega2560 todos os threads são tratados com Round Robin ou todos são tratados com FIFO, então não são funcionam como as políticas de escalonamento. O Kernel permite que as opções de escalonamento sejam trocadas enquanto o MOSDevice está funcionando (qualquer thread pode indicar qual algoritmo de escalonamento deverá rodar no kernel).

Classes

Após o estudo dos Escalonadores dos MOSDeviceTypes, é possível criar classes para representar os Escalonadores. O objetivo não é criar as classes que implementam a criação/destruição de threads, ou a mudança de suas prioridades, e sim classes para

guardar as informações do thread ou do escalonador. As classes são usadas dentro do principal grupo de classes, o GModuleOS.

Como classes estão relacionadas à implementação, é preciso comentar sobre as bibliotecas que podem ser usadas. No Windows, a biblioteca padrão Kernel32.lib possui funções para criação, destruição e configuração de threads e processos. O Linux possui uma implementação de threads que segue o padrão POSIX e age de forma muito similar. A estrutura de classes e funções nas duas bibliotecas são muito diferentes, o que atrapalha a implementação do MOS em MOSDeviceTypes.

A biblioteca multiplataforma Boost para C++ equivale a um grande conjunto de biblotecas menores. A Boost Thread cria uma abstração das bibliotecas de thread de sistemas operationais. Isto facilita a criação de programas multiplataforma. A biblioteca Boost Thread usa as bibliotecas nativas do Windows e Linux internamente.

As implementações do MOS para Windows e Linux são muito parecidas. O que as tornam diferentes são as bibliotecas relacionadas às threads e as bibliotecas de acesso às redes. Para facilitar, os MOSProcess devem ser implementados com Boost. É importante ressaltar que, pelo fato do Boost Thread criar uma abstração, ele não é capaz de configurar os threads como as bibliotecas nativas, já que as configurações são específicas de cada Sistema Operacional. Logo, uma parte pode ser implementada com Boost Thread e outra com as bibliotecas nativas.

O GScheduler e o GThreads, apesar de relacionados, representam funcionalidades diferentes. O GScheduler está relacionado às opções do Escalonador, enquanto o GThreads às opções dos threads. É preciso analisar cada MOSDeviceType para saber o que cada grupo terá.

Os MOSProcess são processos em seus respectivos Sistemas Operacionais. As opções básicas do Escalonador no Windows são a PriorityLevel e PriorityClass. A PriorityLevel é uma característica do thread, enquanto o PriorityClass é uma caracterítica do processo. Ao mudar o ProrityClass de um MOSProcess, todos os threads do MOS serão influenciados, portanto esta característica pertencerá ao GScheduler. O PriorityLevel pertencerá ao GThreads.

O Linux não tem algo como o PriorityLevel no Windows. Todos os threads têm características únicas, ou seja, não estão atrelados ao processo à que pertencem (MOSProcess). A Policy e a prioridade do thread (usada apenas nos SCHED_RR e SCHED_FIFO) pertencem ao GThreads.

O Escalonador proposto para o ATmega2560 tem duas opções que pertencem ao GScheduler, as opções de Round Robin ou FIFO. No GThreads estão as prioridades do thread para Round Robin e para FIFO. O Escalonador deve saber qual thread está sendo executado no momento, e todos os threads ativos devem saber quem é o próximo na lista (RR ou FIFO). O parâmetro de configuração do Escalonador no RR é a fatia de tempo máxima que os threads podem executar antes de trocar de

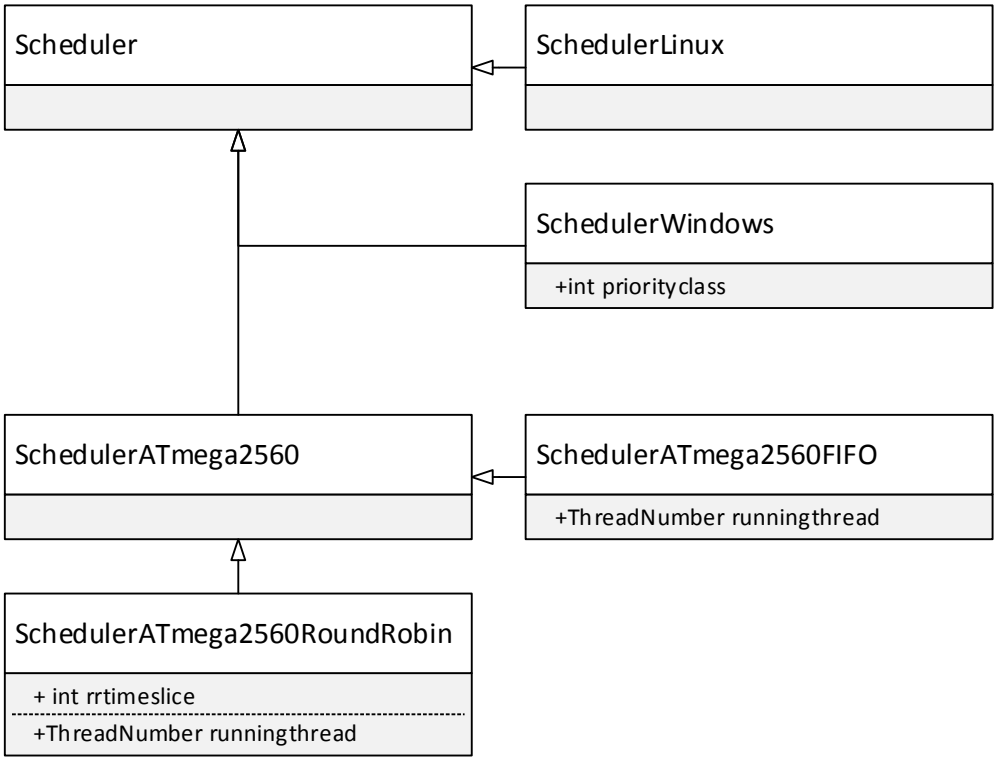

Figura 4.1: Classes do GScheduler.

thread, enquanto a FIFO não tem nenhum parâmetro.

Existem muitas outras características de threads que podem ser levadas em consideração no GScheduler e GThreads. No entanto, o objetivo desta dissertação não é implementar o MOS por completo, e sim indicar os elementos chave na implementação.

Com isto, propõe-se as seguintes classes para o GScheduler, como ilustrado na figura 4.1. A classe *Scheduler* é usada apenas pela lógica de programação a objetos. A *priorityclass* no Windows determina a priority class do processo (o MOS). A *SchedulerLinux* fica vazia por não precisar de nenhuma configuração. A *SchedulerMicro* é como a *Scheduler*. A *SchedulerATmega2560RoundRobin* tem uma variável que indica a fatia de tempo máxima de execução de threads. A *SchedulerATmega2560RoundRobin* e *SchedulerATmega2560FIFO* tem a runningthread para saber qual thread está sendo executada atualmente (as classes do Windows e Linux não precisam disto pois usam o Escalonador do SO). Os threads no MOS são identificados por um número, que é representado pelo tipo *ThreadNumber* que pode variar com o MOSDeviceType (*unsigned int*, *unsigned char* etc.).

Veremos agora como o GThreads deve ser construído. Na classe do Windows podemos indicar o Priority Level e do Linux as Policy e Priority. No MOSMicro não precisamos de nenhuma variável de configuração porque os algoritmos a serem usados são bem simples. Considerando o escalonamento tem-se classes muito parecidas com

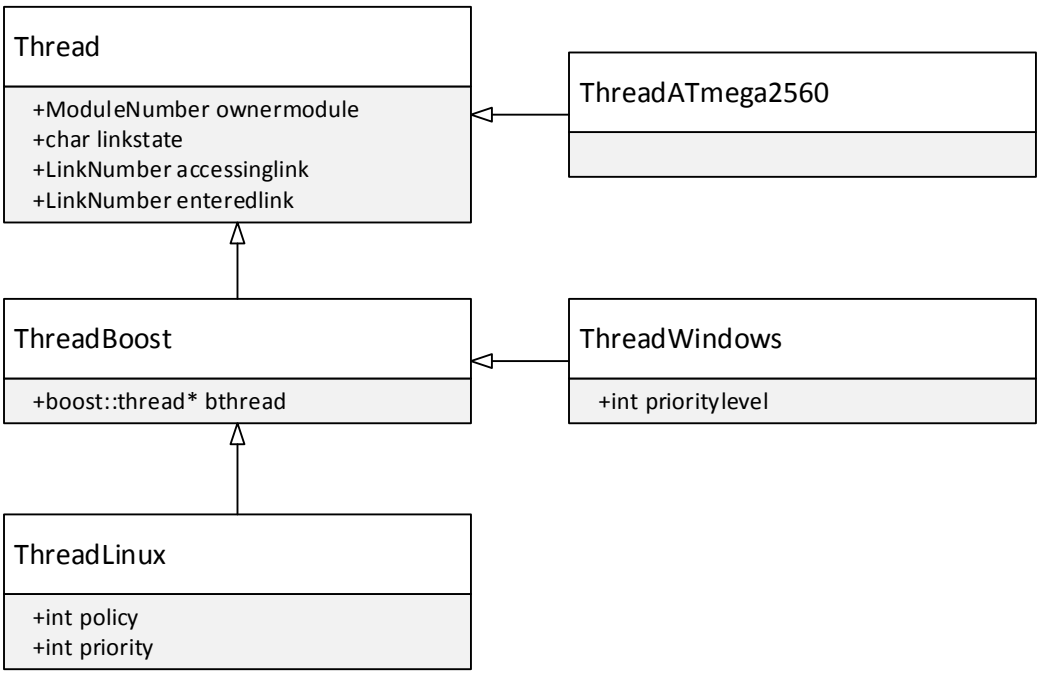

Figura 4.2: Classes do GThreads.

as do GScheduler, mas os thread envolvem outras coisas.

Threads pertencem às ModIns que os criam, mas não são exclusivos das ModIns. Eles podem navegar pelas ModIns graças aos MOSLinks. Existem duas formas de uso no MOSLink, o acesso e a entrada. Posteriormente as classes do GLink e GModule serão discutidas, mas o que é preciso entender aqui é que os MOSLinks e ModIns são identificados no C++, assim como o próprio thread (*ThreadNumber*). Todo thread deve saber qual ModIn o criou, se está acessando ou entrou uma ModIn, e quais são estas ModIns.

Threads não são explicitamente criados pelo programador na MOSLanguage. Como não há alocação dinâmica na MOSLanguage os threads são referenciados de forma estática nas classes/ModClasses. O que o programador pode fazer é decidir quando o thread será iniciado ou reiniciado (se já acabou sua execução). Esta limitação também foi imposta para que uma ModIn tenha todos os recursos alocados antes de sua execução, e com isto não dependa da falta de recursos (comum em MOSMicros).

A figura 4.2 mostra as classes do GThreads.

O MOS possui uma classe principal, a *ModuleOS*, que funciona como casca para os Modules e representa o System na MOSLanguage. A classe *ModuleOS* também é derivada seguindo os MOSDeviceType. As figuras 4.3 e 4.4 mostram como o *ModuleOS* se relaciona com o Escalonador e os threads. O *ThreadNumber* nada mais é do que o índice do vetor de threads criado nas classes derivadas de *ModuleOS*. No *ThreadBoost* temos uma variável de mutex do boost que é usado para gerar um

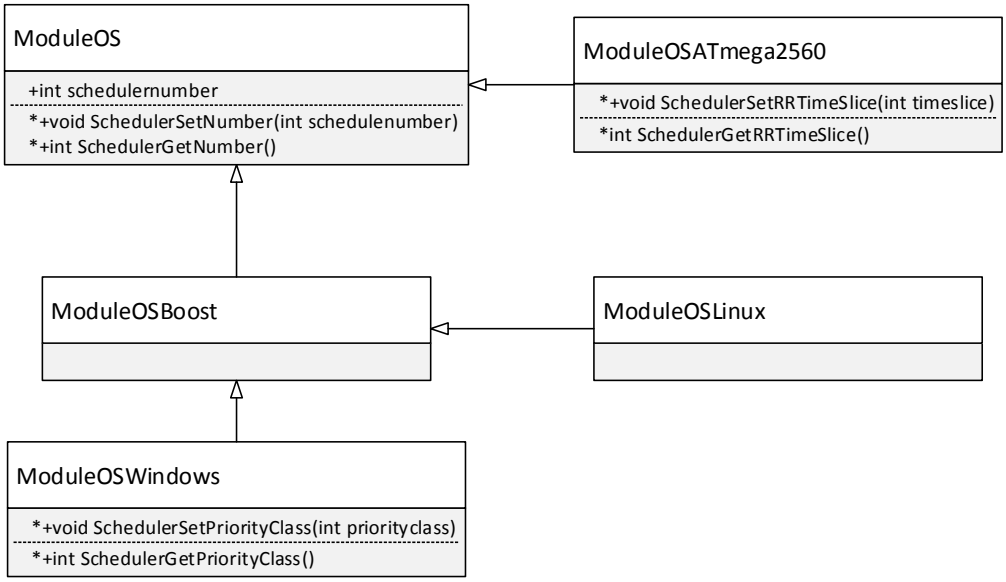

Figura 4.3: Classes do GModuleOS em relação ao GScheduler.

thread de verdade. A *StartThread()* é invocada na MOSLanguage para iniciar a rotina de um thread.

As variáveis relacionadas às configurações do Escalonador e do thread são específicas de MOSDeviceTypes, portanto estarão acessíveis na MOSLanguage apenas nas BModClasses.

4.1.2 GSyncObjects

A princípio os objetos de sincronização deveriam ser implementados em todos os MOSDeviceTypes, de forma que o uso dos recursos na MOSLanguage não seria vinculado a MOSDeviceTypes. Não convém explicar como todos funcionariam ou quais de fato seriam implementados. A lista de possibilidades inclui Mutex, Conditional Variables, Semaphore, Monitor, Read/Write Mutex, Barriers etc.

É preciso lembrar que alguns MOSDeviceTypes possuem quantidade fixa destes objetos enquanto outros podem alocar novos de forma dinâmica. A limitação imposta na MOSLanguage é que estes objetos não podem ser criados dentro de uma função, só podem ser criados dentro de classes/ModClasses. Ou seja, quando uma ModClass é definida é possível saber todos os objetos de sincronização que serão usados. Uma ModIn não pode ser criada sem alocar estes recursos. Este mecanismo foi criado para a MOSLanguage nos MOSMicros. A criação/destruição dos objetos é um dos temas.

O objetos estão intimamente relacionado ao Escalonador pois influenciam no fluxo de execução de threads (bloquear/desbloquear). Isto é especialmente importante nos MOSMicros pois o Escalonador destes é desenvolvido pelo próprio MOS.

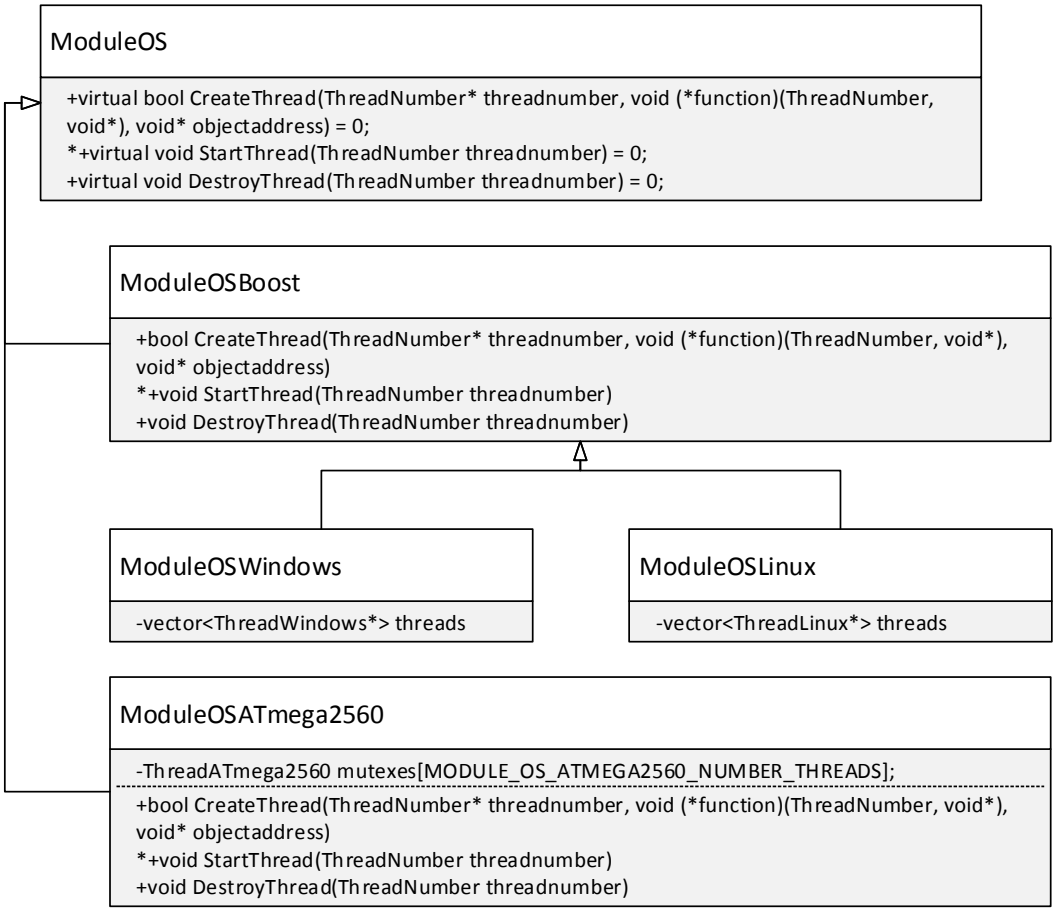

Figura 4.4: Classes do GModuleOS em relação ao GThreads.

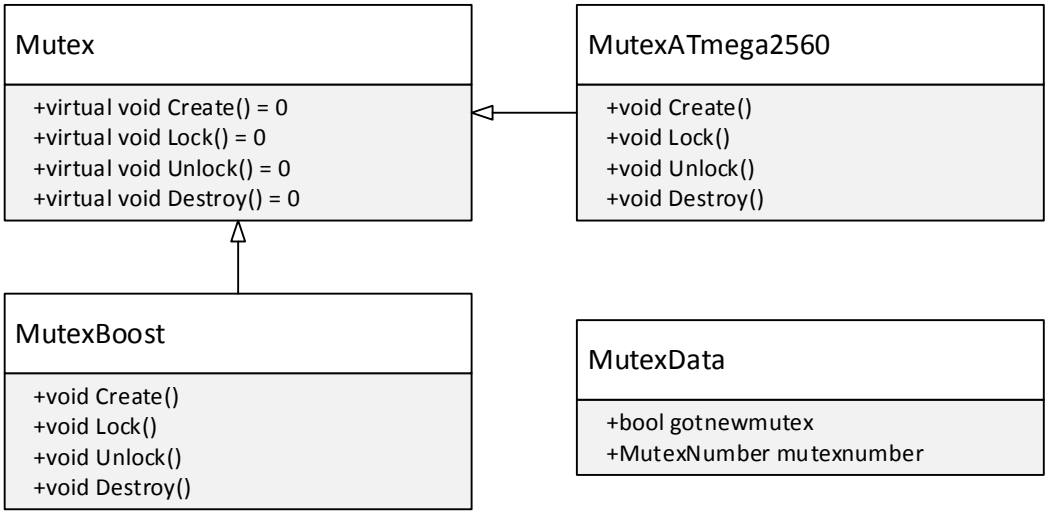

Figura 4.5: Classes do GSyncObjects.

Foquemos em apenas um destes objetos, o Mutex. Este objeto está relacionado ao lock e unlock das regiões críticas, portanto o núcleo das classes de Mutex são as funções *Lock()* e *Unlock()*. Além disto é preciso cuidar da criação e destruição destes objetos.

Assim como nos threads, a biblioteca boost será usada no MOSProcesses (Windows e Linux), portanto os lock, unlock, criação e destruição são triviais. A implementação nos MOSMicros é mais complicada. Novamente não convém definir como os Mutexes seriam implementados, basta explicar alguns pontos chave. As funções *Lock()* e *Unlock()* continuam sendo as necessárias. O número de Mutexes que o MOS terá é definido na compilação do MOS, então a criação e destruição consiste em encontrar um Mutex que não esteja sendo usado ou liberar um Mutex para ser usado novamente por outro.

Com isto a figura 4.5 ilustra as classes do Mutex no MOS. Todos possuem as mesmas funções, mas em cada MOSDeviceType a implementação será diferente. A classe *MutexData* é usada para indicar se foi possível criar um novo Mutex. A *MutexBoost()* usa um Mutex do Boost. O mesmo não pode ser definido nos MOSMicros porque está muito relacionado à implementação do Escalonador do MOSMicro (não será tratado aqui).

O *ModuleOS* deve ter funções dos Mutexes para que o programador utilize Mutexes pela MOSLanguage. O programador não enxerga os identificadores mas a MOSLanguage chama as funções do *ModuleOS* indicando o identificador do Mutex (valor fica na ModIn). As classes do GModuleOS ficam como na figura 4.6.

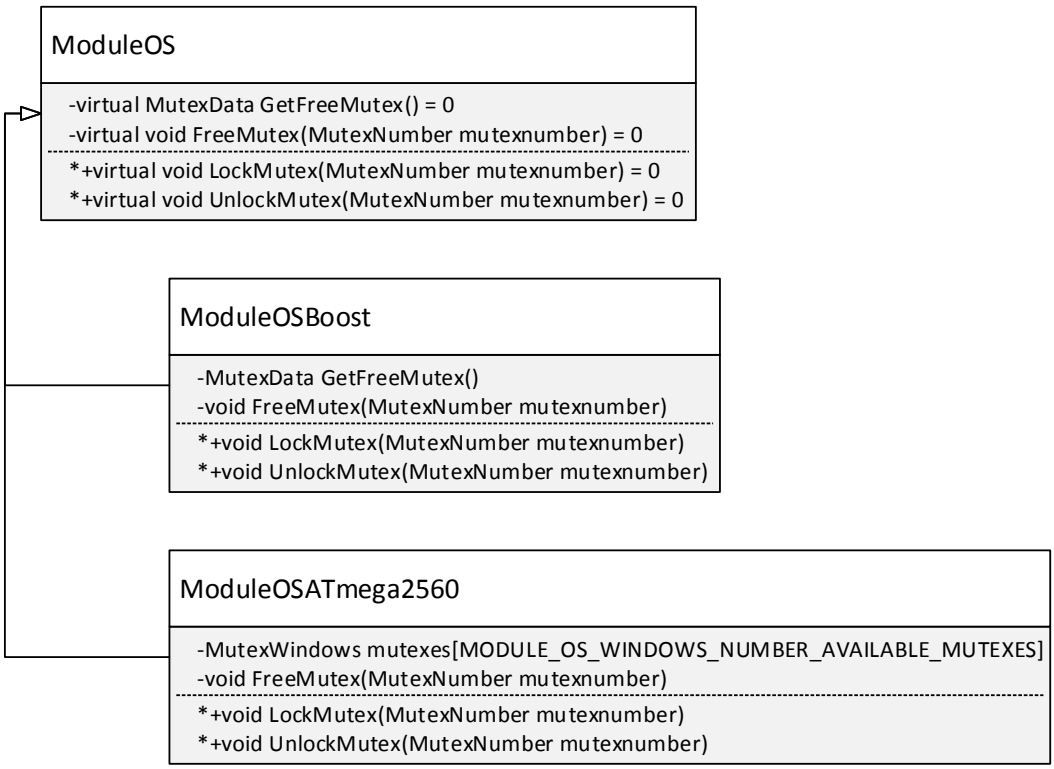

Figura 4.6: Classes do GModuleOS em relação ao GSyncObjects.

4.1.3 GLink

Os MOSLinks criam uma conexão entre um LinkOwner e um LinkEnd. Apenas um thread do LinkOwner pode acessar o LinkEnd por vez, mas não existe o número máximo de threads que podem entrar no LinkEnd. Pela segurança na hora de destruir uma ModIn é preciso contar o número de threads que entraram e retornam de um LinkEnd (destruição é segura apenas se 0 threads estão no LinkEnd), ou seja, O MOSLink precisa de um contador.

Os MOSLinks são alocados em um vetor e são identificados pelo índice do vetor. É possível que não haja MOSLink disponível no MOSDevice. Neste caso a estrutura do MOSLink é muito parecida com a do Mutex (existe um *LinkNumber* e um *LinkData*.

As ModClasses definem o número de LinkSlots que terão quando forem instanciados. O LinkSlot na ModIn indica se o LinkSlot está livre/ocupado e quem é o LinkOwner quando ocupado.

A figura 4.7 mostra as classes do GLink. As classes do GLink apenas contém dados do MOSLink.

As funções que tratam o MOSLink estão todas no GModuleOS. Além do MOSLinkDirect (entrada e retorno), os LinkOwner e LinkEnd podem transferir mensagens de um para o outro como MOSLinkMP. O GModuleOS deve definir e criar o MPThread que irá entrar em MOSLinks ou criar novos MOSLinks.

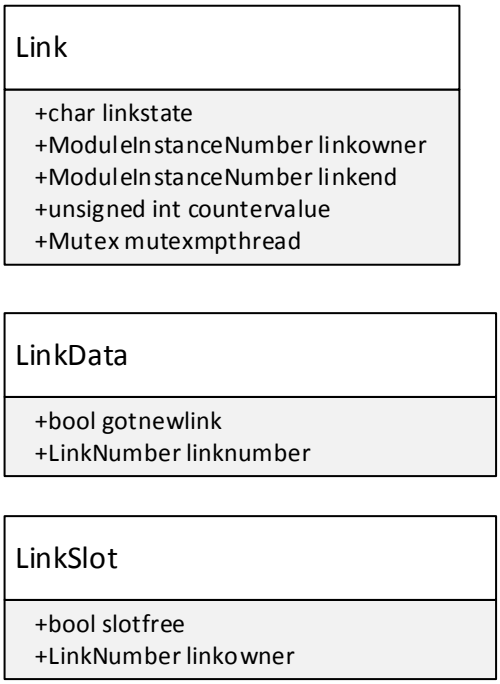

Figura 4.7: Classes do GLink.

As funções da GModuleOS sobre o GLink estão ilustradas na figura 4.8. Note que existem funções acessíveis pela MOSLanguage para entrar, retornar, acessar, enviar mensagens para os Owners ou para o End de um MOSLink. O retorno do MOSLinkDirect não não é usado pelo programador mas a MOSLanguage compilada usa ela nas funções públicas para iniciar o retorno para o LinkOwner.

Para criar um MOSLink o LinkOwner deve primeiro adquirir um *Link* com *GetFreeLink()*, para então preenchê-lo com seus dados. Sabendo o *LinkNumber* do LinkEnd, o LinkOwner deve chamar a função *LinkModules()* indicando o *LinkNumber* e os dados básicos de um Module. Os dados são usados para verificar se o LinkNumber realmente é quem o LinkOwner deseja se conectar. O programador na MOSLanguage não preenche os dados do LinkEnd nessa função (o código de máquina da MOSLanguage preenche os campos).

4.1.4 GModule

Quando uma ModClass é compilada algumas informações básicas são salvas no setor 2 do CMod. Entre eles está o nome do MOSProject, nome da ModClass, número da MVersion, número da MPubVersion, posição da função de inicialização no setor 1, o tamanho em bytes da ModClass, o número de Mutexes, o número de threads, o número de LinkSlots etc. A última informação do setor 2 é a posição destes objetos enumerados na instância da ModClass. Estes objetos ficam armazenados em

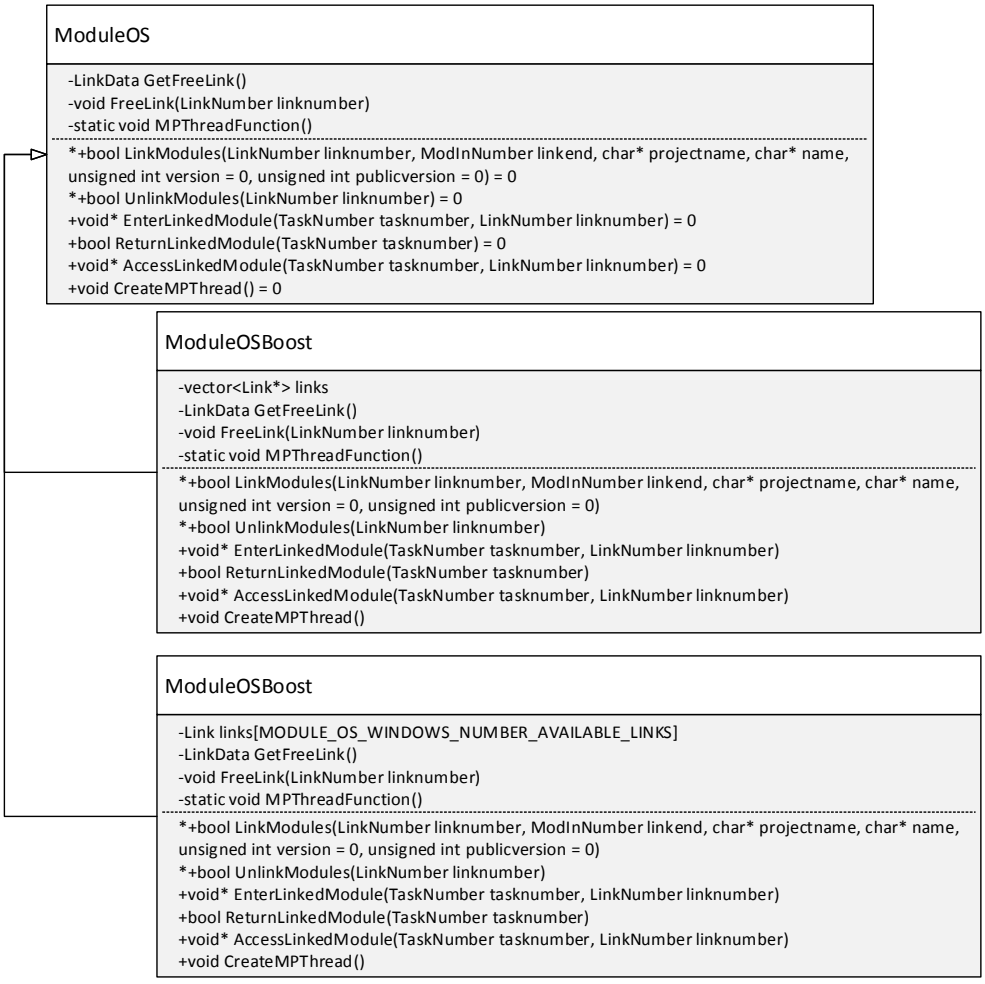

Figura 4.8: Classes do GModuleOS em relação ao GLink.

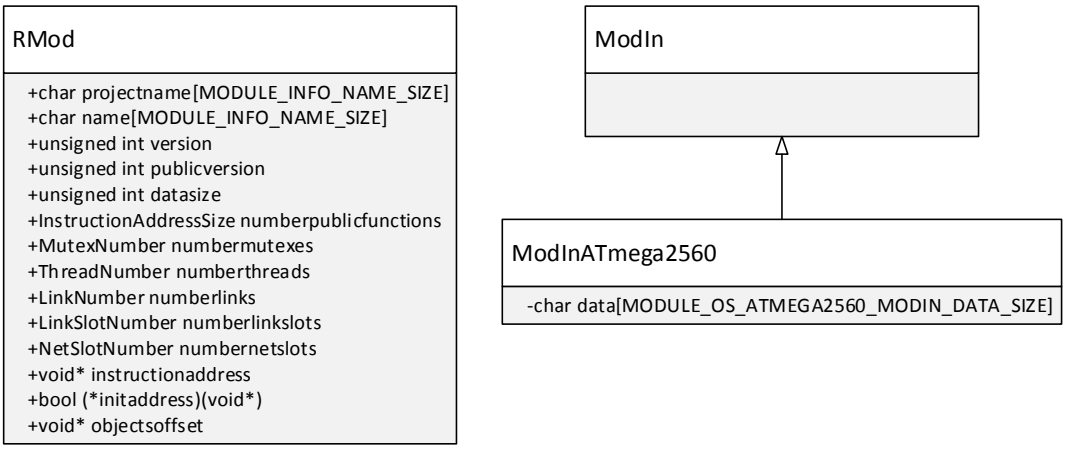

Figura 4.9: Classes do GModule.

sequência. Quando tenta-se criar a instância é verifica-se a disponibilidade dos Mutex, threads etc. Se houverem, a ModIn é criada e ganha uma área na memória que conterá as variáveis da ModClass. Os objetos enumerados são então preenchidos de acordo com a última informação, ou seja, a ModIn inicia podendo usar os Mutexes, threads etc.

Não iremos tratar nesta arquitetura classes envolvendo a transformação de um CMod em um RMod sobretudo porque isto está muito relacionado ao MOSBuilder. Como foi dito anteriormente o MOSBuilder não é o foco da dissertação pois sua implementação é obscura.

A figura 4.9 contém as classes do GModule. A *RMod* contém todas as informações necessárias para que uma instância seja criada, inclusive contém o ponteiro do início das instruções de máquina. A *ModIn* é uma classe bem simples. Nela deve-se associar a região na RAM que será usado para o objeto. Quando é possível usar alocação dinâmica usa-se o *new*, mas como o ATmega2560 tem problemas de espaço uma solução melhor é derivar a classe e alocar este vetor. O *ModuleOS* conterá um array da derivação, portanto o tamanho do *data* é escolhido na compilação do MOSDevice. Vale lembrar que se o número de bytes de uma ModClass for maior do que o máximo determinado no MOSDevice, a ModIn não pode ser criada.

A interação do GModule no GModuleOS é maior do que nos outros grupos como mostra o número de funções no GModuleOS (figura 4.10). As primeiras funções estão relacionadas a criação e destruição de RMods e ModIns. Criar um RMod basicamente preenche um *RMod* com informações de um CMod (supõe-se que o CMod já foi carregado no MOSDevice, e as instruções de máquina já estão na memória de programa). A criação da ModIn verifica se os recursos (indicados no RMod) estão disponíveis para então configurar a *ModIn*.

As funções *New* e *Delete* dos *RMod* e *ModIn* são funções privadas então não estão acessíveis na MOSLanguage, são usadas apenas pelo MOS. As informações básicas

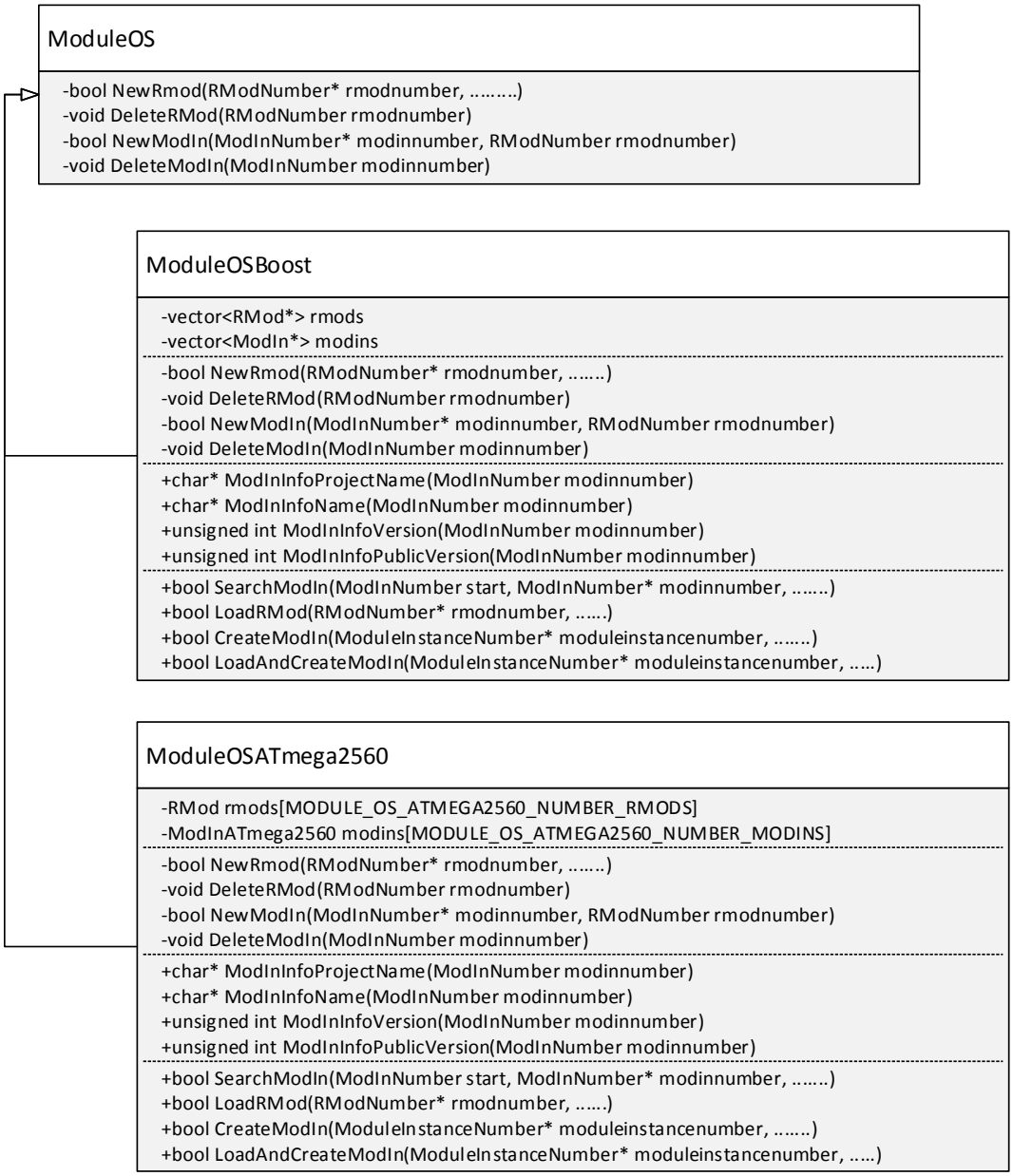

Figura 4.10: Classes do GModuleOS em relação ao GModule.

dos Modules (nomes do MOSProject etc.) são visíveis para todos as ModIns. O programador pode usar as informações básicas para buscar uma ModIn que está executando, pedir que o RMod seja carregado na memória (poderia pedir para que o outro MOSDevice gerar o CMod, transferir e gerar o RMod) e pedir para que uma ModIn seja criada. Tendo a referência de uma ModIn compatível, o programador pode criar um Link entre as ModIns com comentado no do GLink.

4.1.5 GNetwork

Quando uma ModClass é definida o programador pode definir objetos especiais para lidar com as Nets, são os NetSlots. Eles são vinculados a um Protocol e um NetType.

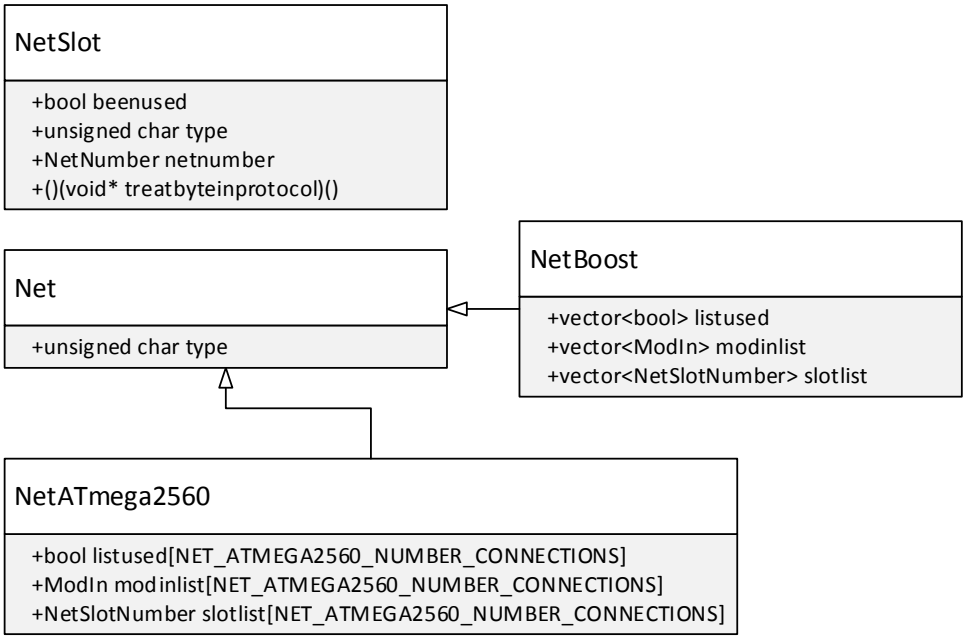

Figura 4.11: Classes do GNetwork.

Os NetSlots ficam dentro das ModIns, ou melhor, dentro do espaço reservado para as variáveis da ModClass. O MOS sabe onde os NetSlots estão dentro deste espaço. Os NetSlots não iniciam associados a Nets, mas só podem ser usados de verdade quando associados.

A Net é um objeto que representa qualquer tipo de rede. Ela deve ser conectada com ModIns e seus NetSlots.

A figura 4.11 ilustra as classes da GNetwork. Podemos ver que a *NetSlot* sabe qual é seu NetType. Quando associada saberá a identificação da Net. Ela também guarda a função de tratamento do Protocol associado. A *Net* também tem uma identificação do NetType. A Net pode se associar a uma lista de ModIns e seus NetSlot. Note que a *NetSlot* usa o *NetNumber* para encontrar a Net no MOS, e a *Net* usa o *NetSlotNumber* para encontrar o NetSlot dentro da ModIn.

O System possui algumas funções relacionadas ao uso das Nets. Funções permitem encontrar a próxima Net de um NetType a partir de um ponto do vetor de Nets, e os NetSlots podem ser adicionados/retirados de uma Net. No entanto apenas nas BModClasses é possível criar Nets (usa-se as configurações dos MOSDeviceTypes).

A figura 4.12 ilustra as classes da GModuleOS relacionadas GNetwork. Note que existem funções de Thread de NetTypes. Quando as Nets forem criadas um thread será executado para realizar o tratamento de mensagens, o NetThread, e o thread irá executar uma destas funções para lidar com o rede no dispositivo.

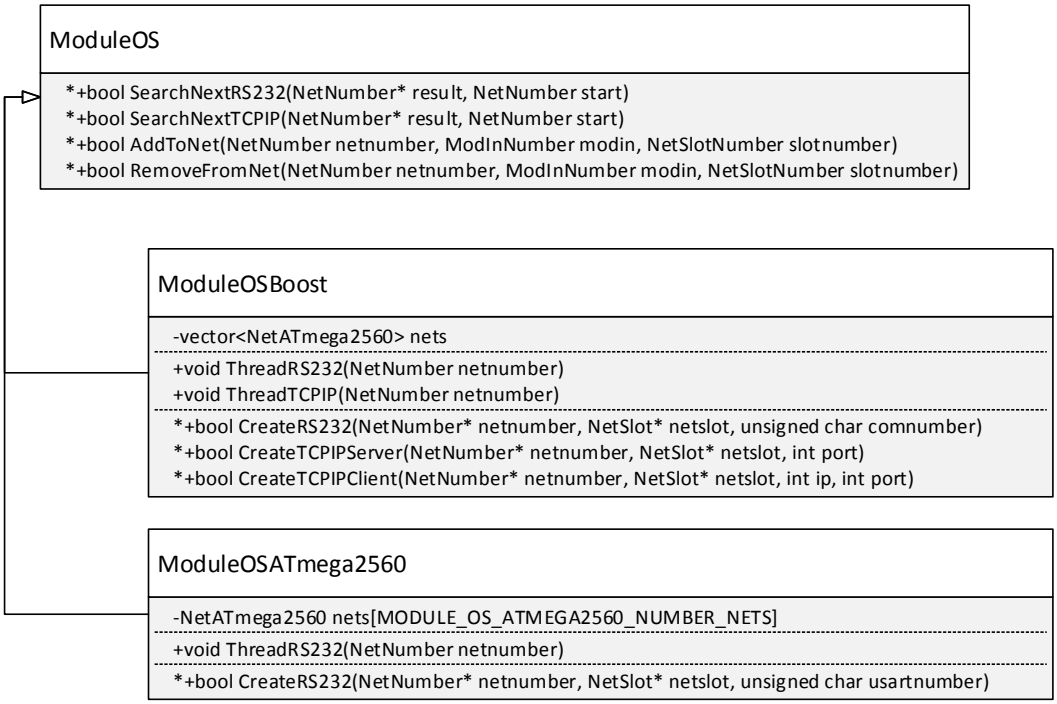

Figura 4.12: Classes do GModuleOS em relação ao GNetwork.

4.1.6 GModuleOS

Muito já foi dito sobre o GModuleOS nos outros grupos. Aqui veremos o que falta. Considere a figura 4.13.

A função *Start()* inicializa o MOS. Um arquivo de configuração (ou pedaço da memória EEPROM no ATmega2560) pode ser aberto para indicar as instruções. Por exemplo, quando um MOSDevice inicia ele configura uma Net e faz com que esta seja usada na MOSNetwork. Outra exemplo é a iniciar uma ModIn. No caso dos MOSMicros esta é a função que inicia o Escalonador do MOS.

A temporização de threads é um requisito básico de todo SO. A temporização consiste em poder adquirir o tempo atual e botar um thread para dormir por um certo tempo. Os MOSProcesses podem fazer isto com facilidade pelos SO já terem bibliotecas nativas que lidam com isso. O problema está nas aplicação de tempo real (o MOS precisaria ser implementado em um Sistema Operacional de Tempo Real).

Alguns microcontroladores possuem Timers para tratar com o tempo. O próprio MOS usa um destes Timers para implementar o Escalonador. A implementação da temporização no MOS é feita com o mesmo Timer do Escalonador.

Repare que a *ModuleOSATmega2560* possui uma série de funções especiais relacionadas ao próprio dispositivo. Estas funções estão disponíveis apenas nas BModClass vinculada ao ATmega2560. Isto permite o programador acessar os recursos exclusivos do ATmega2560 com a MOSLanguage como as portas digitais e analógicas.

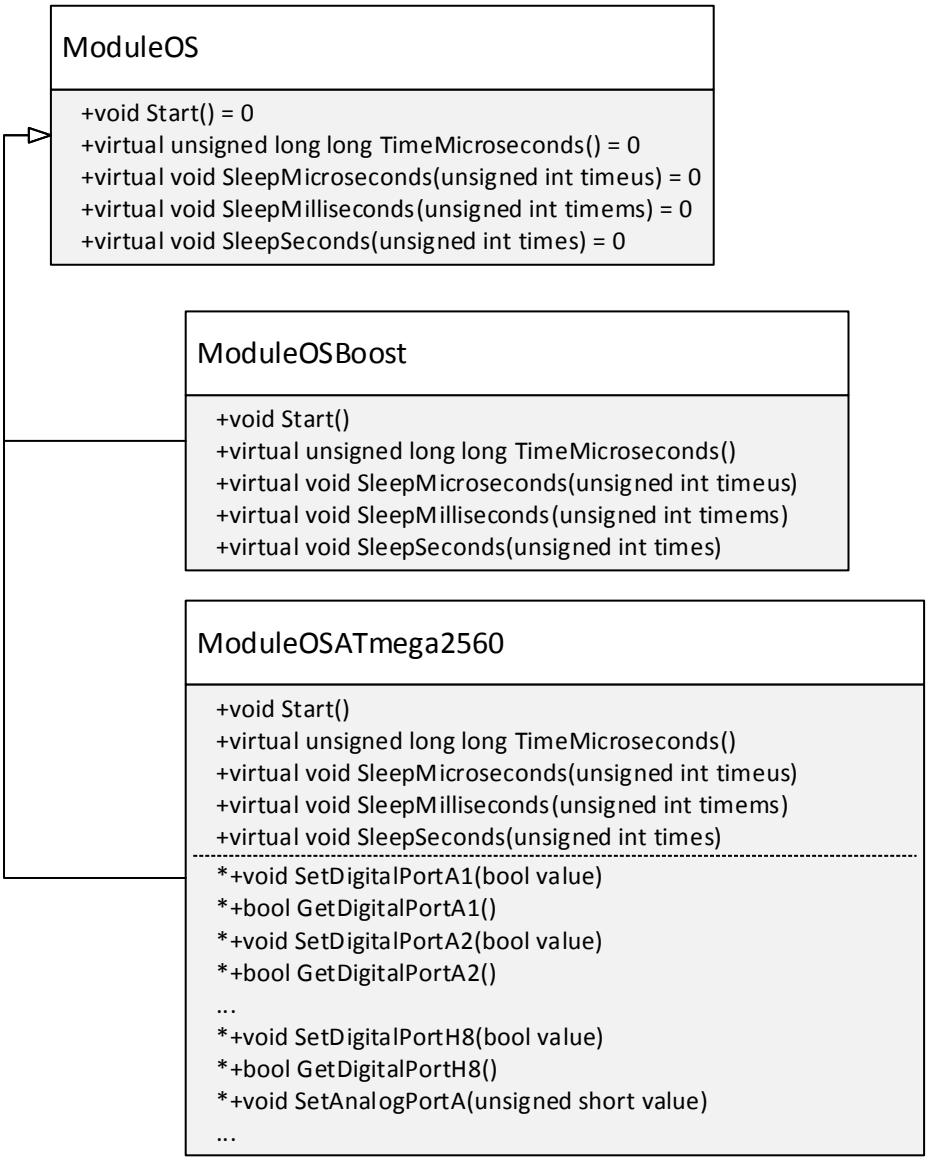

Figura 4.13: Classes do GModuleOS.

4.2 Implementação do MOS para Sistemas Operacionais

A implementação do MOS para Sistemas Operacionais é muito mais direta do que para microcontroladores pelos SOs criarem a abstração do Kernel (a implementação das classes é mais simples). Ainda assim existem aspectos a considerar sobre os MOSProcesses lidam com os CMods e RMods (não estão ligados ao acesso do Kernel).

4.2.1 Memória

Este item apresentará a divisão de Memória de um processo no Windows para então comentar como o MOS lida com isto.

Ao invés de processos em Sistemas Operacionais modernos utilizarem a memória física diretamente para armazenar/ler dados eles usam a memória virtual. A memória virtual é uma técnica de gerenciamento de memória que é implementada com hardware e software que mapeia endereços virtuais (usados pelos programas) em endereços físicos da memória do computador. A memória do computador consiste na tradicional RAM (Random Access Memory) e outros dispositivos de armazenamento como o HD (mais lentos). Ou seja, a memória virtual permite que o computador use mais memória do que a física. Se um pedaço da memória está no HD e precisa ser acessado por um programa que estava dormindo, o pedaço é passada para a RAM. Isto não é feito byte a byte, e sim em blocos ou páginas de memória [14]. O hardware que traduz endereços virtuais em físicos é a MMU (Memory Management Unity). Um fator importante é o controle de permissão na memória virtual. Por exemplo, o controle impede programas de acessarem a memórias de outros programas.

Quando um processo é criado (Windows/Linux), um pedaço de memória virtual é alocado com um tamanho já definido pelo Sistema Operacional. A figura 4.14 representa a divisão dessa memória em segmentos [41][3].

O segmento Code (ou text) contém instruções de máquina do programa. O registrador Instruction Pointer (ou Ponteiro de Instruções) da arquitetura x86 indica as instruções que o programa deve executar, e aponta primariamente para o segmento Code. Este segmento é preenchido apenas quando o programa é carregado na memória, ou seja, é protegido contra escrita pelo próprio programa. A segurança é um padrão dos Sistemas Operacionais para evitar códigos maliciosos que pode ser implementado graças ao sistema de permissões da memória virtual. Para um programa executar novos códigos é preciso carregar uma biblioteca dinâmica. Um novo espaço é alocado na memória para conter as novas instruções de máquina.

O segmento Data contém constantes e variáveis globais. Estes possuem um

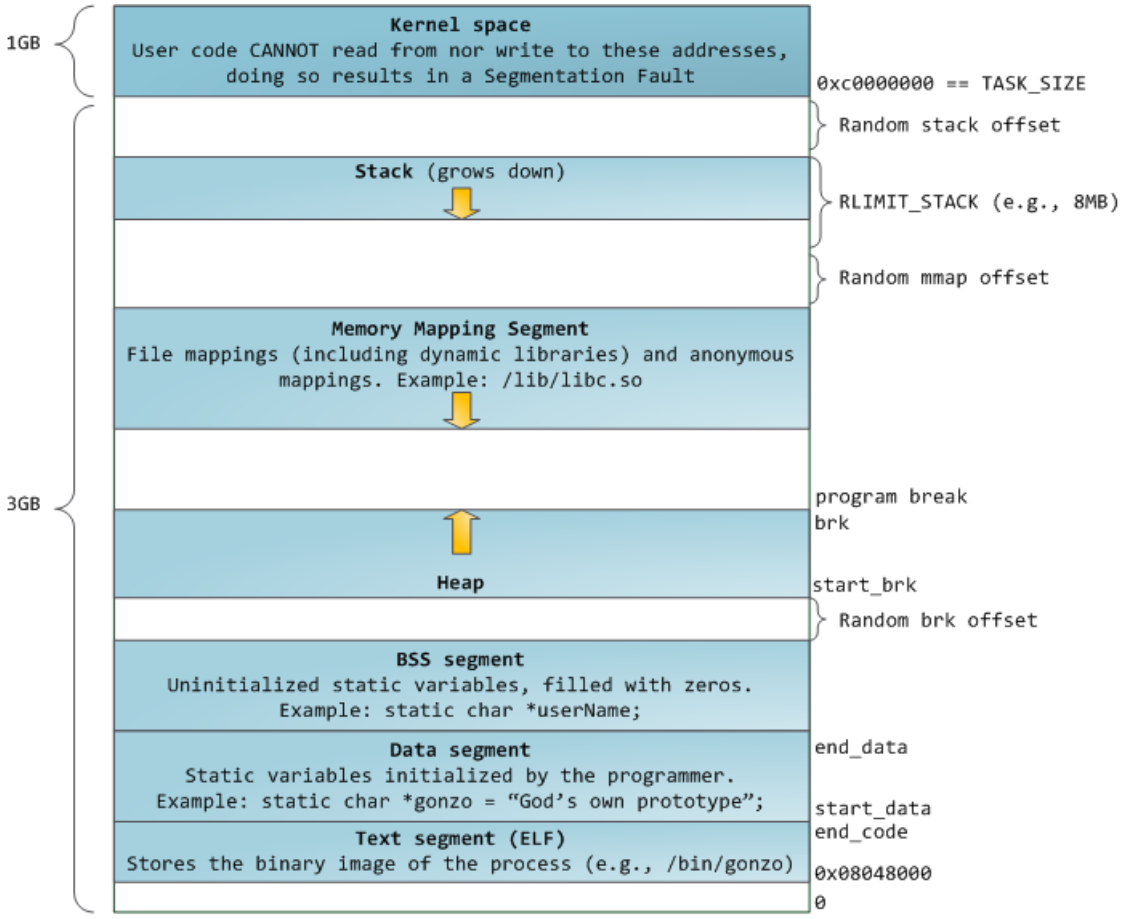

Figura 4.14: Segmentos de memória em um programa (figura obtida de [3]).

endereço fixo no programa. As instruções de máquina do Code usam endereços fixos para pegar/alterar valores no segmento Data. Na realidade o segmento Data é dividido em duas partes, dividindo o que é inicializado pelo programador do que deve não é inicializado e portanto deve iniciar com o valor 0. A segunda divisão é conhecida como BSS.

O Data e o Code possuem um tamanho fixo (definido quando o programa é compilado). O segmento Free Space é o espaço que sobra ao retirar o Data e o Code da memória do programa. O Free Space tem dois usos simultâneos, é usado como Heap e como Call Stack (Pilha de Chamada). Quando se cria uma variável de forma dinâmica (malloc ou new no C++) um espaço da Heap é alocado. Uma Pilha é um tipo abstrato de dado estrutura de dados baseado no princípio LIFO (Last In First Out). A Pilha de Chamada é a região da memória em que dados são adicionados e retirados no modelo LIFO. Para manter a coerência com os nomes em inglês, daqui em diante usaremos apenas Stack para referenciar a Pilha de Chamada/Call Stack.

A chamada de função (ou subrotina) envolve passar os argumentos da função e reservar um espaço para o retorno na Stack. Quando uma função cria uma variável local, outro pedaço é alocado na Stack, e quando retorna este pedaço é desalocado. Valores de registradores também são salvos na Stack antes da chamada da função e restaurados após o retorno para dar continuidade na função. A figura 4.15 mostra como a Stack é preenchida na chamada de funções (também conhecida como conhecida como calling convention) na arquitetura x86. Existe um procedimento de prólogo e um de epílogo na chamada. O retorno da função *int g(int x)* é salvo em registradores ao invés da Stack.

Quando Threads são criados pelos SOs uma nova área na Stack é alocada para o thread. O programa quando iniciado reserva sua memória na Stack (execução da função main() do programa), e os threads criados no processo alocam mais memória na Stack, fazendo com que a Stack cresça. Na criação de threads deve-se no mínimo indicar o endereço de uma função global que o thread executará, e é esta função que irá usar a Stack, inserindo e retirando dados. Portanto, a alocação não implica no thread estar usando todo pedaço de memória, e sim que o thread pode usar no máximo aquele pedaço como Stack.

Note na figura 4.14 que a Heap e a Stack começam em lados opostos do Free Space, e começam com tamanho zero. Na medida que se aloca espaço em cada um, a memória "cresce" em sentidos opostos. SOs modernos conseguem detectar a colisão das duas memórias e ou emitem um erro, ou impedem a alocação de novos trechos de memória.

Um núcleo de processamento executa apenas um thread. Os registradores do núcleo representam a execução do thread. Em certo momento, uma interrupção no SO inicia a troca de execução de threads. A troca deve guardar todos os valores

```
void f()
{
        int a = 10;
        int b = 16;
        b = g(a);
}

int g(int x)
{
        int c = 2*x;
        return c;
}
```

OBS: "*" é um ponteiro para uma instrução de máquina de b = g(a)

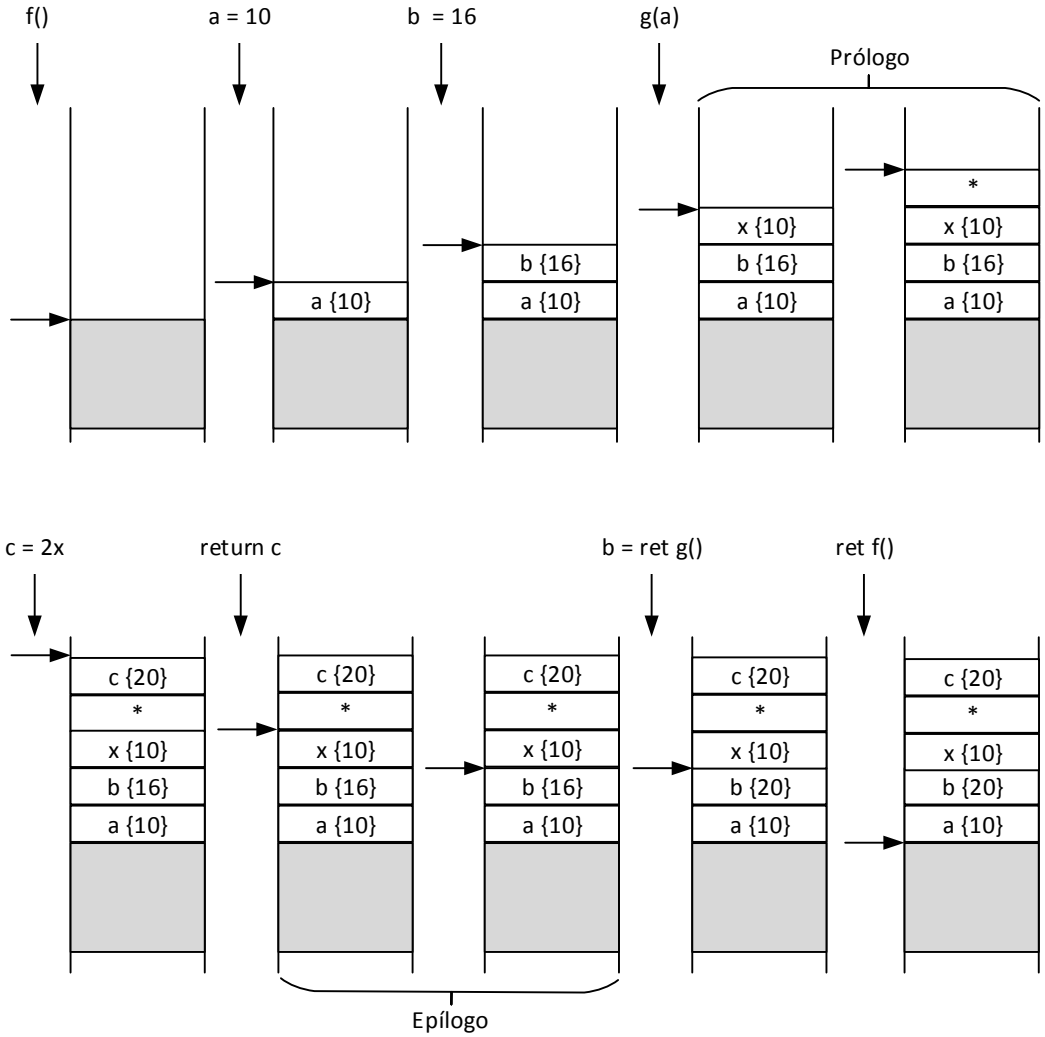

Figura 4.15: Stack na chamada de funções na arquitetura x86.

dos registradores e quando o thread voltar a ser executado, estes valores devem ser carregados de volta. O procedimento é chamado de Troca de Contexto. Os valores dos registradores são armazenados na Stack. A restauração do contexto implica na continuação da execução dos threads já que o Instruction Pointer volta à mesma posição antes da Troca de Contexto. A figura 4.16 exemplifica o procedimento discutido. É importante lembrar que o Escalonador do SO determina quando um thread executando deve pausar e outro deve voltar/começar a executar.

Será visto daqui em diante como o MOS lida com a memória em Sistemas Operacionais. O primeiro ponto importante é que, apesar da MOSLanguage não permitir alocação dinâmica (ou seja, as ModIns não alocam variáveis, não ocupando a Heap), o MOS usa alocação dinâmica nos MOSProcesses. Isto está relacionado sobretudo ao limite de de threads e recursos não ser definido. As classes list/deque do C++ criam listas cujo tamanho pode aumentar com a demanda (alocando mais memória quando preciso). Os objetos relacionados a threads, mutex, variáveis condicionais, Modules, instância de modules, Links e etc. exemplos do que pode ser criado sobre demanda.

Apesar do MOSProcess alocar dinamicamente, os ModIn de MOSProcesses continuam sem essa capacidade. Permitir que ModIn de certos MOSDeviceTypes tenham essa habilidade seria uma mudança considerável a ser feita na MOSLanguage, MOSBuilder e MOSProcesses. Por exemplo, como a MOSLanguage não utiliza ponteiros, a alocação/desalocação teria de ser feita pelo MOS, talvez implementando mecanismos similares aos Link entre Modules, como um Link entre um Module e um espaço de memória alocado.

Toda ModIn tem seu tamanho definido (de acordo com o setor 2 do CMod). O MOSProcess aloca esse espaço dinamicamente. Como já foi mencionado, quando um thread é criado um novo espaço é alocado na memória virtual para servir de Stack do thread, portanto não é preciso se preocupar com a Stack quando uma ModIn cria um thread ou quando o próprio MOS cria um thread.

4.2.2 MOSBuilder

O MOSBuilder possui três etapas. A primeira gera um CMod, a segunda transfere o CMod para o MOSDevice de destino, e a terceira transforma o CMod em Rmod. Carregar o código de uma CMod no programa (criando um RMod) não é tão simples porque o segmento Code na memória não pode ser escrito pelo MOSProcess. A solução para este caso é tornar o RMod uma biblioteca dinâmica. Na primeira etapa, o primeiro setor do CMod (instruções de máquina) deve ser compilado como uma biblioteca dinâmica, e as outras tabelas devem ser criadas da mesma forma. Na segunda etapa, o CMod deve ser armazenado no HD como um arquivo enquanto é

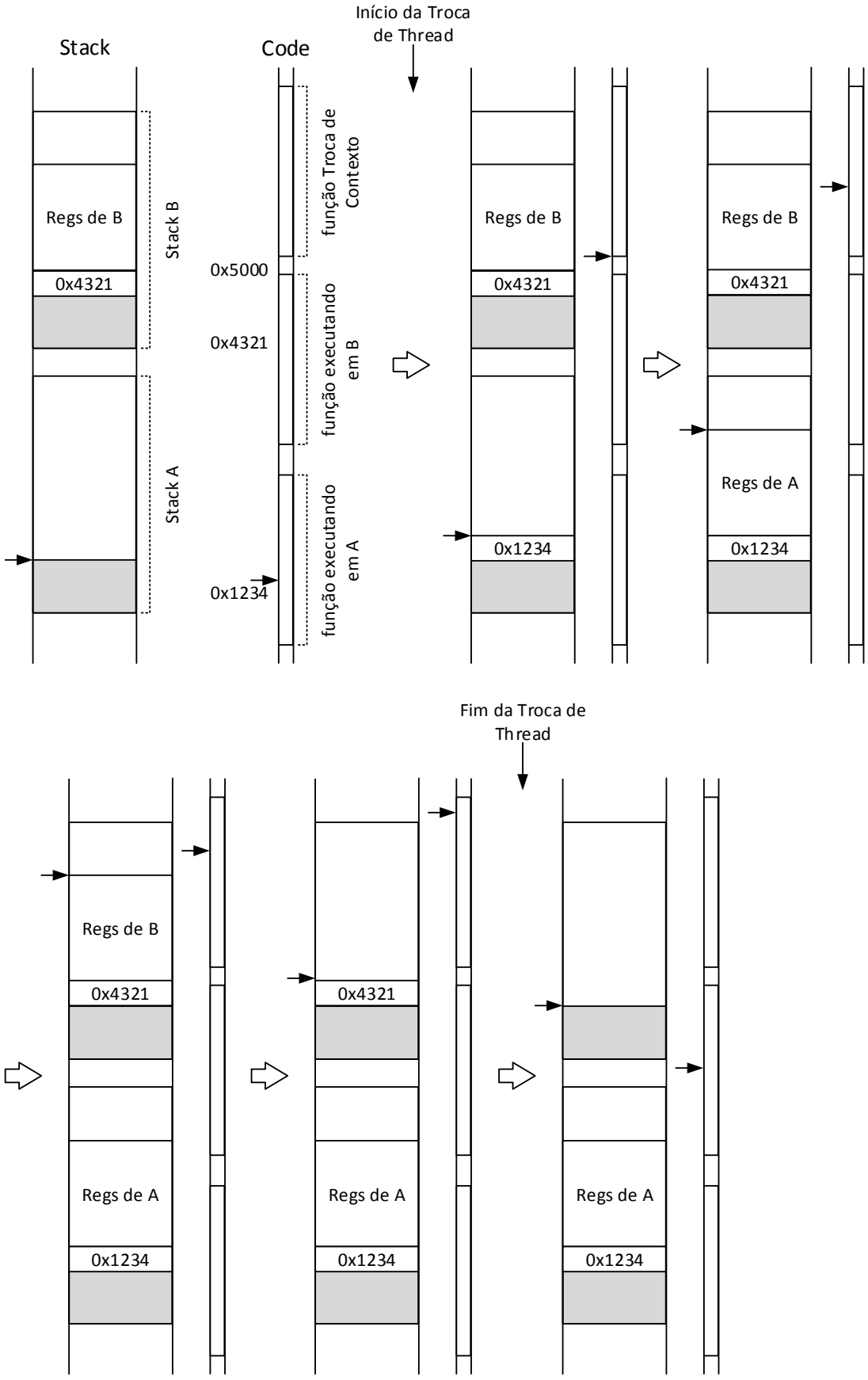

Figura 4.16: Stack na troca de threads.

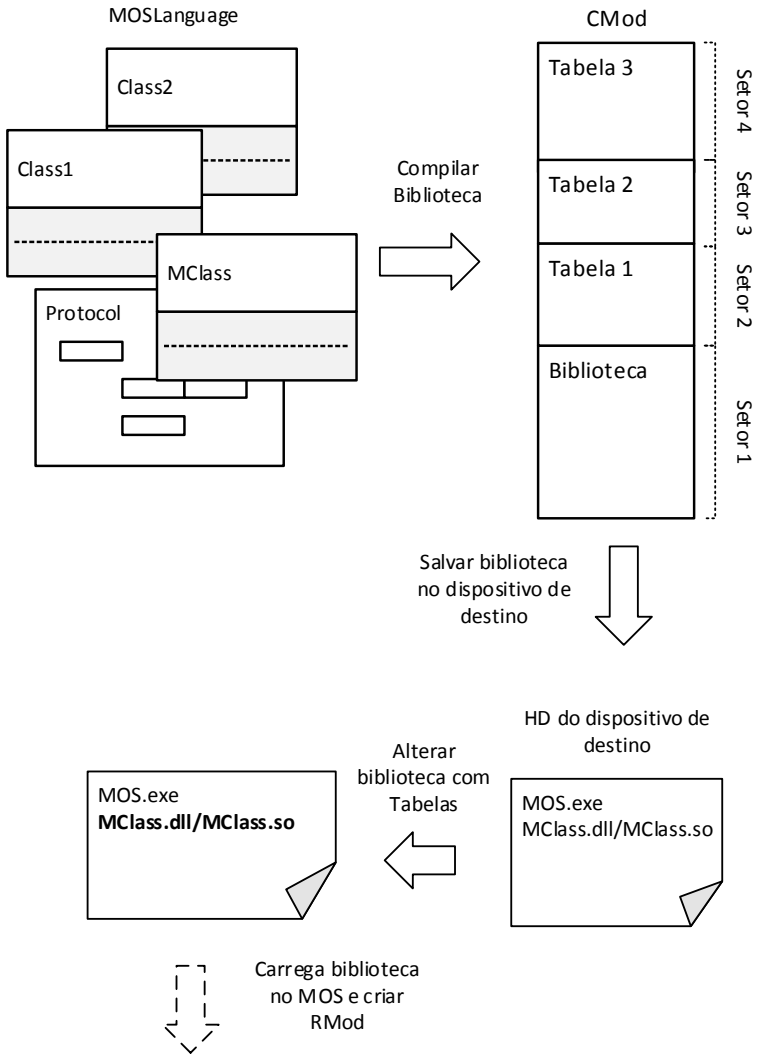

Figura 4.17: MOSBuilder lidando com bibliotecas nos MOSProcesses.

recebido. Na terceira etapa, um outro arquivo irá conter o primeiro setor es será de fato uma biblioteca dinâmica (extensão .dll no Windows ou .so no Linux). Depois as tabelas do CMod serão usadas para corrigir os endereços dentro desse arquivo. Para finalizar a terceira etapa basta o MOSProcess abrir a biblioteca e chamar uma função para criar o Module. O procedimento é descrito na figura 4.17.

4.2.3 Observações

A grande vantagem da implementação para SOs surge das bibliotecas dos SOs e/ou bibliotecas multiplataforma (Boost). Desta forma manipular threads e objetos de sincronização de threads na implementação do MOS é ainda mais simples.

Além do Boost muitas outras bibliotecas podem ser incorporadas no MOS. Esta é a principal vantagem do MOSProcess em relação aos MOSMicros. Entre elas estão bibliotecas de Interface Gráfica, Visualização em 3D, áudio, vídeo etc. Vale lembrar

que existem três formas para desenvolver este conceito. Uma consiste nas bibliotecas serem usadas em NativeModules, que poderiam ser conectados (MOSLink) ou se comunicar (Protocol) com outros Modules. Outra consiste em criar outros processos (que usam a biblioteca) que se comunicariam com o MOS através da MOSAPI. A terceira seria inserir as funções de bibliotecas dentro do System de BModClasses de MOSProcesses. Na visão do autor as funções extras das BModClasses devem ser específicas do SO, portanto a última opção é desencorajada. Considerando o que foi proposto as BModClasses de MOSProcess não tem acesso a um grande número de funções extras, possui apenas as funções relacionadas à criação de Nets. Algumas bibliotecas poderiam ser escolhidas para serem incorporadas no System.

A alocação dinâmica é outro fator positivo na implementação do MOS em SOs. Ao invés de se criar um array com tamanho fixo para os objetos das classes *Thread*, *Link*, *Mutex*, *RMod*, *ModIn* e etc, o MOS pode criá-los sob demanda, não impondo limitações no MOS como acontece nos MOSMicros. Um grande vilão na MOSLanguage são as strings. Um simples texto pode não ser nada em um computador mas nos MOSMicros que possuem uma pequena memória isto é um problema. Até mesmo guardar a informação básica dos RMods é um problema (strings para os nomes do MOSProject e Module) quando o número de RMods é grande. Os MOSProcesses se saem bem neste problema.

4.3 Implementação do MOS para Microcontroladores

Para adaptar o MOS para o microcontrolador ATmega2560 (escolhido como microcontrolador de exemplo), pode-se usar a biblioteca de C++ AVR Libc [55] com compilador GCC. As portas e os registradores especiais (ligados ao hardware) estão disponíveis para o programador. A AVR Libc possui conjuntos de funções para lidar com a EEPROM ou até mesmo o bootloader. Antes de ver a adaptação das classes, é necessário estudar como os MOSMicros lidam com a memória, como funcionará o kernel do MOS e como o MOSBuilder lida com estas diferenças.

4.3.1 Memória

O ATmega2560 foi usado como estudo de caso de adaptação do MOS para microcontroladores. Será feita uma análise da organização de suas memórias.

O ATmega2560, escolhido como teste de conceito para microcontroladores possui 3 memórias diferentes. A SRAM (Static Random Access Memory) de 4KB é a memória usada pela CPU para armazenar e acessar dados aleatoriamente durante seu funcionamento. A In-System Programmable Flash de 256KB é a memória que

contêm as instruções de máquina que a CPU irá executar. Diferente da Arquitetura Harvard (o qual o x86 se baseia), a memória de dados e das instruções de máquina são separadas. O documento [56] possui uma descrição da capacidade de auto programação do AVR. A memória Flash é dividida em duas seções, a NRWW e a RWW, sendo que instruções na RWW são capazes de alterar a própria Flash. A mesma Flash é dividida em Boot Loader e Application Sections, mas o Boot Loader está sempre contido na seção RWW. A EEPROM (Electrically Erasable Programmable Read-Only Memory) de 8KB é usado para armazenar dados de forma não volátil de acordo com a necessidade da aplicação.

Os códigos do Boot Loader e da Application são compilados separadamente e podem ser carregados no microcontrolador separados. O código na NRWW não consegue alterar o código na Flash, mas a RWW consegue alterar qualquer setor da memória Flash. A área do RWW é muito menor do que o tamanho do Application, então o código relacionado à aplicação do microcontrolador deve residir na NRWW e apenas a capacidade de auto programação deve residir no RWW. Ou seja, a Application fica na NRWW e o Boot Loader fica na RWW. A figura 4.18 mostra a relação entre as seções da memória Flash. É possível que o MOS no ATmega2560 resida parcialmente no RWW mas o MOS certamente usará toda a seção NRWW para criar o Escalonador.

O setor de memória Application é organizada como na memória virtual de um processo. O ATmega2560 utiliza o compilador AVR-GCC [57], uma modificação do compilador GCC da linguagem C++, por isto o modelo de memória é muito parecido. No entanto, o microcontrolador não tem proteção de memória na SRAM e possui apenas a proteção de RWW/NRWW na Flash. Sistemas Operacionais reservam pedaços de Stack (com tamanhos determinados) quando threads são criados, aumentando a área da Stack mesmo se cada pedaço da Stack não estiver sendo usado por completo. Isto torna possível detectar a colisão entre a Heap e a Stack em SOs. No ATmega2560 não existem threads, apenas um "processo", cujo pedaço de Stack é todo Free Space. A Heap cresce no sentido oposto à Stack sem qualquer forma de detectar a colisão no Free Space. A alocação dinâmica no ATmega2560 é perigosa.

Um aspecto muito importante de microcontroladores são as Interrupções. O programador tem liberdade para ligar/desligar/disparar-los. O capítulo 7.8 de [35] explica o funcionamento das interrupções enquanto a tabela 14-1 lista as interrupções. Interrupções possuem prioridade de acordo com a tabela. O programador pode definir a função de atendimento da interrupção. Enquanto uma interrupção está sendo atendida as outras interrupções esperam o término do atendimento. Este é o comportamento padrão mas o programador pode liberar o atendimento de outras interrupções enquanto atende uma interrupção (Nested Interruption).

Uma análise das interrupções pela ótica da memória será apresentada. Quando

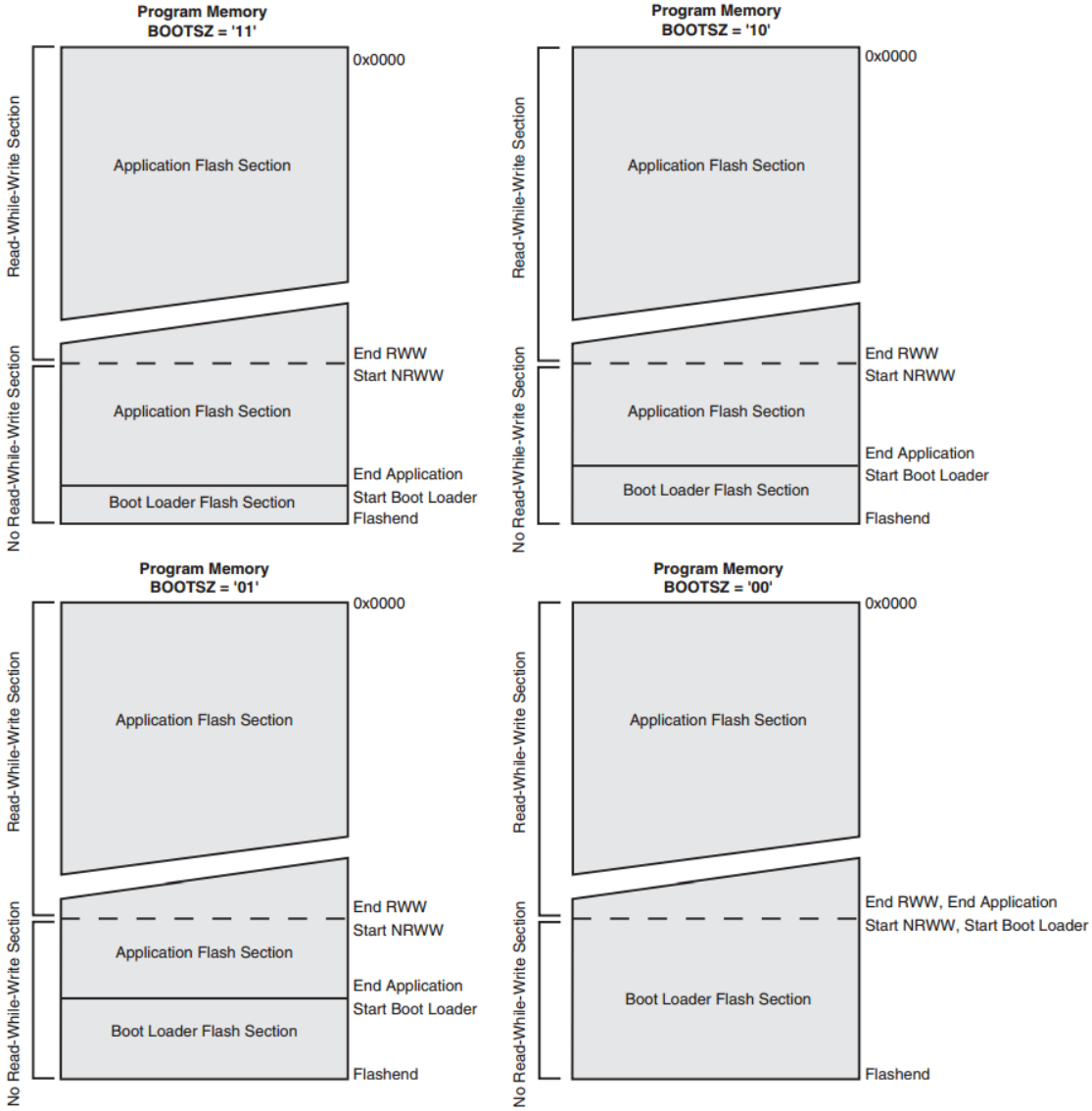

Figura 4.18: Divisão da memória RAM no AVR.

o ATmega2560 é iniciado a função main() é executada. Dois registradores muito importantes são o Program Counter (PC) que indica o endereço da próxima instrução na Flash a ser executada, e o Stack Pointer (SP) que indica a posição atual da Stack na SRAM. Ambos seguem a execução do main() definido pelo programador, executando instruções e alocando memória na SRAM. Quando uma interrupção é iniciada uma Troca de Contexto de armazenamento é executada, salvando os registradores usados pela CPU na posição da Stack. O PC passa a apontar em outra posição na Flash onde está definido a função de tratamento da interrupção definida pelo programador. Quando a função termina, outra Troca de Contexto é feita mas no sentido contrário, uma Troca de restauração, retirando bytes da Stack e passando para os registradores, o que inclui o antigo PC. Ou seja, no fim do atendimento da interrupção o programa principal vota a ser executado. A figura 4.19 representa este procedimento. Uma Nested Interruption equivale a duas Trocas de Contexto de armazenamento e depois duas de restauração. A figura 4.20 representa a Troca de Contexto na Nested Interruption. Uma observação para finalizar: a Troca de Contexto no atendimento de uma interrupção feita pode ser desabilitada pelo programador caso deseje isto (é o caso do MOS).

4.3.2 Kernel

A implementação de threads é bem diferente entre MOSDeviceTypes. Nos MOSProcesses, threads são implementados com bibliotecas de threads. Algumas são bibliotecas dos Sistemas Operacionais e as outras são criadas em cima das bibliotecas dos SOs. As dos SOs são as que acessam o Kernel do SO para criar, destruir threads e etc. Já os MOSMicros não possuem um Sistema Operacional, portanto o próprio MOS deve conter um Escalonador para realizar o gerenciamento de threads.

Desconsidere temporariamente a eliminação de alocação dinâmica que o MOS impôs para microcontroladores. Os threads poderiam ser alocados dinamicamente junto de outras variáveis que o próprio MOS ou as ModIns alocassem (MOSLanguage permitiria a alocação dinâmica de variáveis). Um grande problema que surge nos algoritmos de alocação dinâmica em microcontroladores é a fragmentação de memória na Heap. Não há o conjunto SO, MMU, memória virtual e grande quantidade de memória para reorganizar a alocação e "curar"o sistema da fragmentação. Se considerarmos o tamanho de variáveis alocadas dinamicamente por programadores com o tamanho da Stack de threads deduz-se que em quase a totalidade das vezes a Stack será consideravelmente maior do que as variáveis alocadas. A fragmentação que ocorre com a alocação e liberação de variáveis pode rapidamente impedir a alocação de novos threads pois a memória fragmentada passa a não ter pedaços grandes (do tamanho da Stack de threads) desalocados. Se um algoritmo de alocação pudesse

```
void i{
            ...
} // Interrupção

int void f(){
            ...
}

int main(){
            f();
            return 0;
}
```

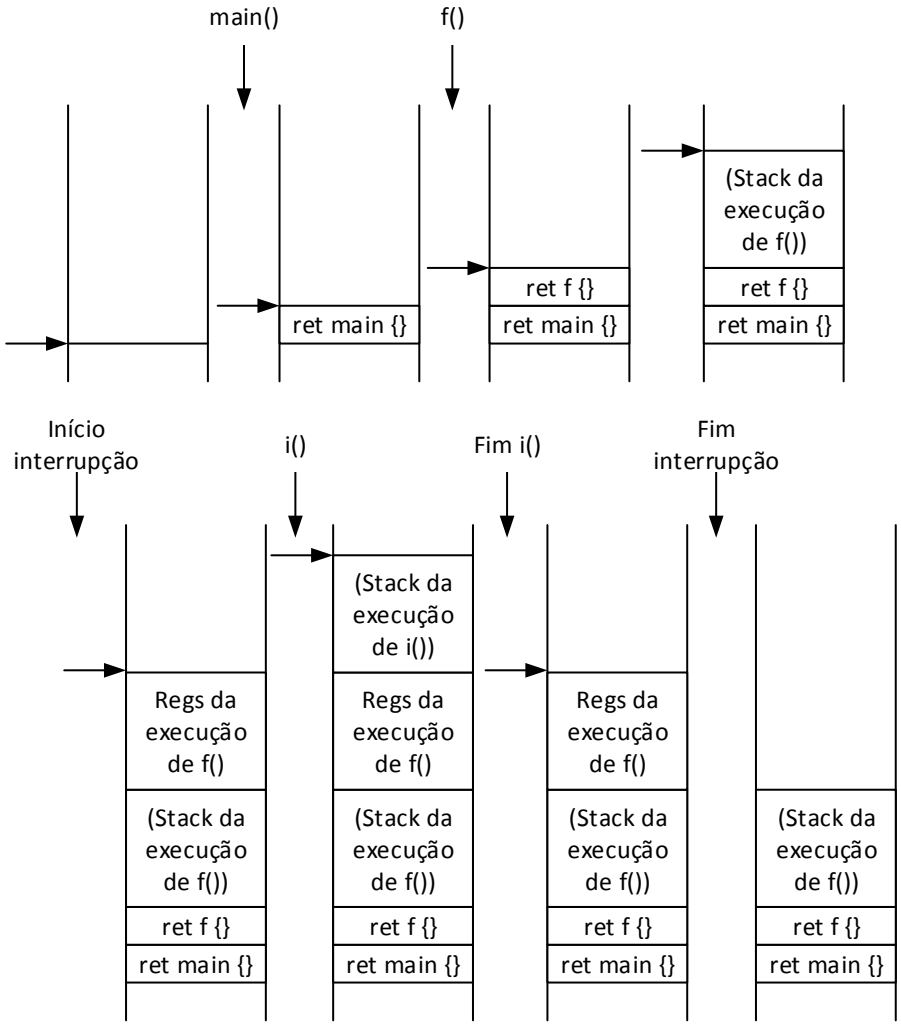

Figura 4.19: Stack no atendimento de uma interrupção.

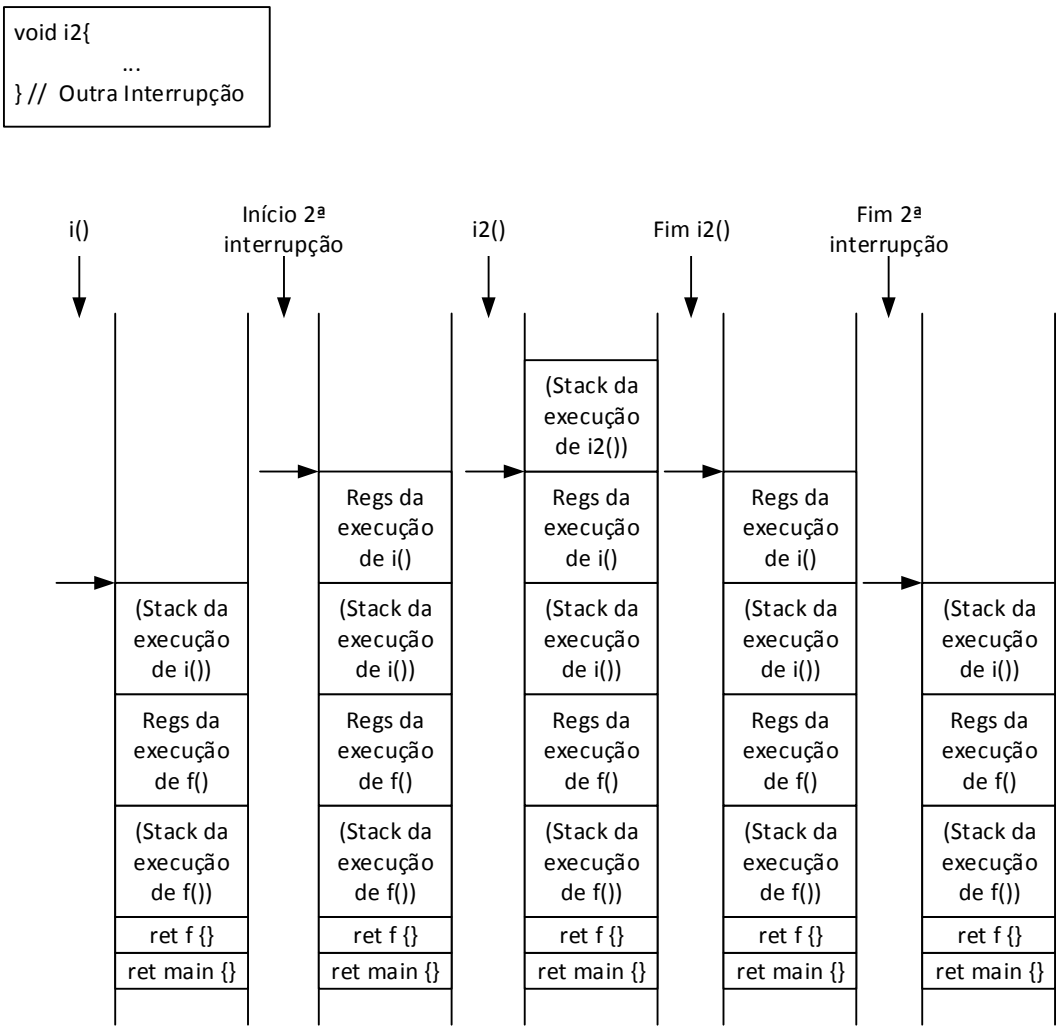

Figura 4.20: Stack no atendimento de Nested Interruptions.

garantir que a fragmentação fosse corrigida ou atenuada então MOSMicros poderiam ter de fato alocação dinâmica. Vale lembrar que estes algoritmos de alocação são mais complexos e portanto possuem um custo computacional maior. Um Kernel deve ser usado nos microcontroladores. O caminho adotado na dissertação é o de não tentar resolver a questão de fragmentação e basear a implementação do Kernel para o MOS no FreeRTOS, e com isto proibindo a alocação dinâmica. Vale lembrar que este capítulo trata de uma sugestão de como o MOS pode ser implementado, e portanto se um algoritmo de alocação escolhido/criado, e atpós testado fosse comprovada sua eficácia para resolver a fragmentação, o MOS passaria a ter alocação dinâmica, ou seja, os MOSMicros poderiam alocar variáveis como os MOSProcesses e a MOSLanguage teria alocação dinâmica.

O documento [54] é um ótimo ponto de partida para a criação de um Escalonador para o ATmega2560 baseado no FreeRTOS. Isto não é o suficiente pois não leva em conta a Arquitetura De Software proposta para o MOS, mas a ideia continua válida.

Criaremos um exemplo para demonstrar como um Escalonador pode ser criado no AVR levando em consideração uma classe global que representa um SO ou Kernel. O exemplo consiste na configuração de threads para em seguida executar um Escalonador do tipo Round Robin. Um algoritmo RR cria fatias de tempo máximo que threads podem executar. Quando um thread estoura sua fatia de tempo o próximo thread da lista passa a executar. O próprio thread pode causar sua interrupção quando deseja dormir ou quando espera um recurso ficar disponível, mas por questões didáticas consideraremos apenas o caso em que o thread chama a função para dormir.

Normalmente o Kernel de um Sistema Operacional lida com uma série de recursos do dispositivo, mas concentraremos apenas no Escalonamento. O desejado é a criação uma classe (*Kernel*) com uma função de inicialização (*Start()*), uma função que um thread chama para dormir por um tempo (*ThreadSleep()*) e uma função para que um thread crie outro thread (*ThreadCreate()*). Cria-se um objeto global dessa classe (*kernel*) e na função *main()* do microcontrolador chamamos a função de inicialização desse objeto (*kernel.Start()*). Na *Start()* divide-se o Free Space no número de threads (número fixo, determinado na hora da compilação) que podem ser criados pelo Kernel. A *Start()* cria threads de teste e começa a executar um deles. A figura 4.21 mostra como fica a Stack no instante em que o *Start()* realiza tal procedimento, dividindo a Stack para cada thread. Uma interrupção de tempo do Round Robin ou um *kernel.ThreadSleep()* no próprio thread faz com que o algoritmo de escalonamento seja executado, levando outro thread a ser executado. Vale lembrar que os threads podem chamar o *ThreadSleep()* por que o *kernel* é um objeto global. A interrupção de Timer também chama uma função do *kernel* para executar o escalonamento. A figura 4.22 mostra como a Stack fica no atendimento

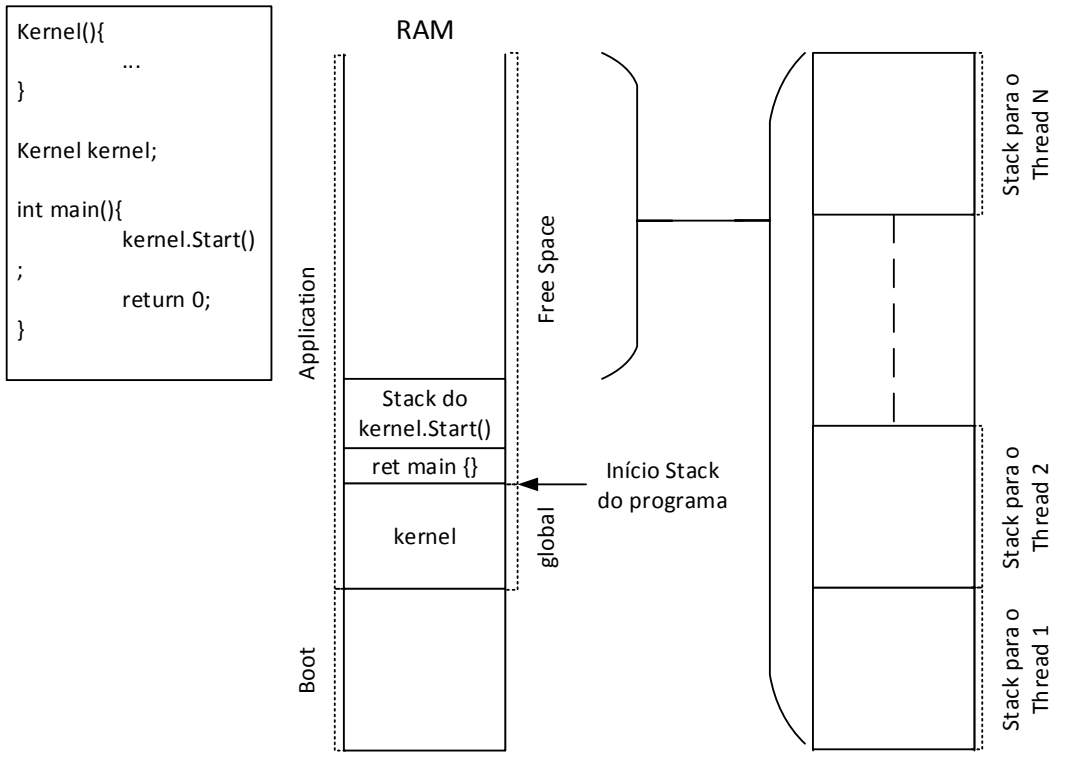

Figura 4.21: Stack no kernel.

da interrupção de Timer e no *ThreadSleep()*.

Um problema que não tem uma solução prática para a implementação do Kernel para ATmega2560 é o Stack Overflow. Este termo significa que um thread tentou alocar mais memória em sua Stack mas a Stack reservada para o thread acabou, e uma próxima alocação ocorreria na memória reservada para outra coisa. Sistemas Operacionais modernos conseguem detectar esse problema e emitem um erro. O AVR infelizmente não tem qualquer tipo de proteção neste sentido. A figura 4.23 mostra o que acontece na Stack do Kernel planejado quando um thread começa a usar a Stack de outro thread. Para impedir o Stack Overflow, sempre que as instruções *push* (instrução que adiciona ou retira bytes da Stack no endereço do registrador SP) fosse usada seria preciso verificar se o novo endereço da SP ultrapassa o limite da Stack. Esta instrução é usada tanto em C++ (implementação do MOS) quanto na MOSLanguage (ModIns). Códigos em C++ compilados não realizam essa verificação, portanto seria necessário modificar o AVR-GCC. O compilador da MOSLanguage que não foi desenvolvido mas poderia ter esta modificação. Mesmo sendo possível implementar esta solução, ela não é viável pois seria o equivalente a mudar uma instrução por 10 ou mais instruções em um microcontrolador de apenas 16 MHz.

O algoritmo Round Robin necessita de um temporizador para identificar o fim da fatia de tempo. Os timers do ATmega2560 possuem 8 e 16 bits. É indicado o

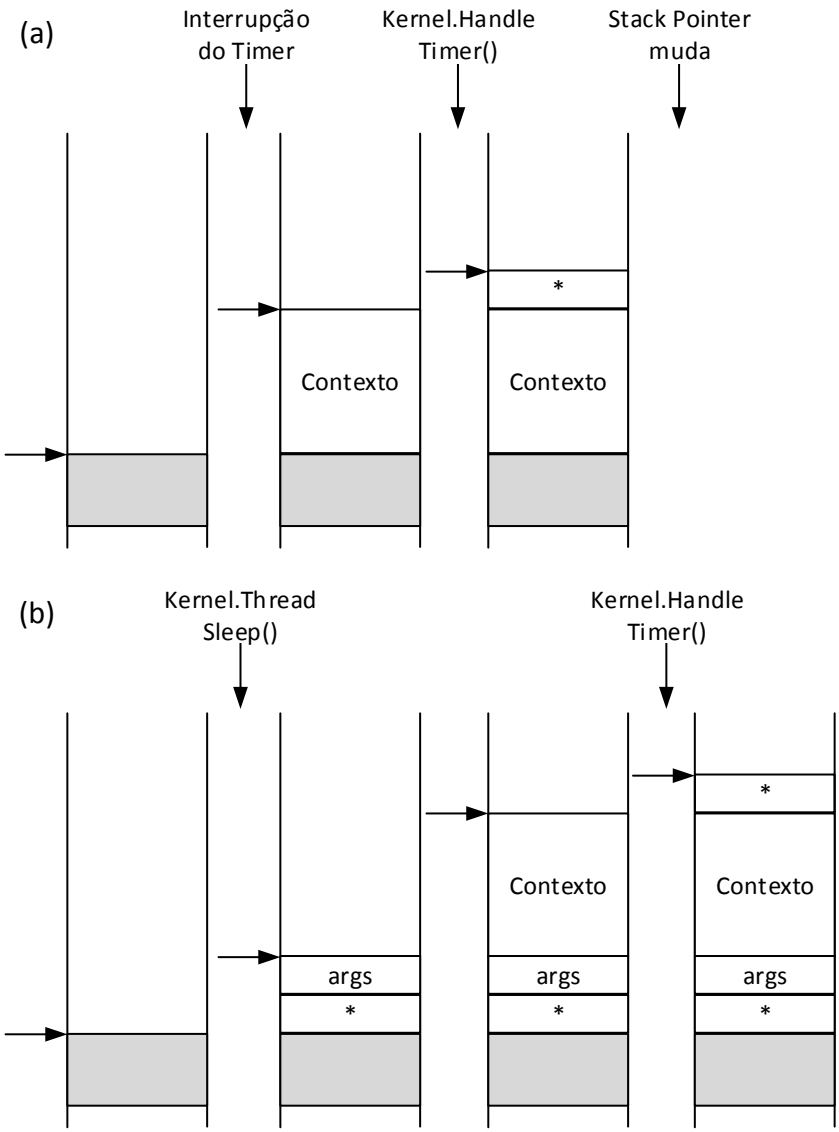

Figura 4.22: (a) Stack na interrupção de comparação do Timer no kernel (b) Stack na chamada da função ThreadSleep() do kernel.

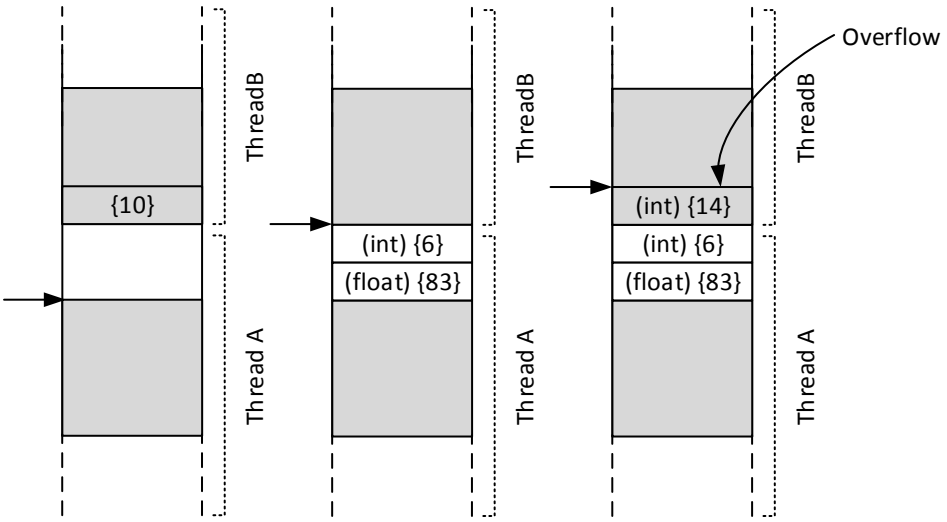

Figura 4.23: Stack Overflow.

uso do de 16 bits para aumentar a precisão do Timer. As interrupções de Timer que usaremos são a Compare e a Overflow. Um período (múltiplo do período da CPU) será usado para representar um tick do timer, sendo 65535 o número máximo de ticks (16 bits). A interrupção Compare ocorre quando certo número de ticks (determinados pelo programador) tiverem passado. Se ocorrerem 65536 ticks a interrupção de Overflow ocorre. Por exemplo, se cada tick for um período da CPU 62,5ns (1/16MHz), um overflow ocorre a cada 4,096ms se o timer não for reiniciado, logo, para medir 10ms é preciso identificar 2 overflows mais um certo número de ticks do Timer. Este timer será exclusivo para o kernel, ou seja, este timer não estaria disponível para as BModClasses do ATmega2560.

A figura 4.24 mostra um fluxograma das etapas de execução do Kernel para o ATmega2560. As etapas destacadas são explicadas a seguir:

Configurar Interrupções de Timer Interrupções são ligadas. O período do Round Robin é usado para indicar o número de ticks e overflows em que o primeiro thread deve ser interrompido e o segundo iniciar sua execução.

Configurar Threads Funções globais são escolhidas para serem executadas em threads. Cada thread ganha uma Stack cujo tamanho depende do Free Space e do número de threads máximo. Será feita uma analise do que ocorre em cada thread. Muda-se o SP para o início da nova Stack do thread, insere-se o endereço da função global na Stack e depois um contexto de "lixo" (não importa os valores). Nova SP é salva nas informações do thread (dentro do objeto kernel). Volta-se o SP para a posição anterior.

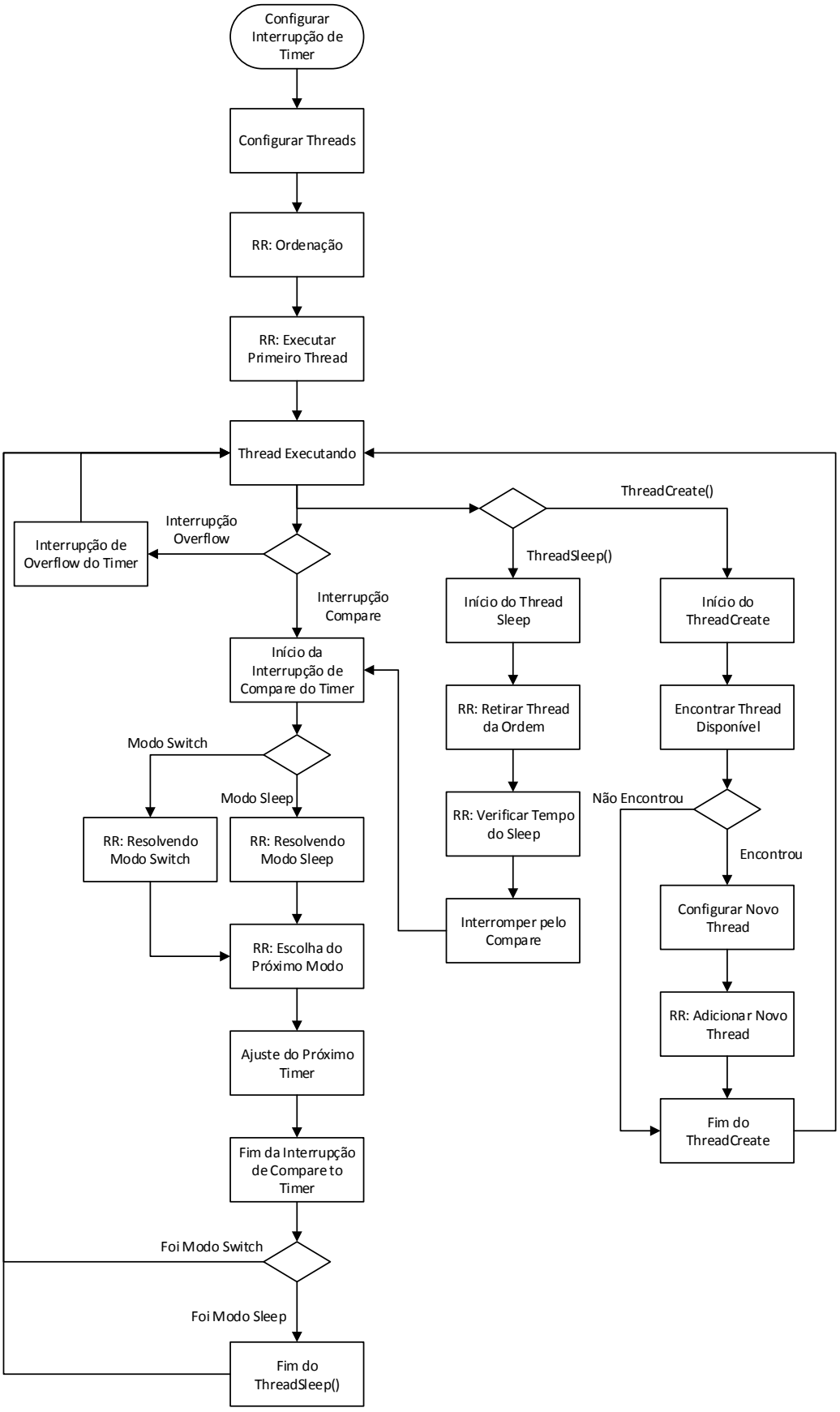

Figura 4.24: Etapas de execução do Escalonador do kernel.

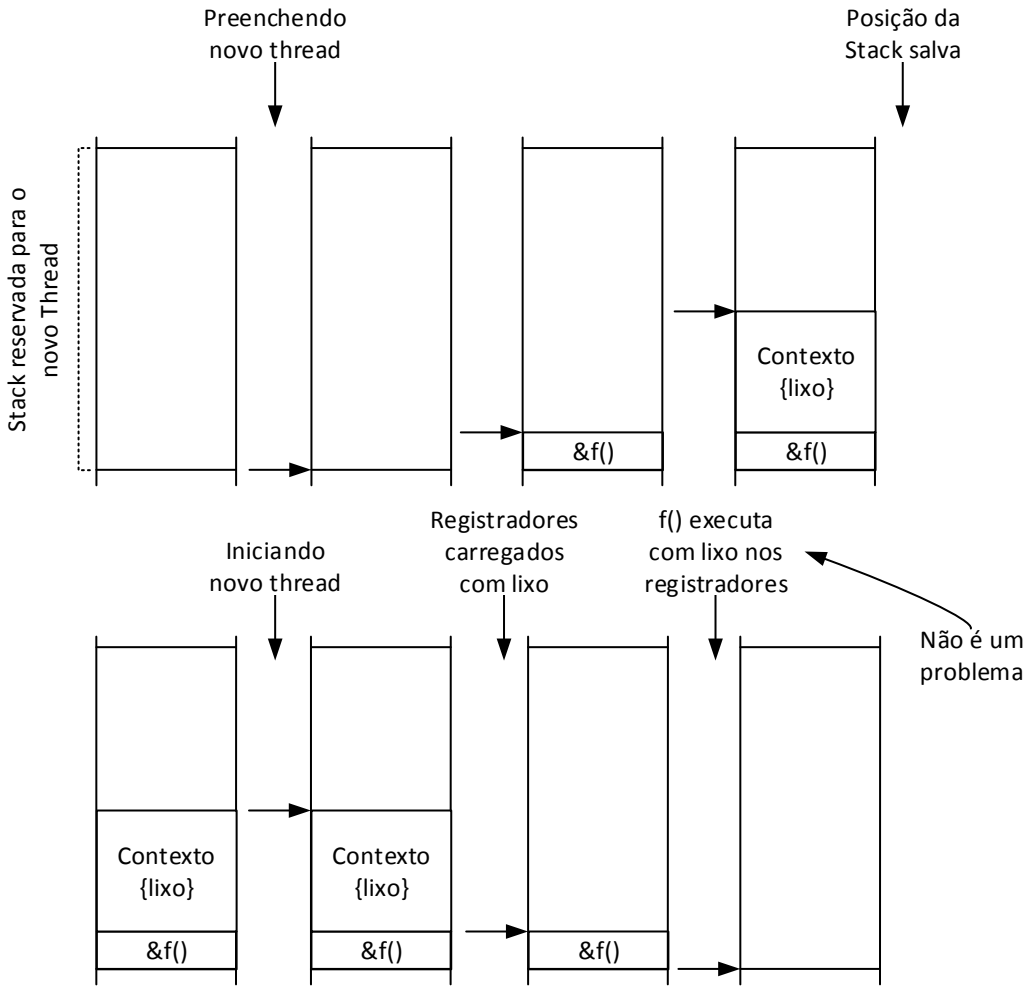

Figura 4.25: Stack inicial de um thread no kernel.

Observação: Os valores do contexto inserido não importam porque quando um thread for escolhido para execução, o SP será direcionado para o topo da Stack do thread, depois o contexto será restaurado. O último item a ser restaurado é o PC, cujo valor será o endereço da função global do thread. Como esta função será iniciada o valor de nenhum registrador importa. No entanto quando o thread executando for pausado em diante a Stack o contexto salvo contém valores pertinentes aos registradores da execução da função global. A figura 4.25 mostra a Stack quando o thread é executado pela primeira vez e quando for salva novamente.

RR: Ordenação Os threads criados já estão prontos para execução. O Escalonador deve escolher a ordem de execução de cada um dos threads. Quando um thread executando se bloquear ou se sua fatia de tempo de execução esgotar, o Escalonador já sabe qual é o próximo thread que deve ser executado. Esta

ordem cria uma lista circular. Todo thread pronto para executar ou executando sabe quem é próximo e o anterior na lista.

RR: Executar Primeiro Thread Primeiro indica-se o endereço do SP do thread a ser executado e inicia-se a restauração do contexto. O Program Counter é preenchido com o endereço o endereço da função do thread (o thread é de fato iniciado) que estava armazenada na Stack. Uma parte da figura 4.25 (figura acima, figura da observação) mostra o procedimento de execução do primeiro thread.

Thread Executando A função global do thread está executando e sua Stack é preenchida com variáveis locais ou chamada de funções (figura 4.25). O thread executa até que uma interrupção ocorra, o thread chame *ThreadCreate()*, ou o thread chame *ThreadSleep()*.

Interrupção de Overflow do Timer A Interrupção de Overflow salva o contexto do thread. O *kernel* possui variáveis que indicam quando o próximo thread deve executar, além de manter um relógio interno. A interrupção os atualiza e reconfigura o Timer para o compare ou outro overflow. A interrupção acaba, restaurando a execução do thread.

Início da Interrupção de Compare do Timer A interrupção atualiza o relógio interno do *kernel*.

Esta interrupção pode ocorrer por dois motivos:

- Um thread estourou seu tempo de execução e outro thread precisa ser iniciado de acordo com o algoritmo Round Robin, trocando o thread em execução.
- Um thread dormindo passou pelo tempo que devia dormir e deve entrar novamente no algoritmo Round Robin como o último na lista.

Internamente o Round Robin é visto com dois modos que representam os motivos do Timer estar funcionando, os modos Switch e Sleep.

RR: Resolvendo Modo Switch O thread que executará a partir de agora está indicado no thread sendo executado. O endereço da Stack do novo thread é guardado em uma variável do *kernel* que é usada na restauração de contexto. Isto fará o final da interrupção continuar a execução do próximo thread. Outra variável interna do *kernel* é atualizada com o tempo do próximo switch.

RR: Resolvendo Modo Sleep O thread que está acordando acessa o executando atualmente e o último executado (início e fim da lista do RR) para virar o

novo último executado (o thread acordando entra no final da lista). Um loop varre todos os threads que estão dormindo atualmente para descobrir qual o próximo thread que deve acordar. Esta informação é salva em uma variável do *kernel*.

RR: Escolha do Próximo Modo As variáveis de tempo de quando próximo switch acontece ou o próximo thread acorda são comparados. O menor é o próximo modo.

Ajuste do Próximo Timer Deve-se configurar qual interrupção do Timer (Compare ou Overflow) deverá ser ativada. Cálculos de ticks e overflow usam as variáveis de tempo mencionadas.

Fim da Interrupção de Compare do Timer O SP aponta para a memória de um thread que pode ser diferente de quando a interrupção ocorreu. O contexto é restaurado, levando um novo valor ao PC. O thread volta a executar. Se o thread a executar estava no *ThreadSleep()*, voltará para o *ThreadSleep()*.

Início do ThreadCreate A função *ThreadCreate()* recebe o endereço de uma função global. As interrupções são desabilitadas enquanto o RR é alterado.

Encontrar Thread disponível O número de threads é limitado no dispositivo. Um algoritmo busca um thread disponível (não está sendo usado) dentro do *kernel*. Se encontrar, o thread deve ser configurado. Se não encontrar, a criação do thread é cancelada.

Configurar Novo Thread Esta etapa é muito similar à etapa Configurar Threads, em que a Stack é preenchida com valores "lixo" e o endereço da função global.

RR: Adicionar Novo Thread O thread deve entrar no final da lista circular de execução do RR. Todo thread sabe quem é o próximo a thread a ser executado o thread que deve que foi executado antes. O thread executando atualmente e o último executado são alterados para inserir o novo thread na lista.

Fim do ThreadCreate As interrupções são reabilitadas e o thread que estava rodando volta a executar.

Início do ThreadSleep Quando um thread deve mudar seu estado para dormindo, outro thread deve ser escolhido para execução imediatamente. As interrupções do Timer são desabilitadas pois não deve-se esperar que elas aconteçam para o próximo thread começar a ser executado.

RR: Retirar Thread da Ordem O thread será bloqueado por certo tempo, então deve sair da lista circular de execução.

RR: Verificar Tempo do Sleep Calcula-se o tempo em que este thread deve executar. Uma variável do *kernel* indica quando o thread que acorda mais cedo irá acordar. Esta variável é comparada com o tempo calculado agora. Se o calculado for menor a variável do *kernel* é atualizada.

Interromper pelo Compare A interrupção de Compare do Timer precisa ser acionada para que outro thread seja executado. O modo do Timer é configurado como Switch para que um novo thread seja escolhido. Um malabarismo envolvendo Assembly talvez seja necessário para iniciar a interrupção deste ponto já que a interrupção deveria ocorrer pelo hardware. Para todos os efeitos, a interrupção de Compare é iniciada dentro do *ThreadSleep()*. Quando o thread acordar voltará no mesmo ponto em que a interrupção foi chamada.

Fim do ThreadSleep() A interrupção do Compare retorna dentro do *ThreadSleep()*. Operações em Assembly são feitas para corrigir o endereçamento da Stack no retorno da interrupção. O retorno da *ThreadSleep()* faz o thread voltar a executar.

As etapas de execução mostram como o Kernel funcionaria, mas não mostra como ele poderia ser implementado. Um código em C++ foi criado para testar um Kernel para o ATmega2560 seguindo as etapas mencionadas. O código não foi testado em um dispositivo real, e sim simulado com o AVRStudio5 [58]. O código se encontra em [ANEXO 1]. A figura 4.26 é um diagrama UML deste código retirando o que não foi implementado: um segundo algoritmo de escalonamento; o Priority Scheduler (Escalonador de Prioridade); os objetos de sincronização de threads (barreira e mutex).

As funções globais *TestFunction1()*, *TestFunction2()* e *TestFunction3()* são funções de teste que serão executadas em threads criados pelo Kernel. Não importa o que as funções fazem, apenas que elas podem acessar o objeto *kernel* para chamar *ThreadSleepMicroseconds()*/*ThreadSleepMilliseconds()* e dormir por um período de tempo, ou chamar *ThreadCreate()* e iniciar outro thread. As funções equivalem à etapa Thread Executando.

É importante levar em conta que não é possível implementar o Kernel com apenas C++. Instruções em Assembly são necessárias nas operações que lidam diretamente com registradores.

Os defines *SaveContext()* e *RestoreContext()* definem a sequência de instruções de Troca de Contexto de armazenamento e restauração escritas em Assembly. Estes defines não devem ser funções de verdade pois funções adicionam/retiram bytes na Stack. Elas são usados em alguns lugares no código, mas sua funcionalidade fica evidente nas interrupções do Timer 1 *TIMER1_COMPA_vect* (Compare) e *TIMER1_OVF_vect* (Overflow). O comportamento padrão de uma interrupção é salvar

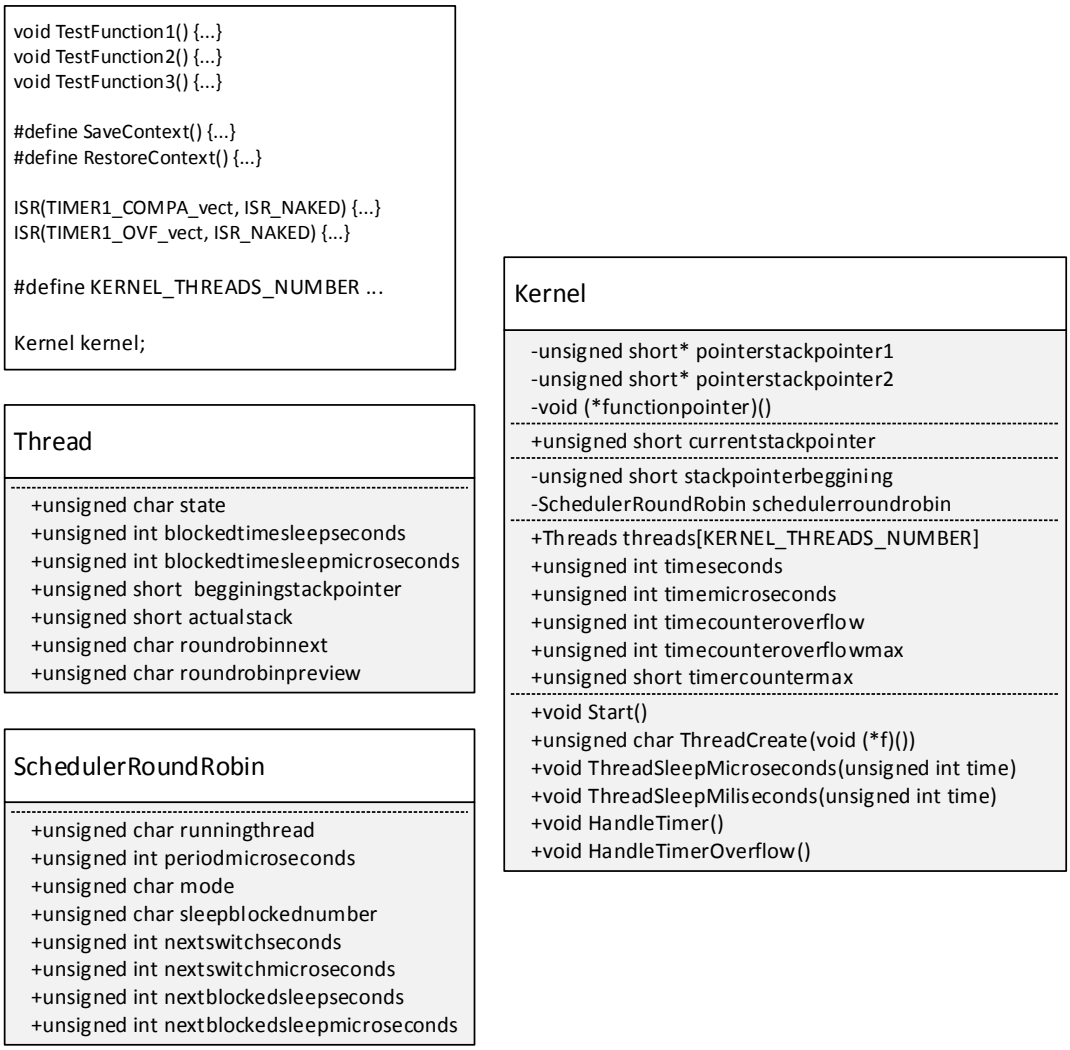

Figura 4.26: Classes e definições para o kernel.

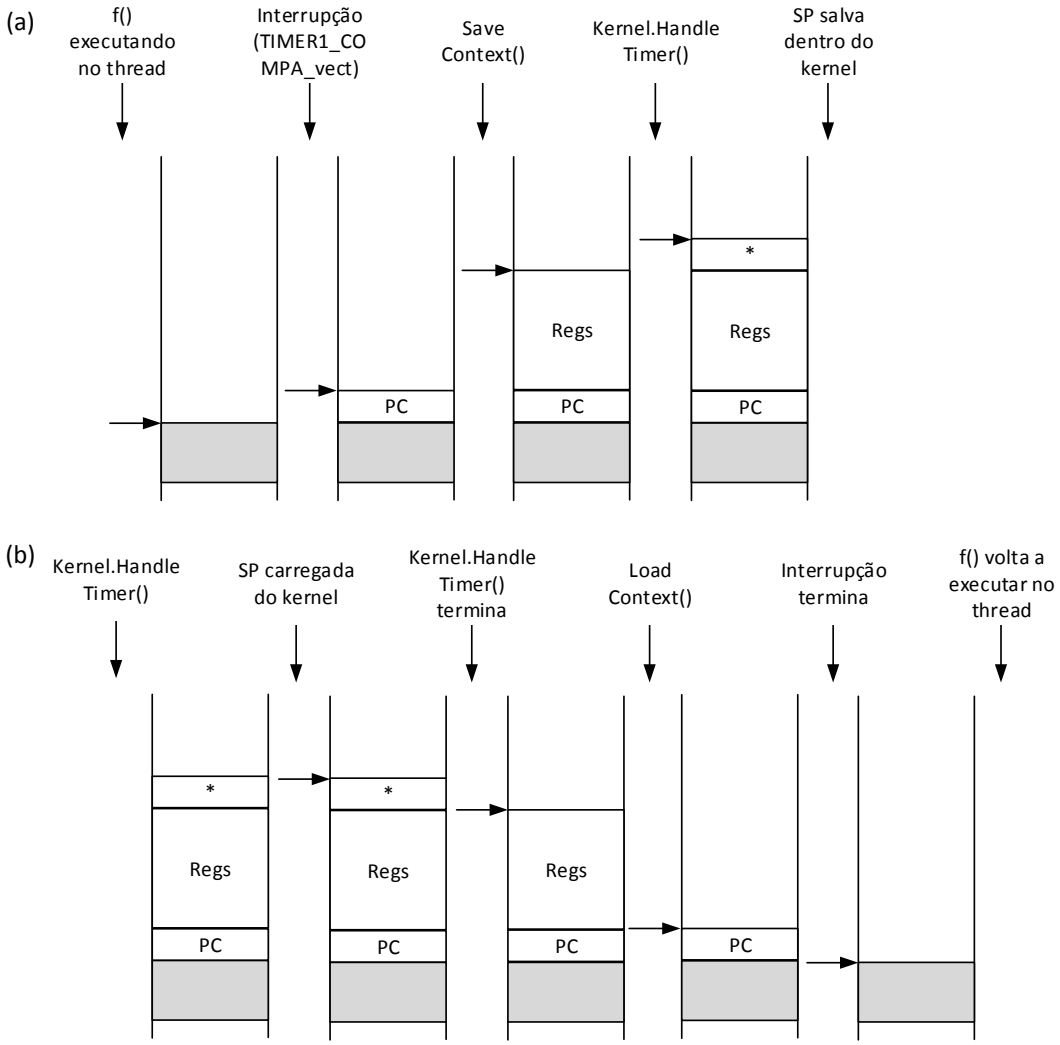

Figura 4.27: SaveContext() e RestoreContext().

o contexto, executar a interrupção, e restaurar o contexto salvo. No caso do Kernel deve-se salvar o contexto na posição atual e restaurar a de outro lugar. A informação é armazenada no objeto *kernel* então a Troca de Contexto padrão não pode ser usada. A *ISR_NAKED* retira a troca de contexto da interrupção, então *TIMER1_COMPA_vect* e *TIMER1_OVF_vect* chama *SaveContext()* e no final *RestoreContext()*, que usam as *pointerstackpointer1* e *pointerstackpointer2* do *Kernel* (acessados por Assembly).

SaveContext() e *RestoreContext()* não implementam a Troca de Contexto inteira por um item, o PC. Com ou sem o *ISR_NAKED* toda interrupção insere o PC automaticamente na Stack e o retira quando a interrupção termina. Este procedimento é comum a qualquer chamada de função. A figura 4.27 mostra a mudança na Stack quando uma interrupção é chamada.

Quando a interrupção do Timer é atendida usa-se a Stack do thread que está executando, e como o Stack Overflow é sempre um perigo, a interrupção deve evitar

preencher a Stack. Um truque que facilita a implementação do Kernel e ameniza este perigo é, ao invés de criar variáveis locais ou chamar funções globais, chamar uma função do *kernel* que usa suas próprias variáveis (o objeto já ocupa um pedaço da memória). As *TIMER1_COMPA_vect* e *TIMER1_OVF_vect* apenas chamam *kernel.HandleTimer()* e *kernel.HandleTimerOverflow()* entre o armazenamento e a restauração do contexto.

A classe *Thread* contém as informações pertinentes a um thread. A classe *Kernel* possui um vetor da *Thread* (objeto *threads*) que é preenchido de acordo com a demanda. Os threads são identificados com o índice desse vetor. A *state* indica se objeto está FREE (não há um thread criado nesse objeto), READY (o thread está sendo gerenciado pelo escalonador) ou BLOCKED (o thread está bloqueado por uma das funções de sleep). As *blockedtimesleepseconds* e *blockedtimesleepmicroseconds* indicam o instante em que o thread deve acordar. A *begginningstackpointer* aponta para o início da Stack do thread, e a *actualstack* aponta para a posição atual da Stack do thread. Dado um thread T que a classe representa, o *roundrobinnext* indica o índice do thread que deve ser executado depois de T, e o *roundrobinpreview* antes de T.

A classe *SchedulerRoundRobin* contém informações sobre o Escalonador Round Robin dentro do *Kernel* (objeto *schedulerroundrobin*). A *runningthread* indica o índice do thread que está sendo executado atualmente. A *periodmicroseconds* possui o período de tempo máximo que um thread pode executar no RR. O *mode* indica se o Timer do RR está sendo usado para acordar um thread ou para executar o próximo thread. Quando o *mode* é para acordar, o *sleepblockednumber* contém o índice do thread a ser acordado. As *nextswitchseconds* e *nextswitchmicroseconds* armazenam o instante em que a fatia de tempo do RR estoura, enquanto *nextswitchseconds* e *nextswitchmicroseconds* armazenam o instante que o primeiro thread dormindo deve acordar.

A classe *Kernel* representa um Kernel para o microcontrolador ATmega2560 que possui um Escalonador Round Robin. As variáveis *pointerstackpointer1* e *pointerstackpointer2* são acessadas por Assembly em diferentes pontos do código. O Stack Pointer não pode ser acessado diretamente, seu valor só pode ser obtido ou escrito por valores em registradores. O procedimento então consiste em carregar um valor nos registradores para depois passar para os registradores do SP, ou o contrário. As variáveis *pointerstackpointer1* e *pointerstackpointer2* são usadas neste procedimento. Elas apontam para o endereço da memória que contém o SP desejado, ou onde o valor atual do SP deve ser carregado. Em geral elas apontam para as variáveis *stackpointerbeggining* e *currentstackpointer*, que indicam onde a Stack de um thread começa, e em que posição está atualmente.

Deseja-se botar o valor desejado do PC na Stack. A *functionpointer* deve conter

o endereço da função que o thread executará. Um código em Assembly passa este valor para um registrador, que depois é adicionado na Stack. Quando um contexto for restaurado (pelas instruções *"ret"* ou *"reti"*) estes valor na Stack será enviado ao PC. Este código é usado na criação de threads.

A classe Kernel mantém um relógio interno indicando o tempo em segundos/microsegundos desde que o Dispositivo foi iniciado (*timeseconds* e *timemicroseconds*). Sempre que uma das duas interrupções ocorrerem o relógio será atualizado. As variáveis são usadas para calcular quando o thread irá acordar (thread dormindo) ou executar (próximo thread no RR). Após o cálculo um destes valores é escolhido para o tempo do próximo Timer. O valor escolhido é convertido no número de ticks (*timercountermax*) e no número de overflows (*timercounteroverflowmax*) do Timer do RR. Quando uma interrupção de Overflow ocorre a variável *timercounteroverflow* (que começa em *timercounteroverflowmax*) decai em 1. Quando chega a 0 a interrupção pelo número de ticks do Timer é acionada.

A partir de agora as funções do *Kernel* serão tratadas. A explicação é direta pois basta fazer referência às etapa de execução do Kernel já apresentadas. Note que até o momento nenhuma função de classe foi explicada. Isto ocorre porque existe uma classe principal, uma casca que pode ser acessada pelos threads e que possui todas as funções importantes. O código foi inspirado no MOS e ajudou a Arquitetura de Software do MOS ser desenvolvida.

A função *Start()* é chamada no *main()* do programa. O *Start()* consiste nas etapas Configurar Interrupções de Timer, Configurar Threads, RR: Ordenação e RR: Executar Primeiro Thread. A função *ThreadCreate()* percorre a etapa Início ThreadCreate até a Fim ThreadCreate.

A função *ThreadSleepMilliseconds()* chama *ThreadSleepMicroseconds()*. A *ThreadSleepMicroseconds()* é a própria *ThreadSleep()*, e percorre da etapa Início ThreadSleep até a Fim ThreadSleep.

A *HandleTimer()* percorre as etapas do Início da Interrupção de Compare do Timer até o Fim da Interrupção de Compare do Timer, e a *HandleTimerOverflow()* corresponde à etapa Interrupção de Overflow do Timer. Note que *TIMER1_COMPA_vect*, *TIMER1_OVF_vect*, *SaveContext()* e *RestoreContext()* estão associadas às estas duas funções, portanto também estão associados às etapas mencionadas.

4.3.3 MOSBuilder

Na transferência de um CMod pela MOSNetwork uma seção da EEPROM é alocada temporariamente. Quando os bytes chegam pela MOSNetwork são armazenados nesta seção. Após o fim da transferência (CMod transferido por completo no MOSMicro), uma função especial é invocada. Ela encontra um espaço livre na memória

Flash onde as instruções de máquina do CMod ficarão. Depois, os bytes das instruções de máquina (primeiro setor do CMod) serão varridas para corrigir o endereço de chamada de subrotinas (usando o terceiro setor do CMod), permitindo que funções chamadas da ModClass apontem para o endereço correto da memória Flash. Por último, o primeiro setor do CMod é copiado da EEPROM para a memória Flash, para de fato tornar o CMod em um RMod.

A operação de escrita na memória Flash é feita com a instrução SPM que só pode ser executada dentro da Boot Loader Section. A implementação do MOS pode variar, mas a passagem das instruções de máquina de um CMod da EEPROM para a Flash provavelmente acarretaria em uma suspensão temporária do MOS. Este é um grande problema de implementação que talvez não tenha solução, mas não são será mais estudada na dissertação.

4.3.4 Observações

A classe *Kernel* usada como exemplo na explicação do Kernel para o ATmega2560 faz parte do próprio *ModuleOSATmega2560*. As mesmas variáveis e funções poderiam seriam usadas. Obviamente a classe *Kernel* ainda é muito rudimentar. Não implementa a prioridade no Round Robin nem o outro algoritmo de escalonamento FIFO. O Escalonador não leva em conta as formas de bloqueio de threads pelos objetos de sincronização (Mutex, Monitor, etc).

Como já foi visto os MOSMicros adicionam possibilidades para o programador nas BModClasses ao permitir o uso dos recursos exclusivos de microcontroladores. Citamos apenas a criação da rede RS232 e os recursos presentes no ATmega2560. No entanto o MOS deve ser implementando em uma gama de microcontroladores deixando o mecanismo de derivação em BModClasses verdadeiramente rico.

Um objeto da classe *ModuleOSATmega2560* deve ser global, fora da função *main()*. Isto é necessário pois manobras envolvendo código em Assembly (Escalonador) e este objeto global são usados na implementação dos threads. Após a compilação e carregamento na memória Flash (Application Section e/ou Bootlader Section), a classe será construída antes da função *main()* ser executada. Um pedaço da memória SRAM será preenchida como o objeto *ModuleOSATmega2560*. A função *main()* chama a função *Start()* deste objeto para de fato inciar o MOS, e esta função só retorna quando o MOS for desligado.

O MOSMicro não deve usar Heap pois todo o Free Space no microcontrolador é usado como Stack de threads. Por isto, o *ModuleOSATmega2560* deve criar várias objetos de forma estática, como arrays de tamanho definido. Isto é equivalente ao explicado na implementação do MOS em SOs, o que leva a um número fixo de *Thread*, *RMod*, *ModIn* etc. O número máximo de cada tipo é determinado com

defines da linguagem C++. Quanto maior o número destes objetos, maior será o tamanho da classe *ModuleOSATmega2560* na SRAM. O objeto global da classe *ModuleOSATmega2560* ocupa um espaço na SRAM, diminuindo a área destinada às Stacks dos threads. O que mais influencia o tamanho reservado para a Stack dos threads é o número de ModIns que podem ser criadas o número de threads que podem ser criados no Free Space.

Para uma ModIn ser criada é preciso alocar um espaço que conterá o objeto de sua ModClass. O endereço desse espaço é passado para threads de uma ModIn quando criados, permitindo que o thread execute a função da ModClass associada. Nos MOSMicros este espaço por thread é fixo e determinado antes da compilação. Este é um array da *ModuleOSATmega2560*. Se o MOSMicro aceitar apenas uma ModIn de 50 bytes usa-se apenas 50 bytes na Application Section, mas se aceitar 10 threads de 100 bytes será 1KB a menos no Free Space.

O ATmega2560 possui 8KB de SRAM. Suponha que 1KB é destinado ao Boot Loader Section, portanto a Application Section possui 7KB. Considere também que a até o Escalonador ser executado, 1KB está na Stack (*main()* e *ModuleOSATmega2560*). Sobram 6KB para serem divididos pelos threads. Seria possível ter 1 thread de 6000B, 2 de 3000B, 4 de 1500B, 8 de 750B, 16 de 375B. Esses são números realmente pequenos para Stack. O tamanho padrão de uma Stack no Windows é de 1MB e no Linux aproximadamente 8KB. É importante lembrar que o próprio MOS pode criar alguns threads para si. O MOS passa a responsabilidade para o programador. Ele deve medir quantos threads serão necessários para sua aplicação e inferir se a Stack será o suficiente para não gerar Stack Overflow.

Capítulo 5

Conclusões

A dissertação apresenta o Modular Operating System (MOS) e sua linguagem de programação MOSLanguage para ser usado em aplicações que envolvam microcontroladores e/ou múltiplas unidades de processamento. O sistema proposto é complexo e portanto de difícil análise. Para facilitar iremos tirar conclusões sobre o trabalho analisando os capítulos da dissertação.

Capítulo 1 A dissertação começa o capítulo 1 apresentando exemplos de aplicações que envolvam mais de uma PU, o servidor/cliente, os supercomputadores e a robótica. Depois são mostradas várias tecnologias/ferramentas que permitem um usuário acessar as PUs remotamente, e algumas destas ajudam os programadores a desenvolver as aplicações. A robótica possui a arquitetura mais complexa e acaba se tornando o foco da análise e ferramentas. Este tipo de aplicação envolve microcontroladores e vemos que as ferramentas analisadas não estão preparadas para lidar com eles.

Em seguida o capítulo desenvolve as primeiras idéias por trás do MOS. Ele é visto como um Framework e como um Sistema Operacional dependendo de onde é usado (em cima de SOs e de microcontroladores). No MOS o programador deve criar um projeto para sua aplicação. No projeto estarão definidos Classes, Modules e Protocols que são programados com a MOSLanguage. Os Modules são pequenos programas que serão executados nos dispositivos com MOS. Os Modules podem se comunicar dentro de um dispositivo. Os Protocols são usados para que um Module se comunique por um rede do dispositivo. O MOS também define a rede MOSNetwork que é aplicada sobre as outras redes do dispositivo e permite dispositivos MOS se comunicarem. O diferencial desta rede é que ela pode fazer os Modules serem compilados, transferidos e iniciados em dispositivo que não contém o Module.

Por último o capítulo compara o MOS com outras tecnologias para mostrar que ele tem um foco diferente das outras. O MOS propõe soluções concretas

para aplicações que envolvem microcontroladores.

Capítulo 2 Primeiramente todos os termos relacionados ao MOS são definidos. Isto é essencial para não gerar conflitos nos conceitos do MOS (ocorre na Introdução) para com isto explicar como ele funciona. Em seguida, o primeiro item a ser discutido é o Module. As vertentes do Module (ModClass, CMod, RMod e ModIn) são explicadas assim como o sistema de transferência de Modules.

O capítulo comenta sobre a proteção de memória no MOS. O MOS deve funcionar em microcontroladores e eles não tem proteção de memória. Explica-se como o MOS tira vantagem deste fato e quais são as desvantagens. Depois comenta-se sobre a falta de segurança no MOS relacionado a intrusos no sistema, e sobre as medidas adotadas pelo MOS para contornar a proteção de memória. Em seguida explica-se melhor a dualidade Framework/Sistema Operacional do MOS, e que é necessário criar um Kernel nos MOSMicros. Também comenta-se que na MOSLanguage existe o objeto global System (representa o MOS) que é usado pelos Modules para que interajam com o sistema.

O próximo assunto é o aspecto multiplataforma do MOS. A dissertação usa o Windows, Linux e ATmega2560 para mostrar como o MOS trata este aspecto. Os Modules podem ser vinculados a um MOSDevice para terem acesso às particularidades do dispositivo. Isto é melhor representado pelo acesso aos recursos de microcontroladores (ATmega2560) como portas analógicas. O funcionamento do MOSBuilder é detalhado, mostrando como um MOSDevice executa gera uma ModIn a partir de um CMod compilado em outro MOSDevice. O segredo está em gerar um código de máquina e tabelas indicando posições no código, e quando chegarem no dispositivo a tabela é usada para corrigir o código de máquina, ajustando ao dispositivo que irá executar.

O MOS tem várias limitações. Uma das formas de vencê-las é ligar o MOSDevice a programas externos. Estes programas incorporariam a MOSAPI que permite um programa em C++ se comunicar com um MOSProcess por uma comunicação IPC. Outra forma é adicionando no MOS Modules especiais que são feitos em C++ ao invés da MOSLanguage.

Os threads no MOS são criadas por ModIns, mas não pertencem à ModIn (relação diferente de threads e processos). ModIns podem estar conectadas. Quando duas ModIns são conectadas por um MOSLink, uma pode chamar funções ou acessar variáveis da outra. O thread em uma ModIn pode então entrar ou acessar outra ModIn, mesmo que temporariamente. O MOS também possui objetos de sincronização para lidar com a sincronização de threads no acesso de regiões críticas. A dissertação usa apenas o Mutex como exemplo. O trabalho volta ao assunto da conexão de Modules (MOSLink), e mostra que

existem duas formas de ModIns se comunicarem, com o MOSLinkDirect (threads entram e retornam, ou acessam ModIns conectadas) ou o MOSLinkMP (passagem de mensagens).

Em seguida o trabalho trata de redes no MOS. A proposta do MOS é criar uma abstração sobre as redes (Nets) de forma que o programador não se preocupe em qual rede vai passar os dados, e sim quais são os dados. As Nets consideradas são a RS232 e a TCP/IP. Como cada MOSDevice possui seus próprios NetTypes e formas de configurá-los, apenas os Modules vinculados podem criar redes. Qualquer Module pode se conectar a uma Net já criada. O protocolo de dados a serem usados em uma Net é chamado de Protocol e o programador tem liberdade para defini-lo como bem entender. Os Protocols não são usados diretamente nas Nets, existe o intermediário NetSlot. As ModClasses definem NetSlots de Nets e as associam com um Protocol.

Depois o capítulo explica o funcionamento da MOSNetwork. O MOS define o MOSNetProtocol, um Protocol escrito em C++ que pode ser adicionado nas Nets dos MOSDevices. O protocolo um byte de comando, e isto permite que dois dispositivos conectados por uma MOSNetwork interajam. Isto permite criar uma rede entre estes dispositivos que não tem topologia definida. A MOSNetwork necessita de um MOSMaster, um nó especial que gerencia a MOSNetwork. Um diferencial nesta rede é que Modules podem criar conexões com outros Modules usando a MOSNetwork. As mensagens são roteadas entre os dispositivos intermediários até chegarem no destino.

O capítulo finaliza analisando onde o MOS pode ser aplicado, complementando o capítulo 1. Neste capítulo podemos ver que os conceitos por trás do MOS estão bem estruturados e encadeados. Nele vemos quais são os principais atores do MOS, como eles são implementados e porque certas escolhas são feitas. Isto vai desde o baixo nível (instruções de máquina) até o alto nível (MOSLanguage).

Capítulo 3 Primeiramente o capítulo indica características desejadas para a linguagem a ser usada no MOS, como a linguagem ser compilável e não interpretada, ser orientada a objetos, multiplataforma, proibir alocação dinâmica de variáveis e permitir a criação dos elementos principais (Classes, Modules e Protocols). Com isto uma rápida análise é feita e conclue-se que nenhum linguagem de programação se encaixa nos requisitos. A linguagem C++ é a que chega mais perto mas ainda precisaria ser modificada. Portanto, a MOSLanguage deve ser definida.

A MOSLanguage define *Data Types*, tipos básicos de dados (bool, int, float) como em muitas outras linguagens. A linguagem segua parcialmente a orien-

tação a objetos. *Classes* podem ser criadas com atributos e métodos e podem ser derivadas. A linguagem define um *Module* que é uma *Class* com capacidades extras e representa um pequeno programa (são as ModIns). Os *Modules* podem ter objetos de outros *Modules* que devem ser explicitamente conectados antes de serem usados. *Modules* também podem definir mensagens que quando enviadas podem ser tratadas por seus modules conectados. *Modules* podem ser vinculados a um dispositivos e com isto utilizar recursos específicos do dispositivo. *Protocols* também são definidos mas estes devem ser construídos com ajuda de ferramentas gráficas. Os *MOSPojects* são projetos que agrupam *Classes*, *Modules* e *Protocols*.

Em seguida o capítulo trata de classes especiais definidas na linguagem, os *Time*, *Thread*, objetos de sincronização (*Mutex*), *Module*, *NetSlot* e *System*. Note que vários destes estão relacionados ao *System* que representa o Sistema.

A MOSLanguage descrita segue os requisitos discutidos no início do capítulo. Mesmo com as limitações impostas, a linguagem fornece várias facilidades para o programador para que ele use redes, crie Modules, conecte Modules e etc.

Capítulo 4 O capítulo trata da arquitetura de software em C++ para a implementação do MOS. Esta arquitetura leva em conta o fato do MOS ser multiplataforma, portanto o procedimento padrão é definir classe genéricas que depois são derivadas para classes específicas dos dispositivos. A biblioteca Boost é muito usada portanto ao invés de derivar direto para Windows ou Linux deriva-se para o Boost. Para facilitar as classes são divididas em grupos (GScheduler, GThreads, GSyncObjects, GLink, GNetwork, GModule e GModuleOS). O GModuleOS descreve as classes que representam o próprio MOS enquanto os outros grupos são em geral recipientes de dados, e os recipientes são usados pelo GModuleOS. Inúmeras funções do GModuleOS estão disponíveis na MOSLanguage.

O GScheduler define classes que contêm configurações dos Escalonadores de cada MOSDeviceType. O GThread contém configurações dos thread para cada Escalonador de cada MOSDeviceType. No GModuleOS existem funções para configurar os escalonadores e outras para criar/destruir threads.

Os GSyncObjects são exemplificados com o Mutex. As classes definem as funções de Lock() e Unlock(). Funções do GModuleOS alocam/desalocam Mutexes para serem usados nos Modules.

O GLink contém classes que representam o MOSLink entre Modules. O MOSLink sempre sabe quem são os LinkEnd e o LinkOwner (ModIns). O LinkSlot também é definido e fica dentro das ModIns ao invés do MOS. Existe uma

série de funções nas classes para fazer a conexão entre Modules.

O GModule descreve classes para representar a ModIn e o RMod. A classe do RMod contém uma série de variáveis para representar um Module como um número de Mutex, threads, LinkSlots, nome, número da versão etc. As funções do GModuleOS estão relacionadas a criação/destruição/busca de ModIns e RMods.

O GNetwork define as classes de Net para os MOSDevices. As classes são genéricas e poderiam ser usadas para representar qualquer Net nos MOSDevices. Funções do GModuleOS estão relacionados à busca de Nets já configuradas no sistema, à adição de NetSlots em Nets e à criação de Nets.

O capítulo finaliza a análise dos grupos de classes com o que falta no GModuleOS, que são basicamente funções relacionadas ao tempo (descobrir o tempo atual do sistema e fazer threads dormirem por certo tempo) e ao uso do hardware. Neste último lidamos apenas com o ATmega2560 pois é este que contém periféricos definidos.

Depois de mostrar a arquitetura o capítulo muda o foco para indicar as peculiaridades da implementação nos MOSProcess e MOSMicros. Nos MOSProcess apresenta-se o funcionamento da memória de Sistemas Operacionais para depois mostrar que os MOSProcess podem alocar dinamicamente algumas coisas como ModIns. Quando os MOSBuilder compilam um Module para MOSProcesses na realidade devem compilá-lo como uma biblioteca dinâmica para depois ser alterado e carregado no MOSDevice de destino.

A implementação no MOSMicro é mais complexa o MOS define um Kernel. Primeiro mostra-se que a memória RAM é dividida de duas formas, uma em relação à permissão de gravação e outra nas seções Boot/Application. Depois comenta-se como as interrupções agem na memória e nos registradores. A explicação de como criar um Kernel para microcontroladores é relativamente complexa, mas cobre-se o uso de interrupções, etapas de execução do Kernel e classes/funções relacionadas. Em seguida vê-se como o MOSBuilder lida com a EEPROM para guardar o CMod antes de ser transformado em CMod.

Se comparado com outros frameworks vemos que o MOS possui uma série de desvantagens por escolher a compatibilidade entre dispositivos em prol dos recursos atuais. A MOSLanguage não permite alocação dinâmica. Como ela está sendo definida não possui nenhuma biblioteca para ajudar os programadores. No entanto podemos ver que o MOS possui um nicho em que é muito útil. A maior utilidade do MOS está em aplicações que envolvam a comunicação entre microcontroladores e computadores. No caso de aplicações que envolvam apenas microcontroladores

o MOS é um sistema diferente das soluções atuais por permitir a transferência de programas pela rede além de um micro kernel e acesso a redes.

Podemos ver que a dissertação tenta ser o mais abrangente o possível sobre como o MOS deve funcionar internamente e tenta explicar o porque das escolhas feitas. Isto vale tanto para como a MOSLanguage funciona quanto para a arquitetura de software.

Os programadores devem ser capazes de programar mais rápidos usando a MOSLanguage e devem ter menos necessidade de interação com o hardware com a MOSNetwork funcionando. Também devem ser capazes de reconstruir o protocolo de dispositivos usados na aplicação para usá-los com facilidade. Em suma, o MOS tenta facilitar o desenvolvimento de aplicações (sobretudo na robótica) diminuindo a carga de trabalho que a programação tem na aplicação.

5.1 Trabalhos Futuros

Implementar a proposta da Dissertação O primeiro passo é sem dúvida implementar o MOS como proposto na dissertação, ao menos o mínimo necessário. O item mais complexo a ser implementado é o compilador da MOSLanguage.

IDE Para programadores utilizarem o MOS é preciso desenvolver uma IDE para o MOS. Além da programação, isto permitirá acessar os MOSDevices visualmente, criando/destruindo Modules. Os NativeModules poderiam então fazer parte da IDE, como um console, uma ferramenta para gerar gráficos, visualizador 3D etc.

Respositório público O MOS é um projeto ambicioso e que não pode ser feito por uma única pessoa. Ele ser desenvolvido por uma comunidade. Após uma implementação básica do MOS e documentos detalhando sua estrutura, o código deve ser movido para um repositório público.

Outras Redes Existem algumas outras redes comuns que devem ser adicionadas no MOS como CAN, SPI, I2C. Também é preciso criar uma rede baseada na comunicação IPC entre processos de um SO, permitindo que a MOSAPI seja desenvolvida.

Expansão para outros dispositivos e arquiteturas Não basta o MOS ser desenvolvido para apenas um microcontrolador (ATmega2560), ele deve ser implementado em outros. É neste momento que os requisitos mínimos do MOS poderão ser gerados. Como foi dito anteriormente o MOS deve ser adaptado para a tecnologia ARM que representa um passo adiante em relação aos microcontroladores.

Referências Bibliográficas

[1] SANGKET ET AL. BMC BIOINFORMATICS. "Computer Cluster Architecture". jun. 2013. Disponível em: <http://www.biomedcentral.com/1471-2105/11/217/figure/F1?highres=y>.

[2] GOSTAI. "Integrate URBI". jun. 2013. Disponível em: <http://www.gostai.com/static/img/products/integrateurbi.jpgêxtgreater .

[3] DUARTE, G. "Anatomy of a Program in Memory". jun. 2013. Disponível em: <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>.

[4] MICROSOFT. "Scheduling priorities". jun. 2013. Disponível em: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx>.

[5] DONAHOO, M. J., CALVERT, K. L. *TCP/IP Sockets in C: Practical Guide for Programmers*. 2 ed. , Elsevier Inc., 2009.

[6] FALL, K. R., STEVENS, W. R. *TCP/IP Illustrated, Volume 1: The Protocols*. 2 ed. , Pearson Education Inc., may 2012.

[7] JOSEPH, J., FELLENSTEIN, C. *Grid Computing*. Pearson Education Inc., 2004.

[8] PHILIPS SEMICONDUCTORS. *The I2C-Bus Specification*. Relatório Técnico version 2.1, nov 2003.

[9] MOTOROLA INC. *SPI Block Guide*. Relatório Técnico S12SPIV3/D V03.6, feb 2003.

[10] MONMASSON, E. *Power Electronic Converters: PWM Strategies and Current Control Techniques*. ISTE Ltd., John Wiley And Sons Inc., 2011.

[11] AXELSON, J. *Serial Port Complete: Programming and Circuits for RS-232 and RS-485 Links and Networks*. Lakeview Research, 2000.

[12] ZENG, H., GIUSTO, P., GHOSAL, A. *Understanding and Using the Controller Area Network Communication Protocol Theory and Practice*. 2 ed. , Springer Science+Business Media LLC, 2012.

[13] BARRETT, S. F., PACK, D. J. *Atmel AVR Microcontroller Primer: Programming and Interfacing*. 2 ed. , Morgan And Claypool, 2012.

[14] TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 3 ed. , Pearson Education do Brasil, 2010.

[15] BARRETT, D., SILVERMAN, R., BYRNES, R. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly Media, Inc., maio 2005.

[16] NAD EWING LUSK, W. G., SKJELLUM, A. *Using MPI: Portable Parallel Programming with the Message-passing Interface, Volume 1*. Massachusetts Institute of Technology, 1999.

[17] GEIST, A. *PVM: Parallel Virtual Machine :a Users' Guide and Tutorial for Networked Parallel Computing*. 5 ed. , Massachusetts Institute of Technology, 1994.

[18] GALLI, D. L. *Distributed operating systems: concepts and practice*. Prentice Hall, 2000.

[19] BUYYA, R., BROBERG, J., GOSCINSK, A. M. *Cloud Computing: Principles and Paradigms*. John Wiley And Sons, dec 2010.

[20] HILDEBRAND, D. "An Architectural Overview ofQNX", *Usenix Workshop on Micro-Kernels And Other Kernel Architectures*, abr. 1992.

[21] NAGEL, W. "Soft Real-Time Programming with Linux". nov. 2005. Disponível em: <www.drdobbs.com/soft-real-time-programming-with-linux/184402031>.

[22] HENRIQUES, L. "Threaded IRQs on Linux PREEMPT-RT", *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2009.

[23] FURR, S. "What is Real Time and Why Do I Need It?" jun. 2013. Disponível em: <http://www.qnx.com/developers/articles/article_298_1.html̂xtgreater .

[24] GERUM, P. "Xenomai - Implementing a RTOS emulation framework on GNU/-Linux", *Free Software Foundation*, 2004.

[25] GERUM, P. “Pasi Sarolahti”, *Research seminar on Real-Time Linux and Java*, 2001.

[26] BAILLIE, J.-C., NOTTALE, M., POTHIER, B. *URBI Tutorial for Urbi 1.0*. Gostai, 2006.

[27] SONY. *ERS-7M2 Entertainment Robot AIBO User's Guide Basic*. Relatório técnico, 1999.

[28] QUIGLEY, M., GERKEY, B., CONLEY, K., et al. “ROS: an open-source Robot Operating System”, *ICRA Workshop on Open Source Software*, 2009.

[29] GOEBEL, R. P. *ROS By Example - Volume 1*. abr. 2013.

[30] WILLOW GARAGE. “Documentation - ROS Wiki”. jun. 2013. Disponível em: <http://wiki.ros.org/>.

[31] BARRY, R. *Using the FreeRTOS Real Time Kernel: a Practical Guide*. 2009.

[32] BOVET, D. P., CESATI, M. *Understanding the Linux Kernel*. 3 ed. , O'Reilly, nov 2005.

[33] ABRAHAM, R., MARSDEN, J. E., RATIU, T. *Using OpenMP: Portable Shared Memory Parallel Programming*. 10 ed. , MIT Press, 2008.

[34] CAVANAGH, J. *X86 Assembly Language and C Fundamentals*. CRC Press, jan 2013.

[35] ATMEL CORPORATION. *ATmega640/1280/1281/2560/2561*. Relatório técnico, 2012.

[36] ARDUINO. “Arduino - ArduinoBoardMega2560”. jun. 2013. Disponível em: <http://arduino.cc/en/Main/arduinoBoardMega2560>.

[37] FLYNN, M. J., LUK, W. *Computer System Design: System-on-Chip*. John Wiley And Sons, aug 2011.

[38] SCHILDT. *Mfc Programming From The Ground Up*. McGraw-Hill Education (India) Pvt Limited, jan. 2000.

[39] SMART, J., WITH, K. H., CSOMOR, S. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall Professional, jul. 2005.

[40] DIGIA. “Qt”. jun. 2013. Disponível em: <qt.digia.com>.

[41] TANENBAUM, A. S., WOODHULL, A. S. *Operating Systems Design and Implementation*. 3 ed. , Prentice Hall, jan 2006.

[42] KARLSSON, B. *Beyond the C++ Standard Library: An Introduction to Boost*. Pearson Education Inc., 2006.

[43] SRIKANT, Y., SHANKAR, P. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 2 ed. , CRC Press, 2007.

[44] RICARTE, I. L. M. *IntroduÃ§Ã£o Ã CompilaÃ§Ã£o*. Elsevier Editora Ltda, 2008.

[45] VON HAGEN, W. *The Definitive Guide to GCC*. Apress, ago. 2006.

[46] AKYILDIZ, I., WANG, X. *Wireless Mesh Networks*. John Wiley And Sons, abr. 2009.

[47] SINGH, J., KAUR, P. P. *Object Oriented Programming Using C++*. Technical Publications, jan. 2008.

[48] NETWORK WORKING GROUP. *UTF-8, a transformation format of ISO 10646*. Relatório Técnico RFC 3629, The Internet Society, nov. 2003.

[49] EHRMAN, J. R. "Logical arithmetic on computers with two's complement binary arithmetic", *Communications of the ACM*, 1968.

[50] 754R WORKING GROUP. *IEEE Standard for Floating-Point Arithmetic*. Relatório Técnico IEEE 754-2008, Institute of Electrical and Electronics Engineers, ago. 2008.

[51] MATLAB. *version 7.10.0 (R2010a)*. Natick, Massachusetts, The MathWorks Inc., 2010.

[52] SARKAR, C. S. "Scheduling in Linux". jun. 2013. Disponível em: <`http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/`>.

[53] KERRISK, M. "sched_setscheduler(2) - Linux manual page". jun. 2013. Disponível em: <`http://man7.org/linux/man-pages/man2/sched_setscheduler.2.html`>.

[54] RICHARD, B. *MULTITASKING ON AN AVR - EXAMPLE C IMPLEMENTATION OF A MULTITASKING KERNEL FOR THE AVR*. Relatório técnico, 2004.

[55] “AVR Libc Home Page”. jun. 2013. Disponível em: <http://www.nongnu.org/avr-libc/>.

[56] ATMEL CORPORATION. *AVR109: Self Programming*. Relatório técnico, jun. 2004.

[57] “avr-gcc - GCC Wiki”. jun. 2023. Disponível em: <http://gcc.gnu.org/wiki/avr-gcc>.

[58] ATMEL CORPORATION. “Atmel Studio”. jun. 2013. Disponível em: <http://www.atmel.com/tools/atmelstudio.aspx>.