Capítulo
1

Software Embarcado

Paulo Cézar Stadzisz e Douglas Paulo Bertrand Renaux

Universidade Tecnológica Federal do Paraná (UTFPR)
Av. Sete de Setembro, 3165, Curitiba-PR, CEP: 80.230-901
stadzisz@utfpr.edu.br - douglasrenaux@utfpr.edu.br

Abstract

The embedded system area represents a very large market with some billions units produced every year including 8 to 64-bits microcontrollers. Historically, embedded systems were programmed by hardware designer, since only they understood the details of the electronic circuits they designed. The increasing complexity of embedded software implies a corresponding increase in the need for using software engineering process and techniques for the development of embedded systems. In this sense, this document presents a general overview about embedded systems and discusses the specificities of embedded software development.

Resumo

A área de sistemas embarcados representa um mercado imenso, da ordem de alguns bilhões de unidades produzidas anualmente, envolvendo microcontroladores de 8 a 64 bits. Historicamente, o desenvolvimento de software para sistemas embarcado era conduzido por técnicos e engenheiros em eletrônica uma vez que apenas eles eram capazes de compreender as especificidades dos circuitos por eles projetados. Com o aumento da complexidade do software embarcado, passou a haver uma necessidade crescente de aplicação de processos e técnicas de engenharia de software no desenvolvimento de sistemas embarcados. Neste sentido, este documento apresenta uma visão geral de sistemas embarcados e discute especificidades do desenvolvimento de software embarcado.

1.1. Introdução

Este texto aborda uma categoria de software denominada *embedded software*, termo que pode ser traduzido para o português como **software embarcado** ou, ainda, software embutido. Esta categoria de software guarda muitas similaridades com software utilizado em sistemas de informação em geral, porém possui também diversas especificidades que serão discutidas ao longo deste texto.

Software embarcado não é meramente um software que é executado em um computador pequeno. Os conceitos envolvidos, as características de seu ambiente e os princípios de concepção impõem uma visão própria de computação em software embarcado.

No que diz respeito ao ambiente de execução, software embarcado executa em máquinas que não são verdadeiramente computadores de uso pessoal, como os PCs. Esta categoria de software é executada em sistemas microprocessados dentro de aviões, máquinas industriais, equipamentos de automação comercial e bancária, brinquedos, telefones, robôs e embarcações, entre outros. A expressão "embarcado" vem exatamente do fato do software embarcado ser executado sobre um sistema microprocessado interno, ou seja, embarcado em um equipamento, máquina ou sistema maior.

Pelos exemplos apontados, pode-se perceber que uma das características mais determinantes de software embarcado é a heterogeneidade do ambiente no qual pode ser empregado. Cada projeto de software embarcado pode envolver características de ambiente diferentes em razão dos propósitos de seu uso e da natureza de suas operações. Este aspecto é flagrantemente próprio da área de software embarcado e se opõe claramente ao ambiente padronizado e previsível empregado em sistemas de informação típicos.

Um segundo aspecto importante está relacionado com o nível de abstração em software embarcado. Enquanto a ciência da computação tem sistematicamente abstraído o mundo real de forma a reduzir sua complexidade, o desenvolvimento de software embarcado está intrinsecamente engajado com interações com o mundo real [Lee, 2002]. Considerando esta interação, é fácil entender por que a grande maioria dos desenvolvedores de software embarcado não são engenheiros com formação em ciência da computação. O desenvolvedor de software embarcado é, frequentemente, um especialista, ou seja, aquele que melhor entende determinado domínio do mundo real [Lee, 2002].

Estas especificidades de software embarcado fazem com que muitos dos avanços em ciência da computação, como os altos níveis de abstração, não possam ser aplicados diretamente nesta área. Por outro lado, a evolução dos requisitos observada nos últimos anos tem elevado drasticamente a complexidade do software embarcado. Como exemplo, pode-se citar a expectativa de que, mesmo equipamentos menores, disponham de canais de comunicação com a internet e suporte para comunicações *wireless*. Este tipo de exigência amplia a complexidade do software embarcado mantendo os requisitos de confiabilidade impostos a estes sistemas.

A área de aplicação de software embarcado é ampla tanto em termos da diversidade de empregos quanto em termos de escala de mercado. A ordem de grandeza deste mercado é de alguns bilhões de unidades anuais, superando as poucas centenas de milhões de microcomputadores comercializados anualmente no mundo. Esta área apresenta,

também, índices de crescimento anuais superiores, por exemplo, aos da área de sistemas de informação.

Deve-se considerar também os aspectos estratégicos associados à área de software embarcado. O software embarcado representa, muitas vezes, a parte central de produtos de alta tecnologia onde reside grande parte do *know-how* envolvido. Assim, o desenvolvimento da indústria de software embarcado tem um impacto importante do ponto de vista econômico, de geração de empregos e na redução das dependências de importações.

O objetivo deste documento é apresentar uma revisão ampla dos principais conceitos, características, técnicas e processos associados com o desenvolvimento de sistemas embarcados. O texto discute também as tendências atuais observadas nesta área.

1.2. Visão Geral de Sistemas Embarcados

O software embarcado é um dos componentes dos **sistemas embarcados** (*embedded systems*), também denominados sistemas computacionais embarcados (*embedded computer systems*). A maioria das características do software embarcado advém das peculiaridades do sistema embarcado no qual se insere.

Assim, antes de um aprofundamento sobre software embarcado, será apresentada nesta seção uma visão geral de sistemas embarcados descrevendo conceitos, características, visão de mercado e aspectos de hardware.

1.2.1. Conceituação e Características de Sistemas Embarcados

1.2.1.1. Definições

Um sistema embarcado é uma combinação de hardware e software, e eventuais componentes mecânicos, projetados para executar uma função dedicada [Wolf, 2005]. Em muitos casos, sistemas embarcados são parte de um sistema, produto ou dispositivo maior. O conceito de sistema embarcado está, portanto, diretamente ligado com dispositivos eletrônicos de propósitos específicos, se contrapondo às tendências de forte padronização típicas de computadores de propósito geral.

Em um sistema embarcado, o hardware e software estão intimamente relacionados de forma que o software embarcado interage com o hardware que foi especificamente projetado para interagir com ele [Lee, 2002].

1.2.1.2. Heterogeneidade

A heterogeneidade é um aspecto intrínseco de sistemas embarcados e afeta tanto a eletrônica quanto o software embarcado. Cada projeto de sistema embarcado pode envolver o desenvolvimento de um hardware único, ou a customização de um hardware de referência existente. As necessidades em termos de desempenho, de interfaces e de memória, entre outras, requerem soluções especializadas para cada problema. Tipicamente, soluções de hardware de prateleira não são apropriadas em razão das restrições de projeto, limitando-se a usos nos quais se tem pouca escala e onde uma customização torna-se inviável.

Com relação ao hardware, um grande elemento de diferenciação dos sistemas embarcados é a capacidade de processamento. Sistemas embarcados podem empregar microcontroladores com palavras de 8, 16, 32 e 64 bits, além de DSPs (*Digital Signal Processors* – Processadores Digitais de Sinal) e FPGAs (*Field Programmable Gate Array* – Dispositivos Lógicos Programáveis), e a escolha por uma destas categorias determina a capacidade de processamento disponível e, por conseqüência, o custo, o porte e a natureza de suas aplicações. Deve-se considerar também que, dentro destas categorias, há ainda variações de freqüência de *clock*, de arquitetura (ARM, X86, MIPS, SH, entre outros), de periferia integrada e de fornecedores (NXP, Intel, Atmel, Freescale, ST, National, entre outros).

Considerando a íntima relação com o hardware e a diversidade de usos de sistemas embarcados, o software embarcado também apresenta grande heterogeneidade tanto em termos funcionais (eventos regulares e irregulares, variações nas tolerâncias temporais, atividades seqüenciais e concorrentes, reatividade, diversidade de *device drivers* para interfaces e periféricos, diversidade de protocolos de comunicação, entre outros) quanto em termos de ambientes de desenvolvimento (diversidade de linguagens, compiladores, depuradores, ferramentas de *trace*, canais de *debug*).

A heterogeneidade em sistemas embarcados é uma conseqüência da diversidade de necessidades, da diversidade de tecnologias e das restrições envolvidas. Este é um grande fator complicador para o desenvolvimento de sistemas embarcados, pois as premissas de padronização, como as encontradas para sistemas baseados em computadores pessoais, não se aplicam nesta área.

1.2.1.3. Restrições

Ao contrário do ambiente dos computadores pessoais nos quais se supõe disponíveis uma grande capacidade de processamento, um sistema operacional que gerencia todos os periféricos e interfaces, grande quantidade de memória volátil e espaço em disco, entre outros recursos, os sistemas embarcados se caracterizam pela escassez de recursos além de outras restrições. Algumas das restrições mais comuns são relacionadas a seguir. Deve-se notar, entretanto, que estas restrições podem ser mais ou menos severas conforme as especificidades de cada sistema embarcado.

- **Restrições de Custos**

  O desenvolvimento de sistemas embarcados sofre grandes pressões por redução de custos, pois muitos dos produtos que integram sistemas embarcados são produtos eletrônicos de consumo. Estes produtos são fabricados em larga escala e estão sujeitos a uma grande concorrência. Outros produtos não têm grandes escalas de produção, mas pela especificidade de suas funções sofrem igualmente fortes pressões para redução de custos. Assim, um objetivo comum é desenvolver um produto que sejam o mais "enxuto" possível em termos de componentes de hardware e software para se atingir os menores patamares de custo alcançáveis.

  Estas pressões por baixos custos desencadeia uma série de restrições sobre o desenvolvimento dos sistemas embarcados de forma a reduzir os custos de cada componentes do sistema.

- **Restrições de Dimensionais**

Vários modelos de produtos que integram sistemas embarcados têm severas limitações de dimensão, pois são produtos portáteis ou produtos que são conectados a sistemas maiores com limitações de dimensão. Produtos como um telefone celular ou um PDA (*Personal Digital Assistant*) precisam ter dimensões suficientemente reduzidas para que seu uso seja prático e ergonômico. De forma semelhante, estes produtos precisam ter pouco peso e, mesmo assim, dispor de uma estrutura mecânica suficiente para o tipo de manipulação ao qual se prestam. Estas limitações de dimensão e peso dos produtos se propagam para o sistema embarcado de forma que a concepção da parte eletrônica, em especial, sofre grande restrição.

- **Restrições de Consumo e Autonomia**

  Vários produtos que integram sistemas embarcados são equipamentos eletrônicos móveis como um telefone celular ou um aparelho de localização por GPS (*Global Positionning System*). Estes equipamentos são alimentados por baterias e têm, portanto, uma autonomia limitada de funcionamento. Uma das exigências neste caso é que o sistema embarcado tenha um consumo mínimo de forma que a autonomia do produto seja a máxima alcançável. Esta exigência impõe soluções que operem sob menor freqüência de relógio (*clock)*, que possuam um menor número de componentes eletrônicos a serem alimentados e que empreguem um sistema de gerenciamento de energia. Mesmo para equipamentos que não sejam alimentados por baterias, o consumo continua sendo um aspecto importante, pois além de representar um aspecto econômico e ecológico do produto, afeta a dissipação de calor produzida pelo equipamento.

  As restrições de consumo podem ser críticas a ponto de interferir na própria lógica de construção dos programas embarcados. O consumo de energia para a execução de um programa varia conforme as operações realizadas, pois há uma variação de consumo de instrução para instrução. Tipicamente, as operações que implicam maior consumo de energia estão relacionadas com o sistema de memória. As transferências em memória são as operações mais caras em termos de consumo (cerca de 33 vezes o consumo de uma operação de adição, por exemplo [Wolf, 2005]). Já as operações de acesso a registradores são as mais eficientes em termos de consumo. Operações de acesso à *cache* são mais eficientes em consumo do que as de acesso à memória [Wolf, 2005]. Assim, o desenvolvimento de software considerando uma organização apropriada de instruções e dados em memória permitem otimizações de consumo do sistema embarcado.

- **Restrições de Recursos**

  Como um resultado principalmente das restrições de custo, de dimensões e de consumo, o desenvolvimento de sistemas embarcados deve se limitar aos mínimos recursos possíveis para atendimento dos requisitos funcionais impostos. Severas limitações de memória volátil e persistente, limitações de capacidade de processamento, limitações de interfaces, assim como limitações de suporte de software, são condições encontradas comumente no projeto deste tipo de sistema.

- **Restrições Temporais**

  Em muitos sistemas embarcados, as operações realizadas não só possuem requisitos funcionais a serem atendidos, mas estas operações estão condicionadas a serem

realizadas dentro de certos limites de tempo. Um limite de tempo definido para uma operação ou funcionalidade do sistema é dito uma restrição temporal. As restrições temporais podem ser mais ou menos severas conforme a natureza da operação. Os sistemas computacionais que operam sob restrições temporais são denominados sistemas em tempo real (*real-time systems*).

Restrições temporais são um elemento complicador no desenvolvimento de sistemas embarcados, pois tanto o projeto quanto os testes para garantia de que as restrições são atendidas são bem mais complexos do que projetos que envolvam apenas requisitos funcionais.

1.2.1.4 Complexidade

O desenvolvimento de sistemas embarcados é uma atividade que envolve tipicamente algum grau de complexidade. Esta complexidade pode alcançar níveis muito altos, por exemplo, em de sistemas críticos como equipamentos médicos e de aviônica. A complexidade de desenvolvimento de sistemas embarcados se deve às questões já apresentadas como heterogeneidade e restrições de projeto, mas envolvem também outros aspectos como a necessidade de reatividade do sistema e a complexidade intrínseca das tarefas.

Um importante elemento de complexidade em sistemas embarcados é a necessidade de interação com o mundo real. É comum que o sistema embarcado esteja conectado a diversos dispositivos externos, muitos deles específicos do produto em questão, como sensores, atuadores, dispositivos de captura de sinais, dispositivos de comunicação, dispositivos especiais de interação com o usuário e dispositivos de armazenamento. Cada dispositivo requer protocolos de comunicação e rotinas de controle próprios, muitos deles também específicos para o produto desenvolvido. O desenvolvimento de módulos de software para implementar estes protocolos e rotinas de controle envolve uma programação de mais baixo nível e de maior complexidade. Além disso, a interação com o mundo real impõe requisitos de reatividade do sistema embarcado, de forma que possam ser produzidas com prontidão respostas aos eventos capturados do mundo real. Esta necessidade de reatividade implica algoritmos mais complexos e, freqüentemente, restrições temporais.

Outra conseqüência da forte interação com o mundo real e da necessidade de reatividade é a necessidade de desenvolvimento de aplicações com funcionalidades executadas simultaneamente através de múltiplos processos e tarefas concorrentes. A complexidade de projeto, programação e teste de aplicações concorrentes aumenta consideravelmente, se comparado com aplicações seqüenciais ou orientadas a eventos.

1.2.1.4 Prazos e Riscos de Desenvolvimento

O desenvolvimento de sistemas embarcados é limitado por prazos fixados de acordo com o cronograma de desenvolvimento dos produtos que eles integram. Muitos destes produtos representam novidades ou avanços tecnológicos para seus consumidores e estão sujeitos a uma concorrência crescente. As janelas de oportunidade são estreitas, refletindo em prazos limitados para desenvolvimento dos produtos e, por conseqüência,

dos sistemas embarcados. Prazos limitados para desenvolvimento de produtos com novas tecnologias, aumentam os riscos de projeto e implementação das soluções.

1.2.4. Arquiteturas de Sistemas Embarcados

A arquitetura de um sistema se refere à estrutura do mesmo, ou seja, a descrição das partes que compõem este sistema e dos relacionamentos entre estas partes. No que tange aos sistemas embarcados, diversos aspectos arquiteturais devem ser considerados: a arquitetura do sistema embarcado como um todo, a arquitetura do software embarcado, a arquitetura do hardware e a arquitetura do processador. Freqüentemente, engloba-se na arquitetura da plataforma computacional os aspectos referentes ao hardware, ao processador e ao sistema operacional. Exemplos de plataformas computacionais incluem o PC com Windows, plataformas Java e plataformas .NET.

A arquitetura dos sistemas computacionais embarcados evolui à medida que estes sistemas evoluíram. Os primeiros sistemas embarcados datam da década de 1960 e incluem o AGC (*Apollo Guidance System*) utilizado no programa Apollo que levou o primeiro homem a Lua. Estes sistemas embarcados eram extremamente simples para os padrões de hoje. Utilizavam lógica RTL (*Resistor-Transistor Logic*) e tinham cerca de 30KBytes de memória. A arquitetura destes sistemas continha apenas dois níveis: *hardware* e *software* de aplicação. Ainda hoje, muitos sistemas embarcados utilizam a arquitetura de dois níveis como, por exemplo, os projetos que utilizam um microcontrolador 8051 programado em linguagem *assembly*, ou, eventualmente, em C.

Esta arquitetura de duas camadas não é escalável para o grau de complexidade dos sistemas embarcados atuais. Até em projetos de impressoras laser reporta-se a presença de milhões de linhas de código fonte. De forma geral, os projetos de sistemas embarcados atuais utilizam uma estrutura de três camadas: *hardware*, núcleo operacional (*kernel*) e *software* de aplicação, sendo o *kernel* (núcleo de um sistema operacional) o equivalente ao sistema operacional de um PC. Esta arquitetura de 3 camadas foi refinada em 7 camadas e será detalhada na Seção 1.2.2.2, a seguir.

Ao contrário da arquitetura de *notebooks, desktops* e *workstations,* o *hardware* dos sistemas embarcados difere de forma significativa e é extremamente dependente da aplicação dada ao sistema, bem como as demais exigências: desempenho, robustez, confiabilidade, características ambientais, etc. De forma geral, a arquitetura de *hardware* de um sistema embarcado engloba o processador, memória de escrita e leitura (p. ex. RAM), memória não-volátil (p. ex. Flash), e inúmeros dispositivos periféricos que permitem o interfaceamento deste sistema com seus usuários e com demais sistemas. Há uma forte tendência de se integrar todo o *hardware* de um sistema embarcado em um único circuito integrado, os chamados SOC (*System-on-Chip*). Outra solução comum, em particular quando o produto não tem volume suficiente para o desenvolvimento de um circuito integrado dedicado, é a utilização de SOM (*System-on-Module*) que consiste em um módulo de hardware (placa de circuito impresso com componentes eletrônicos) que implementa a funcionalidade comum a muitos sistemas embarcados (processador, memória e periféricos de uso mais freqüente). Um SOM, combinado a um circuito que implementa interfaces específicas forma então o *hardware* do sistema embarcado.

A seleção da arquitetura dos processadores utilizados em sistemas embarcados é uma decisão de grande importância, face aos reflexos que tem sobre o custo do produto, desempenho, evolução do mesmo. A decisão pela arquitetura do processador também afeta significativamente o desenvolvimento do produto, tanto nos aspectos de desenvolvimento de hardware como no de software. Percebe-se claramente os efeitos quando um novo projeto impõem uma mudança de arquitetura, pois esta mudança afeta o processo de desenvolvimento, a infra-estrutura de desenvolvimento e principalmente a cultura técnica da empresa.

As arquiteturas de processadores são classificadas em CISC, RISC e VLIW. Os processadores VLIW (*Very Long Instruction Word*) de uso geral são usados principalmente em computadores de grande porte, e ainda não muito pouco usados em sistemas embarcados. Os processadores CISC (*Complex Instruction Set Computer*) representam a arquitetura clássica de processadores, que desde seu surgimento na década de 1940 vem evoluindo no sentido de apresentar conjuntos de instruções cada vez mais complexos. A arquitetura RISC (*Reduced Instruction Set Computer*) surgiu em função do estudo quantitativo de programas e conjuntos de instruções, realizado pelos pesquisadores Hennessy e Patterson [Hennessy, 1992] na década de 1980. O resultado foi a proposta de uma mudança de paradigma na área de arquitetura de computadores: a utilização de computadores com conjunto de instruções reduzido. O baixo número de instruções disponíveis, aliado à simplicidade destas instruções, e a arquitetura *Load-Store* (apenas instruções de *Load* e de *Store* têm acesso à memória), permite a implementação dos processadores RISC utilizado um *pipeline*. Este, por sua vez permite a execução de instruções em taxas médias de uma instrução por *clock*, aumentando de forma muito significativa o desempenho dos processadores RISC quando comparados aos CISC. Muitos processadores RISC atuais, utilizam arquiteturas superescalares ou arquiteturas *superpipeline*, aumentando ainda mais seu desempenho.

Outra forma de classificação de processadores é com relação ao tamanho da palavra de dados, equivalente ao tamanho dos seus registradores. Os sistemas embarcados atuais utilizam processadores de 8, 16 ou 32 bits. Os processadores de 4 bits estão em desuso e os de 64 bits são utilizados principalmente em estações de trabalho, servidores e computadores de grande porte.

1.2.2.1 Arquiteturas de Processadores para uso em Sistemas Embarcados

Algumas arquiteturas têm sido utilizadas com maior freqüência no projeto de sistemas computacionais embarcados. Os fatores que levam a esta preferência são diversos, e incluem aspectos técnicos, econômicos e de mercado.

Dentre os processadores CISC de 8-bits, merece destaque o microcontrolador 8051. Quando lançado pela Intel, no início da década de 1980, este microcontrolador integrava uma memória ROM de 4 KBytes, RAM de 256 bytes e diversos periféricos (temporizadores, porta serial, controlador de interrupções, ...). Na época, estes dispositivos eram programados em *assembly*. Ferramentas de depuração eram extremamente custosas, um ICE (*In-Circuit Emulator*) custava dezenas de milhares de dólares, e proibitivos para empresas de menor porte. Atualmente algumas dezenas de empresas fornecem variantes do 8051. O desempenho aumentou muito ao longo destes

anos, bem como a disponibilidade de ferramentas de desenvolvimento (principalmente compiladores para C e C++) e de depuração, que atualmente estão em custos acessíveis. A arquitetura do processador, caracterizada pelo conjunto de instruções, registradores e mapeamento da memória é praticamente a mesma desde o seu lançamento.

Já dentre os processadores RISC de 8-bits um membro de destaque é o microcontrolador PIC. Desenvolvido no final da década de 1970 pela General Instruments, o PIC foi posteriormente adquirido pela Microchip que hoje o produz e comercializa. A arquitetura do PIC é extremamente simples e, embora classificada como RISC, tem algumas características que não são comumente encontradas em processadores RISC, em particular a presença de um único registrador, enquanto os processadores RISC possuem um número elevado de registradores, e o acesso a memória pela maioria das instruções, enquanto os demais RISC seguem a arquitetura *Load-Store*.

Na categoria de 16-bits a família representativa é a 80186. O microcontrolador 80186 foi desenvolvido pela Intel utilizando a arquitetura x86. Durante as décadas de 1980 e 1990 a Intel e a AMD ofereceram algumas dezenas de variantes do 80186 com diversas combinações de periféricos integrados. Estes microcontroladores fizeram muito sucesso por utilizarem a mesma arquitetura dos processadores dos PCs. Desta forma, os ambientes de desenvolvimento disponíveis para programação em DOS (sistema operacional dos PCs precursor do Windows) podiam ser utilizados para desenvolvimento de sistemas embarcados. Ferramentas como o Turbo C da Borland, que disponibilizava um depurador muito acima dos padrões da época para sistemas embarcados, facilitaram em muito o processo de desenvolvimento e particularmente a depuração destes sistemas.

Atualmente, o foco está nas arquiteturas RISC de 32-bits que são utilizados na maioria dos novos projetos de sistemas embarcados. Algumas arquiteturas que merecem destaque são ARM, Power e MIPS.

A arquitetura ARM foi desenvolvida na década de 1980 pela empresa ARM, na época uma subsidiária da fabricante de computadores pessoais Accorn. A ARM nunca produziu circuitos integrados, seu modelo de negócios consiste na venda de licenças do projeto do seu processador para empresas que produzam microcontroladores. Algumas dezenas de fabricantes de microcontroladores hoje fabricam processadores com processadores ARM, incluindo empresas como Intel e Freescale (anteriormente divisão de semicondutores da Motorola) que são tradicionais desenvolvedores de arquiteturas de processadores. As arquiteturas ARM atualmente em uso são: ARM7TDMI, ARM9, ARM11, XSCALE, Cortex-M3, Cortex-R4 e Cortex-A8. Os ARM7TDMI e Cortex-M3 são utilizados em processadores com *clock* até 100 MHz, desempenho de até 100 MIPS e custo inferior a U$10.00, podendo chegar a custos inferiores a U$ 1.00. Já os processadores Cortex-A8 chegam ao desempenho de 2000 MIPS. O modelo de negócios adotado fez com que dezenas de fabricantes adotassem a arquitetura ARM como o processador (ou *core*) dos seus microcontroladores, disponibilizando ao mercado uma grande variedade de opções e independência de um único fornecedor.

A arquitetura Power, originalmente denominada PowerPC, foi desenvolvida pelo consórcio formado pela IBM, Apple e Freescale. Por muitos anos foi a arquitetura utilizada nos computadores fabricados pela Apple. Atualmente é muito utilizada em

sistemas embarcados na área de Telecomunicações (equipamentos de rede como roteadores e *switches*, além das centrais telefônicas) e em alguns consoles de jogos de vídeo.

A arquitetura MIPS, desenvolvida pela empresa de mesmo nome, tem como um de seus desenvolvedores o Prof. John Hennessy, um dos criadores do conceito de arquiteturas RISC. A arquitetura MIPS tem tido muito sucesso em equipamentos para o mercado de consumo, como, por exemplo, *set-top-boxes*, *modems* para TV a cabo e equipamentos de DVD.

A Figura 1, abaixo, apresenta os volumes nos mercados de processadores de 4 a 32 bits nos últimos anos, com uma previsão até 2010. O levantamento foi realizado pelo Gartner Group. É importante notar que embora o volume dos processadores de 8 e 16 bits ainda seja maior do que o volume dos processadores de 32 bits, estes últimos tem tido uma taxa de crescimento muito mais acentuada.

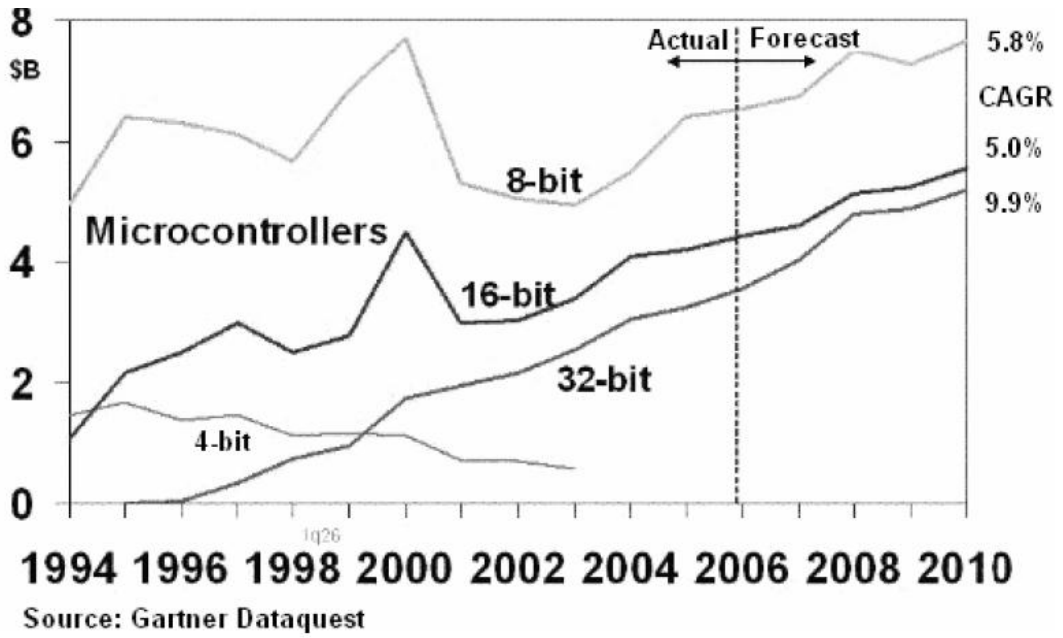
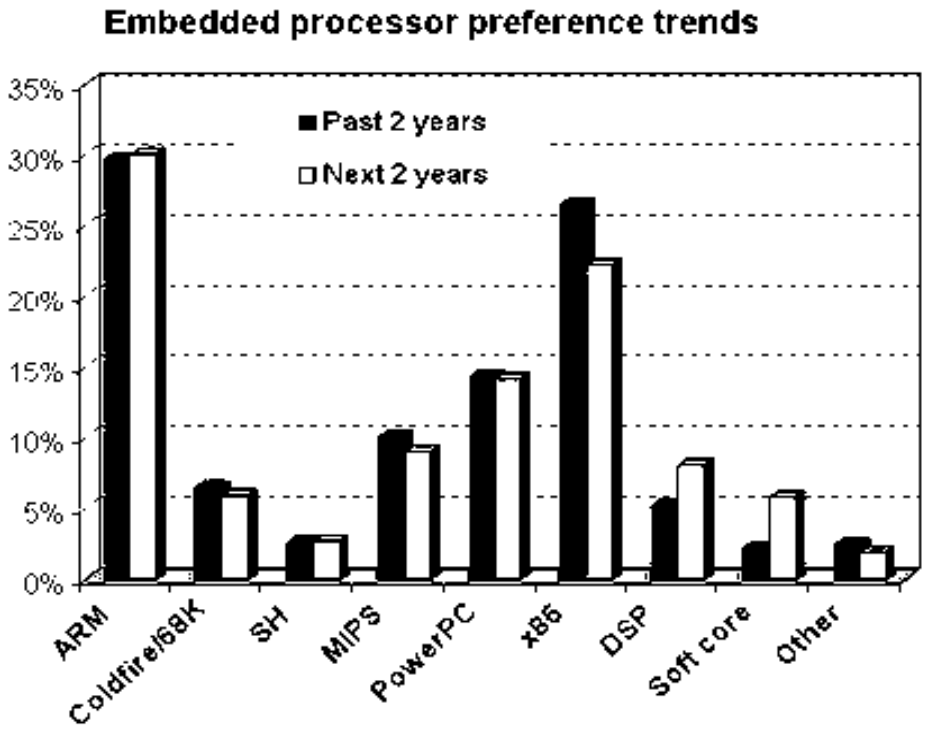


Figura 1 - Mercado mundial de processadores embarcados

Por outro lado, a Figura 2 apresenta a divisão do mercado dos processadores de 32-bits. Trata-se de uma pesquisa de mercado realizada em 2006 visando determinar o volume de projetos com determinada arquitetura. A primeira coluna é referente aos projetos executados nos dois anos anteriores à pesquisa e a segunda coluna é referente aos projetos previstos para os dois anos seguintes. Desta forma, pode-se identificar tendências de mercado com relação ao uso destas arquiteturas. É importante notar como a arquitetura ARM tem tendência de aumento de uso enquanto a maioria das demais arquiteturas tem tendência de declínio.



FONTE: Survey May, 2006 - http://www.linuxdevices.com/articles/AT7070519787.html

Figura 2 - Mercado de processadores de 32-bits

1.2.2.2 Arquitetura de Sistemas Embarcados

No contexto dos sistemas embarcados atualmente em desenvolvimento, o modelo de 2 camadas (Seção 1.2.2) não atende o grau de complexidade necessário. O modelo de 3 camadas (*hardware*, núcleo operacional e *software* de aplicação) é o modelo mais utilizado. Contudo, precisa ser refinado considerando o aumento de complexidade dos sistemas.

O modelo apresentado a seguir é o modelo de 7 camadas [Renaux, 2005]. Ainda é possível identificar as camadas que compõem o *hardware*, o *software* básico (incluindo o núcleo operacional) e o *software* de aplicação.

Cada camada na Figura 3 representa um nível de abstração do *hardware*. É na camada de hardware que a execução efetivamente ocorre, contudo, é inviável, no sentido de ser extremamente moroso e ineficiente, desenvolver sistemas tendo como base as portas lógicas disponíveis no *hardware*. À medida que níveis de abstração vão sendo construídos sobre o *hardware*, o desenvolvimento do sistema torna-se mais ágil e confiável. Embora cada camada apresente para as camadas superiores um nível de abstração maior, é possível, e até freqüente, que a implementação de uma camada acesse serviços de várias camadas inferiores, e não apenas da camada vizinha.

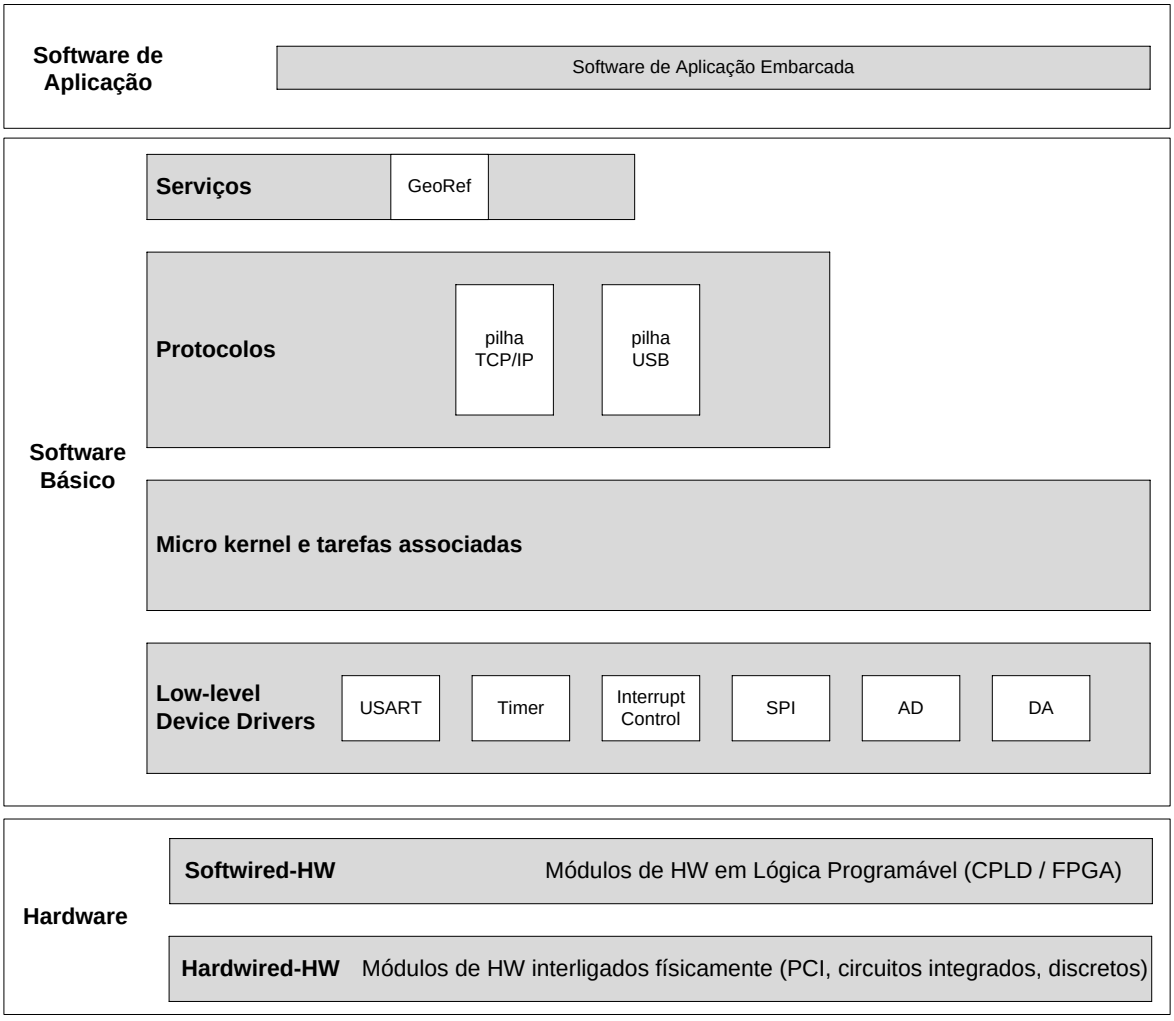


Figura 3 – Modelo de Sistemas Embarcados com 7 camadas.

As camadas que compõem o modelo são:

1) **Hardwired-HW** – é a camada dos dispositivos físicos. É nesta camada que sinais elétricos são comandados por dispositivos semicondutores (transistores) visando a realização de alguma atividade. Os transistores são interligados formando portas lógicas, blocos funcionais, processadores, memórias, e dispositivos de entrada e saída. As interligações são fixas, em parte realizadas internamente aos circuitos integrados e em parte realizadas através de trilhas em placas de circuito impresso.

2) **Softwired-HW** – é uma camada de software armazenada em dispositivos lógicos programáveis. Embora as portas lógicas que compõem estes dispositivos tenham uma matriz de interconexão (camada 1), os pontos de interligação desta matriz podem ser ativados ou não, permitindo desta forma que conexão entre as portas seja efetivamente definida por software. A programação destes dispositivos é tipicamente realizada em VHDL.

3) **Low-Level Device Drivers** – é a camada do *software* básico que interage diretamente com os dispositivos de *hardware* das duas camadas inferiores. Tipicamente esta camada é estruturada na forma de módulos de software que encapsulam cada um dos dispositivos de *hardware* (temporizadores, portas seriais, controladores de interrupção, cartões de memória, etc.). As camadas superiores seguem a regra de não acessar os dispositivos de *hardware* diretamente, mas sempre através dos serviços oferecidos pelos *device drivers*.

4) **Núcleo Operacional** – o núcleo operacional, freqüentemente denominado *kernel* ou RTOS (*Real-Time Operating System*), tem como papel principal a implementação da abstração de concorrência. Embora a maioria dos sistemas embarcados ainda seja baseada em sistemas com um único processador, o núcleo operacional permite a execução concorrente de diversas tarefas. Uma estrutura comum de núcleo operacional é a denominada *microkernel*, que consiste de um módulo central (o *microkernel*) responsável pelo gerenciamento e escalonamento de tarefas, de memória, de comunicação entre tarefas e de serviços de temporização, enquanto as demais funcionalidades do núcleo operacional são implementadas em módulos de software externos ao *microkernel*.

5) **Protocolos** – sobre os canais de comunicação disponíveis (Ethernet, USB, RS-232, RS-485, GPRS, ...) trafegam informações de forma estruturada. Esta estrutura define um padrão de comunicação que permite a interoperabilidade dos sistemas. Esta estrutura está definida na forma de protocolos padronizados que definem tanto o formato dos pacotes de dados a serem transmitidos como a forma de interação entre os equipamentos em comunicação. A padronização destes protocolos justifica a implementação de módulos de software que oferecem serviços de comunicação às camadas superiores.

6) **Serviços** – a camada de serviços implementa módulos de *software* que freqüentemente são reutilizados em diversos sistemas embarcados. Módulos de georeferenciamento e de criptografia são exemplos de serviços nesta camada.

7) ***Software* de Aplicação** – é a camada de software que implementa a funcionalidade específica de um determinado sistema embarcado.

É importante salientar que os sistemas embarcados construídos seguindo o modelo de 7 camadas têm por foco a utilização de módulos de *hardware* e de *software* previamente desenvolvidos e testados, restringindo o esforço de desenvolvimento às camadas 2 (*softwired-HW*) e 7 (*software* de aplicação).

1.2.2. Exemplos de Sistemas Embarcados

Relaciona-se a seguir alguns exemplos de sistemas embarcados com a finalidade de ilustrar a estrutura e aspectos relevantes deste tipo de sistema.

- **Thin Client**

  Um *Thin Client* (“cliente magro” em uma tradução literal) é um equipamento eletrônico com papéis de um microcomputador enxuto para uso dedicado na forma

de um terminal. Freqüentemente, os *Thin Clients* não dispõem de discos rígidos nem de periféricos como unidades de CD/DVD e de disquete. Mesmo interfaces como USB, paralela e serial estão usualmente ausentes. Estes equipamentos se assemelham a um microcomputador PC, porém possuem uma arquitetura muito mais dedicada, ao ponto de serem considerados sistemas embarcados. A figura a seguir ilustra uma estrutura típica de um *Thin Client*.

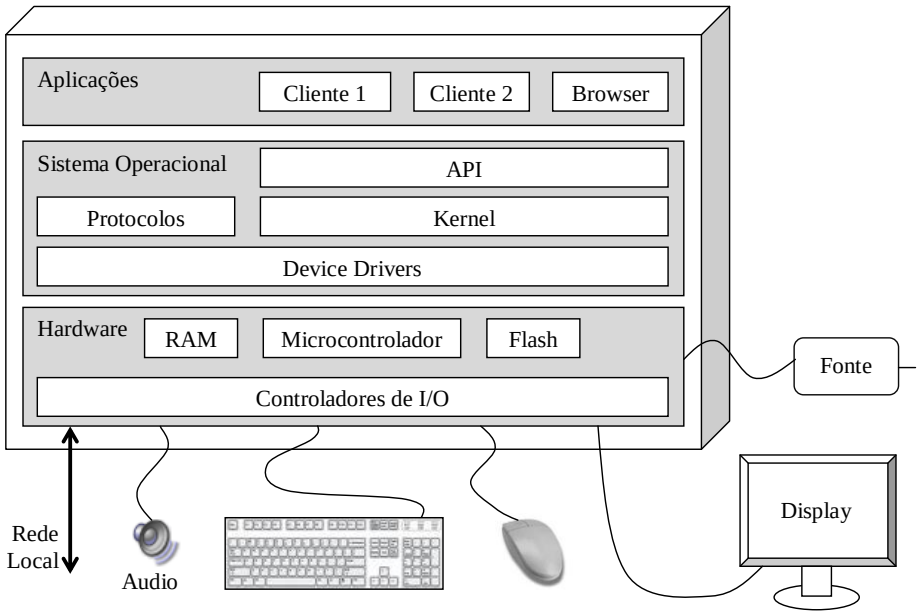


Figura 4 – Estrutura típica de um *Thin Client*.

A maioria destes equipamentos possui uma infra-estrutura mínima necessária para executar programas remotamente de um servidor de aplicações. Assim, em termos de aplicação os *Thin Clients* costumam incluir um ou mais software clientes usados para estabelecer conexões com servidores de aplicação como RDP (*Remote Desktop Protocol* da Microsoft) e ICA (da Citrix). Além disso, a utilização de um *browser* permite, também, acesso a uma aplicação *web* a partir de um servidor remoto (*web server*). As eventuais aplicações locais são freqüentemente relacionadas com operações de configuração do próprio *Thin Client*.

Em termos de sistema operacional, os *Thin Clients* costumam ter versões reduzidas contendo apenas os suportes efetivamente necessários para execução de suas funções. O objetivo é reduzir os requisitos de *hardware* (memória, processador, armazenamento) necessários para executar o *software* do *Thin Client*. Exemplos mais comuns de sistemas operacionais usados em *Thin Clients* são o Windows CE e Windows XPe da Microsoft e versões enxutas do Linux.

No que diz respeito ao *hardware*, a eletrônica dos *Thin Client* possui semelhanças com a arquitetura PC já que eles são inspirados em microcomputadores de mesa. Porém, para se alcançar custos competitivos é comum o desenvolvimento de uma placa mãe própria para *Thin Clients* reduzindo-se os recursos ao mínimo necessário para as funções de terminal. Assim, como ilustra a Figura 4 um *Thin Client* costuma ter apenas interfaces para teclado, mouse, *display* e áudio, e uma interface de rede.

- **PDA – *Personal Digital Assistant***

Um PDA (Assistente Pessoal Digital), também chamado *Handheld*, é um dispositivo eletrônico de uso pessoal que cumpre o papel de um pequeno computador de mão. Trata-se, portanto, de um equipamento de dimensões e peso reduzidos e de uso portátil. Apesar dar dimensões reduzidas, os PDAs atuais dispõem de capacidade de processamento e de armazenamento considerável permitindo a execução de software sofisticados, muitos deles semelhantes ao que dispõe em microcomputadores PCs.

Atualmente, existem três famílias principais de PDAs, baseados no Palm OS (da PalmSource), no Windows Mobile (da Microsoft) e no Linux. Diversos fabricantes desenvolvem e comercializam PDAs baseados nestas famílias. O mercado de PDAs vem crescendo ano a ano, principalmente pela contínua integração de novas funcionalidades como comunicação celular, comunicação por redes sem fio (WiFi), câmera fotográfica e tocadores de música e vídeo.

Os PDAs são exemplos mais claros de sistemas embarcados pois tem limitações mais evidentes de dimensão, de peso, de consumo de energia e de custo. A figura a seguir ilustra a estrutura típica de um PDA.

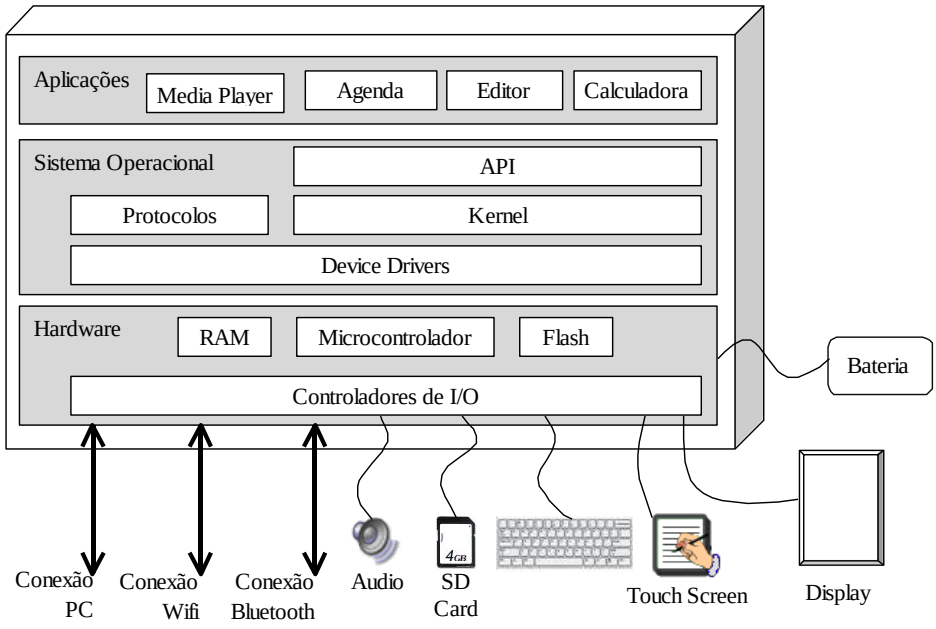


Figura 5 – Estrutura típica de um PDA.

PDAs podem ser muito ricos em termos de aplicações. Sua capacidade de processamento e memória permitem o desenvolvimento de programas elaborados que em muito se assemelham às aplicações encontradas em microcomputadores PCs. A infra-estrutura de software disponível, como .NET e máquinas virtuais Java, facilitam o desenvolvimento de aplicações por equipes que não precisam ser especializadas em sistemas embarcados (embora haja especificidades e restrições a serem respeitadas).

No que diz respeito ao hardware, os projetos são desenvolvidos especificamente para este tipo de equipamento. Os microcontroladores empregados são de 32 bits e a

arquitetura mais utilizada é ARM operando em freqüências entre 200 e 400 MHz. A capacidade de memória pode variar mais é comum encontrar PDAs com 128 MB de RAM para execução de programas e 128 MB de Flash para armazenar o sistema operacional e programas instalados.

- **Telemetria**

Um outro exemplo de emprego de sistemas embarcados é na área de telemetria, ou seja, sistemas de medições à distância. Este tipo de sistema pode ser utilizado em vários campos como: rastreamento de veículos, rastreamento de transporte de cargas e monitoração de estações autônomas diversas. Um sistema de telemetria ilustra ainda melhor uma aplicação de sistemas embarcados, pois destina-se a um propósito ainda mais específico que os *Thin Clients* e PDAs.

A figura a seguir ilustra uma estrutura possível para um sistema de telemetria para acompanhamento do transporte de cargas.

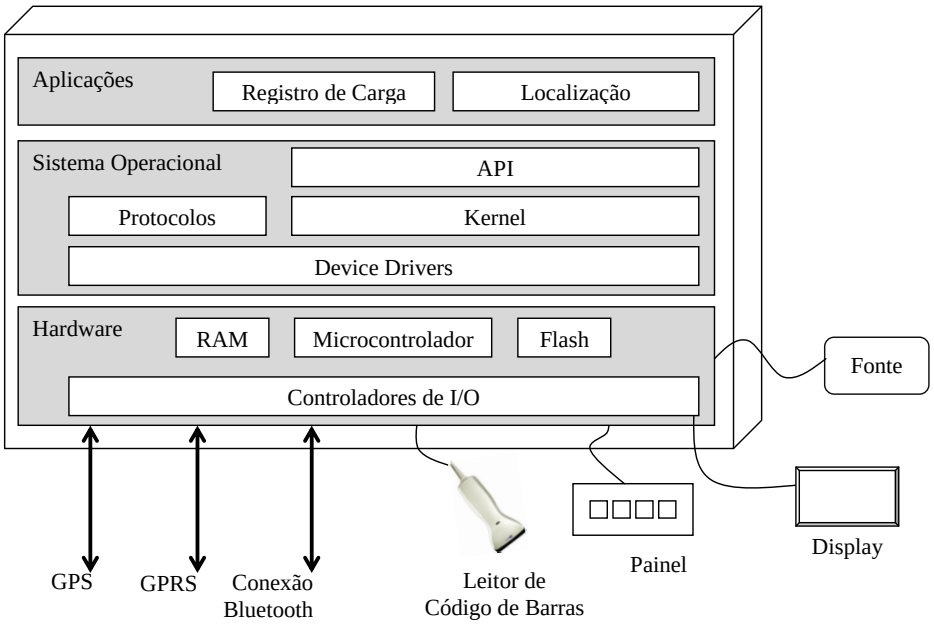


Figura 6 – Estrutura hipotética de um sistema de telemetria.

Neste exemplo, a camada de aplicação é composta por um módulo responsável pelo registro (carregamento e descarregamento) das cargas transportadas e por um módulo de localização da carga. O módulo de registro permite identificar as cargas transportadas através da leitura do código de barras dos produtos carregados. Durante o descarregamento em cada localidade é feito o registro de saída dos produtos. Os registros de carga são armazenados na memória *flash* do sistema embarcado e utilizam um leitor de código de barras para identificação das cargas..

O módulo de localização é responsável pelo georeferenciamento do sistema embarcado, ou seja, identificação de sua localização através das informações do receptor GPS (*Global Positionning System*). Além disso, este módulo é responsável, também, pelo envio periódico da localização e da identificação da carga sendo transportada a um centro de operações. Neste exemplo, adotou-se um *transceiver* GPRS (*General Packet Radio Service*) para comunicação de dados com o centro de

operações. Desta forma, o centro de operações pode monitorar os horários de carga e descarga das mercadorias transportadas e rastrear sua localização continuamente.

1.2.3. Visão de Mercado de Sistemas Embarcados

Esta seção apresenta alguns dados de mercado que fornece uma visão das preferências e tendências atuais em sistemas embarcados.

Arquiteturas de Microcontroladores

Microcontroladores com diferentes comprimentos de palavras são empregados na construção de sistemas embarcados, variando de 4 a 64 bits. Uma análise de mercado [Ganssle, 2006] realizada em 2006, durante a ESC – Embedded System Conference - nos Estados Unidos, mostra que os microcontroladores de 32 bits são os mais citados entre os projetos de sistemas embarcados atualmente em desenvolvimento, conforme o percentual de projetos ilustrado na figura 7.

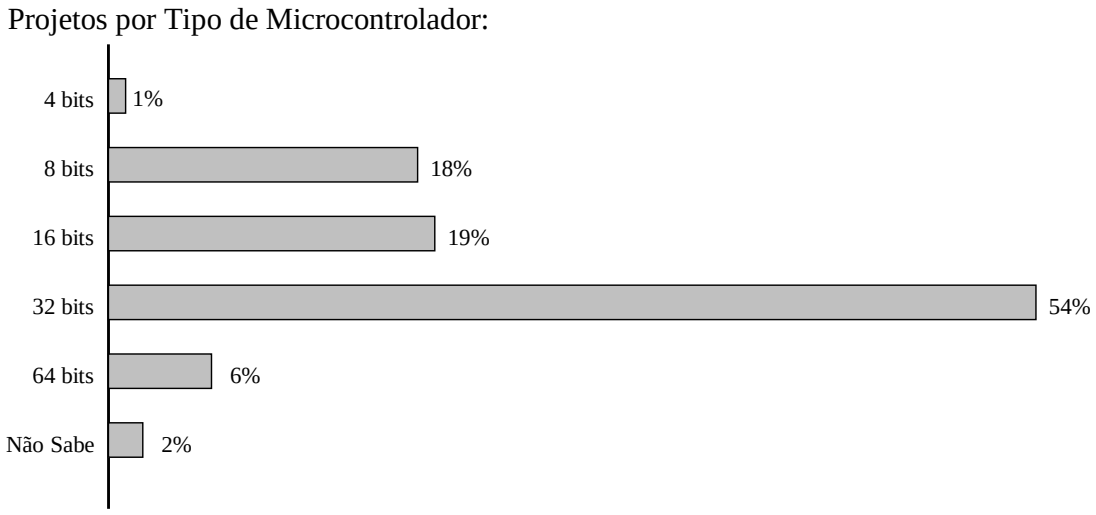


Figura 7 – Percentagem de uso dos diferentes tipos de microcontroladores.

Apesar das previsões de redução de uso dos microcontroladores de 8 e 16 bits, nota-se que estas arquiteturas ainda mantém um espaço considerável nos projetos. A razão principal, segundo Ganssle, é que determinadas aplicações são tão sensíveis a custos que os microcontroladores de 32 bits conduziriam a produtos não competitivos. Assim, pode-se considerar que estas arquiteturas ainda serão bastante usadas nos próximos anos.

Com relação à freqüência de operação dos microcontroladores (*clock rate*), a mesma pesquisa [Ganssle, 2006] mostra uma predominância de sistemas embarcados utilizando microcontroladores operando na faixa de 10 à 99 MHz, conforme ilustrado na Figura 8. As principais razões para este predomínio são o custo dos processadores de maior freqüência e o aumento de consumo. Deve-se notar que um número expressivo de sistemas embarcados são alimentados por baterias fazendo com que o consumo seja um critério fundamental na escolha do microcontrolador.

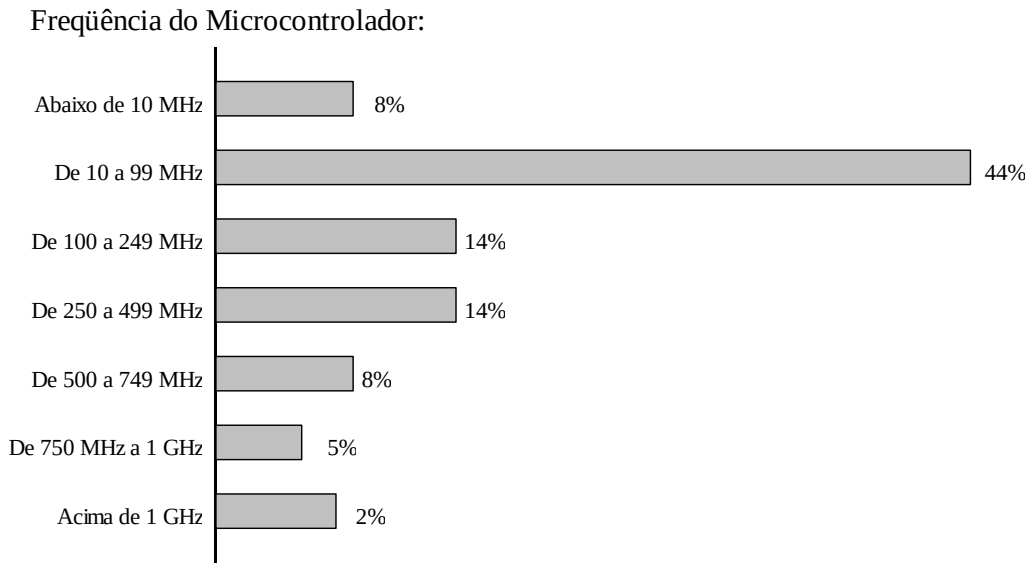


Figura 8 – Percentagem de emprego das diferentes freqüências dos microcontroladores.

Em termos de distribuição de mercado (*market share*) global os principais fabricantes de semicondutores são a Freescale, Intel, Microchip, TI e Atmel [Ganssle, 2006]. No universo de microcontroladores de 8 bits os principais fabricantes são a Microchip, Atmel, Freescale, Intel e Zilog, nesta ordem. No segmento de 16 bits os principais fabricantes são a Microchip, TI, Freescale, Intel, AMD, Philips, Zilog e STMicro. Finalmente, no segmento de 32 bits a Intel, AMD, Atmel, Freescale, IBM e Xilinx são os principais fabricantes.

Sistemas Operacionais

Muitos projetos de sistemas embarcados fazem uso de um sistema operacional (SO) como suporte para o software de aplicação. Segundo levantamento realizado por Jim Turley [Turley, 2006], 71% dos desenvolvedores de sistemas embarcados empregam algum sistema operacional em seus projetos correntes. Estes sistemas podem ser categorizados em SO comercial, SO desenvolvido internamente, SO de código aberto (*open-source*) e SO de código aberto distribuído comercialmente, conforme ilustrado na Figura 9.

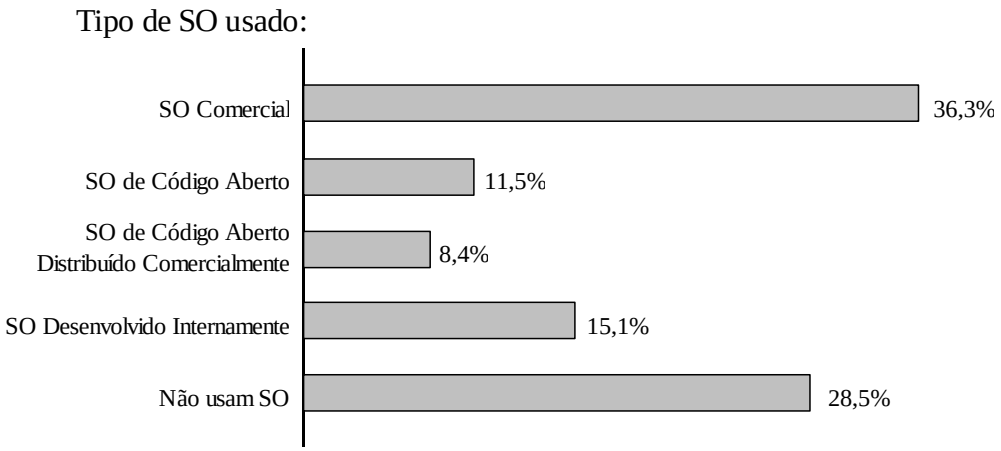


Figura 9 – Percentagem de uso dos diferentes tipos de SO.

Há no mercado um número expressivo de sistemas operacionais próprios para emprego em sistemas embarcados. A Figura 10 apresenta uma lista, extraída de [Turley, 2006], dos principais sistemas operacionais empregados.

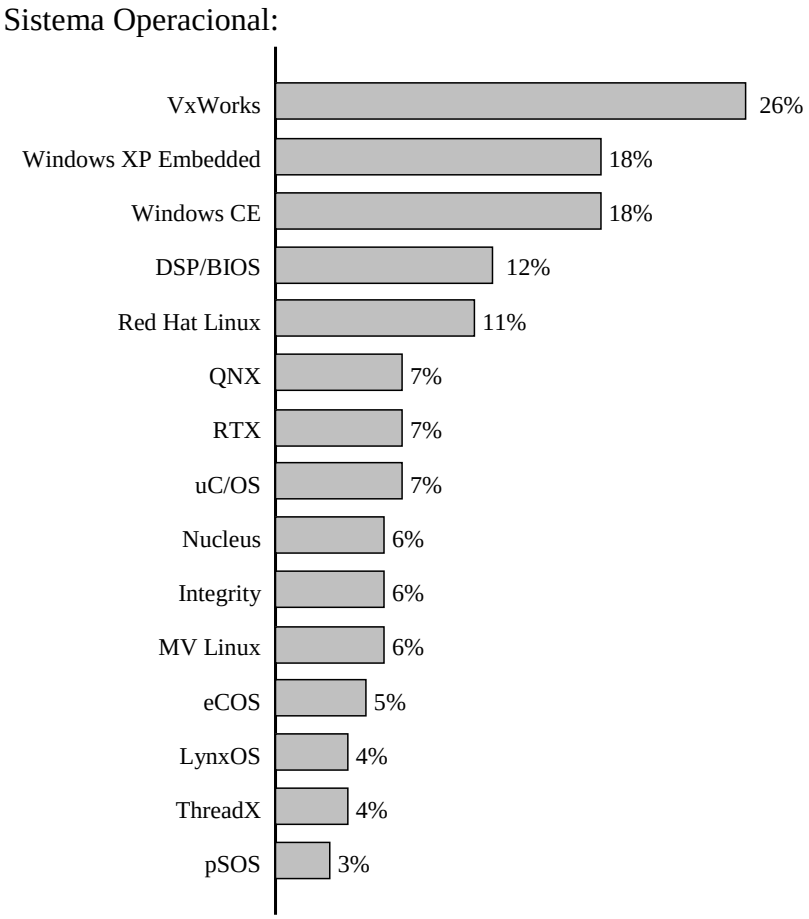


Figura 10 – SOs mais empregados em sistemas embarcados.

Linguagens de Programação

Diversas linguagens de programação são empregadas no desenvolvimento de sistemas embarcados. Na origem dos sistemas embarcados, a linguagem *assembly* foi largamente empregada em razão da especificidade e necessidade de otimização do código. Atualmente, com a disponibilidade de compiladores mais modernos, as linguagens C e C++ são as predominantes nesta área. A Figura 11 apresenta uma estatística de uso das linguagens de programação em projeto de sistemas embarcados [Ganssle, 2006] (mais de uma resposta permitida).

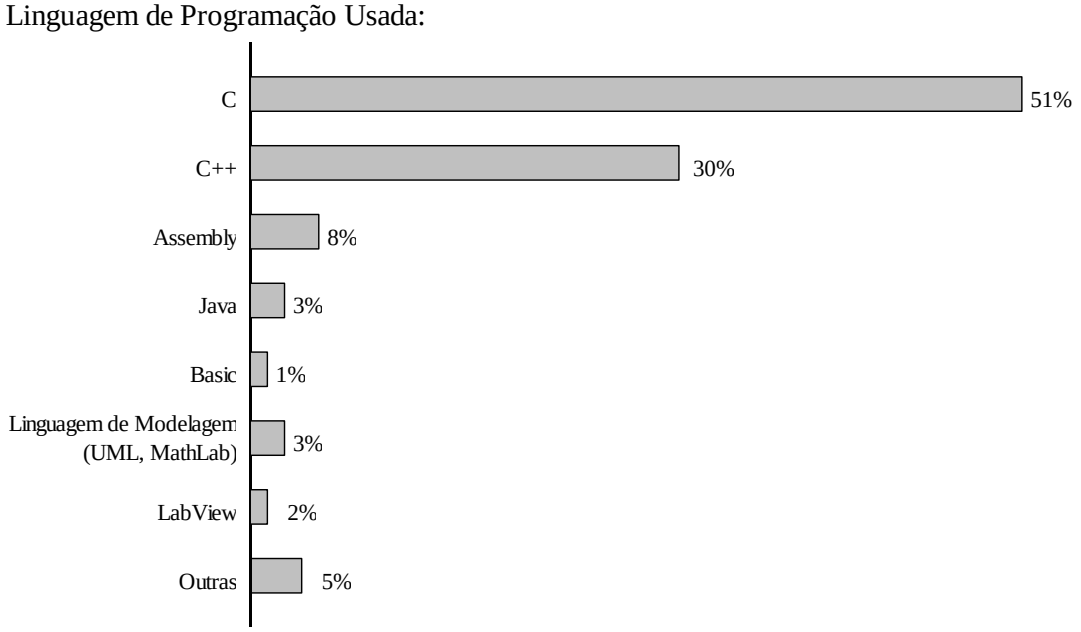


Figura 11 – Estatística de emprego das diferentes linguagens de programação.

1.2.5. Desenvolvimento de Sistemas Embarcados

O processo de desenvolvimento de um sistema embarcado não é único, pelo contrário, precisa ser adaptado às características do Domínio da Aplicação, à cultura da empresa desenvolvedora e ao contexto de cada desenvolvimento específico. O processo apresentado abaixo ilustra um possível processo de desenvolvimento de um produto baseado em um sistema embarcado.

1) Concepção do Produto

Esta fase consiste na identificação de oportunidades de mercado, em um estudo de mercado visando identificar o potencial de mercado deste produto face à concorrência e ao interesse do mercado, na elaboração de um EVTEC (Estudo de Viabilidade Técnica Econômica e Comercial), em um plano de negócios e em um plano de marketing. Confirmada a viabilidade deste produto, elabora-se uma relação preliminar de requisitos.

2) Engenharia de Requisitos

A fase de engenharia de requisitos visa o entendimento dos requisitos do produto e o seu refinamento até a elaboração da especificação do sistema embarcado

(produto). Envolve a Análise de Domínio que determina requisitos comuns a uma família de produtos/aplicações, visando o desenvolvimento de módulos genéricos e a conseqüentemente a reutilização destes módulos. Envolve também a Análise de Requisitos e a Modelagem de Requisitos. Nesta fase é aconselhável a construção de protótipos funcionais, também conhecidos por provas de conceito, que são protótipos elaborados rapidamente, a partir de placas de avaliação, reuso de módulos de software de outros projetos, além de módulos de hardware e de software gerados automaticamente por ferramentas de prototipação rápida (em *hardware* tipicamente se utiliza lógica programável para realizar prototipações). Estes protótipos são utilizados para validar os requisitos em elaboração.

O principal resultado desta fase é a documentação da especificação do produto, que consolida os diversos resultados das atividades realizadas.

3) Engenharia de Sistema

A Engenharia de Sistema trata do produto como um todo, nos seus diversos aspectos: *hardware*, *software*, mecânica (gabinete, componentes mecânicos, ...), ergonomia, robustez (física e lógica), dentre outros. Nesta fase tem início o planejamento da solução, ou seja, quais as partes que compõem o produto e quais as dependências entre estas partes. Um dos resultados da Engenharia de Sistemas são as especificações do *software*, do *hardware* e da mecânica do produto.

4) Processo de Desenvolvimento de Hardware

Esta fase consiste na análise dos requisitos de *hardware*, constantes do documento de especificação do *hardware*, no projeto do *hardware*, na montagem e avaliação. O projeto do *hardware* inicia-se com a definição da arquitetura do *hardware* e dos componentes principais que irão compor o produto (também denominados componentes A). Definida a arquitetura pode-se fazer uso de protótipos funcionais para validar esta arquitetura. A diferença em relação aos protótipos construídos na fase de Engenharia de Requisitos é que os protótipos construídos neste momento utilizam os componentes A selecionados nesta fase. O projeto do *hardware* é então detalhado produzindo o diagrama esquemático e o projeto da placa de circuito impresso. Um número reduzido de placas é montado para fins de avaliação do projeto. Esta avaliação pode indicar a necessidade de alterações no projeto.

5) Processo de Desenvolvimento de Software

O processo de desenvolvimento de software será detalhado na seção 1.4.1. Conceitualmente segue a mesma filosofia do processo de desenvolvimento de hardware: análise de requisitos, definição da arquitetura de software, detalhamento do projeto de software, prototipação rápida (quando for o caso), codificação, testes de módulos e da integração de módulos.

6) Processo de Desenvolvimento da Mecânica

O processo de desenvolvimento da mecânica também segue a os de *hardware* e *software*, para os componentes mecânicos que compõem o produto.

7) Integração do Sistema

A integração do sistema consiste em integrar os módulos de *hardware*, de software e da mecânica, já desenvolvidos e testados individualmente. Correções e ajustes durante o processo de integração podem se tornar muito trabalhosos e complexos, por isso a importância de um trabalho meticuloso nas fases anteriores.

8) Teste de Sistema

O sistema (produto), já integrado, é testado como um todo. Este teste também é denominado de caixa-preta, visto que o os componentes e módulos que o compõem já foram testados individualmente bem como a sua integração as módulos relacionados. Nesta fase valida-se o sistema como um todo em relação aos requisitos do produto. A rastreabilidade ao longo de todo o processo de desenvolvimento mostra-se importante neste momento.

9) Teste em Campo

Um pequeno lote é produzido e distribuído aos chamados *beta-testers*, ou seja, usuários dispostos a avaliar este novo produto. O resultado desta avaliação pode causar o retorno a uma das fases anteriores, com implicações por vezes severas em prazo e custo.

10) Documentação de Produto e de Produção

O processo utilizado até então para a manufatura de protótipos é basicamente artesanal. Dependendo do volume de produção se faz necessário o planejamento e construção da infra-estrutura necessária para produção em escala, envolvendo documentação de produção e teste, projeto e construção de jigas de teste (equipamentos de teste do produto ou de partes deste), instruções de compra de insumos, de montagem, e outras atividades afins.

11) Empacotamento de produto

Uma fase importante no ciclo de desenvolvimento de um produto é o seu empacotamento, termo utilizado para se referir às diversas atividades como: criação gráfica e produção de embalagem, manuais e material de divulgação tanto em papel como em meios eletrônicos.

O processo de desenvolvimento acima indica que um grande número de atividades são necessárias para o desenvolvimento de um novo produto e a complexidade associada a gestão destas atividades. O desenvolvimento do software embarcado é um aspecto deste processo e deve ser entendido no contexto todo.

1.3. Software Embarcado

Após uma visão abrangente do sistema embarcado como um todo, esta seção trata da caracterização do software embarcado, da sua estrutura e do suporte a execução, em particular do papel dos núcleos operacionais.

1.3.1. Processos e Tarefas

A maioria dos sistemas embarcados tem como requisito executar mais de uma funcionalidade. Assim, por exemplo, em um telefone celular o usuário poderia executar a funcionalidade de ”registro de um novo nome e endereço na agenda”, a funcionalidade de “atendimento a uma chamada telefônica” e a funcionalidade de “seleção de um tom para a campainha”. Dada a independência entre estas funcionalidades e considerando os princípios de modularidade, cada funcionalidade poderia ser implementada em um módulo de software que integrados formariam um programa do sistema embarcado.

- **Processos**

A distribuição das funcionalidades entre os módulos forma a estrutura interna do programa. O software embarcado pode também ser organizado em unidades independentes de execução como um meio adicional de tratar a complexidade. Neste sentido, um dos conceitos empregados é o de **processo**. Um processo é uma unidade de execução definida por seu código (conjunto de instruções) e seu conjunto de dados (registradores e dados de operação), encapsulados em uma área de memória própria. Cada execução de um determinado programa constitui um processo. Um software embarcado cujas funcionalidades são agrupadas em diferentes processos é dito um sistema multiprocesso.

As funcionalidades de um software embarcado podem ser executadas uma a uma conforme a requisição do usuário ou de outros atores associados ao sistema embarcado. Assim, em um telefone celular, por exemplo, o usuário poderia ora ativar a agenda para incluir um novo registro, ora atender a uma chamada e ora selecionar o tom da campainha, executando alternativamente cada uma das funcionalidades existentes. Caso estas funcionalidades estivessem implementadas em processos distintos, cada processo seria executado de maneira alternada.

Em sistemas embarcados, entretanto, é muito comum a necessidade de execução simultânea de funcionalidades, ou seja, a necessidade de se executar mais de uma atividade ao mesmo tempo. Uma das maneiras de tratar esta necessidade de simultaneidade é distribuir as funcionalidades em múltiplos processos e, então, executar estes processos ao mesmo tempo. Como em muitos sistemas embarcados há apenas um microcontrolador, torna-se necessária a utilização de um mecanismo para compartilhar o uso do microcontrolador entre os processos em execução. Este mecanismo chama-se **tempo compartilhado** (*time sharing*) e é responsável por ativar a execução de cada processo por um determinado período de tempo, de forma que todos os processos possam ser executados. Este mecanismo implementa a simultaneidade na forma de uma concorrência entre os processos pelo tempo do microcontrolador. A atividade de alocação do processador entre os processos é chamada **escalonamento**.

A concorrência de processos é um mecanismo eficiente e relativamente simples para implementar funcionalidades simultâneas em software embarcado, em especial quando existe pouca interação entre os processos. Quando o número e freqüência das interações entre os processos são maiores, os recursos de comunicação comuns em sistemas multiprocessos, como chamadas de funções externas e compartilhamento de memória, tornam-se insuficientes. Além disso, quando se tem um número maior de funcionalidades

simultâneas, a criação de vários processos fragmenta excessivamente o software gerando uma sobrecarga de interações entre os processos.

- **Tarefas**

No contexto de um processo, implementa-se a concorrência na forma de tarefas (*tasks*). Uma tarefa é uma linha ou fluxo de execução (*thread*) dentro de um processo. Cada processo quando entra em execução tem uma tarefa criada que pode ser chamada de tarefa primária. Através desta tarefa, o fluxo de execução do código é realizado percorrendo-se o conjunto de instruções que forma o código do processo. A tarefa tem acesso direto aos registradores e área de memória do processo. Quando um software necessita executar funcionalidades simultâneas, pode-se criar mais de uma tarefa dentro de um processo para implementar uma concorrência de atividades. Assim, haverá tantos fluxos de execução dentro de um processo quanto for o número de tarefas. Deve-se notar que todas as tarefas são executadas dentro do contexto do mesmo processo e, portanto, compartilham a mesma área de código e de dados. Cada tarefa possui, entretanto, seus próprios valores dos registradores, seu próprio contador de programa (*program counter*) e sua própria pilha (*stack*).

Cada tarefa representa um fluxo de execução dentro de um processo, freqüentemente, percorrendo uma porção específica do código daquele processo. Nada impede, entretanto, que duas tarefas percorram uma mesma porção do código. A estratégia básica no uso de tarefas é associar a execução de cada funcionalidade simultânea a uma tarefa. Isto permitiria, por exemplo, no caso do telefone celular, que uma tarefa executasse a funcionalidade "registro de um novo nome e endereço na agenda" enquanto outra tarefa independente executasse a funcionalidade "atendimento a uma chamada telefônica". Isto permitira ao usuário fazer um registro na agenda enquanto ele, simultaneamente, atende a uma chamada telefônica.

As grandes vantagens da programação multitarefa em relação à programação multiprocesso são a maior facilidade e eficiência nas interações entre as tarefas e o compartilhamento da área de dados do processo entre as tarefas. Com relação ao escalonamento, não há diferença importante uma vez que o compartilhamento do microcontrolador é, tipicamente, baseado em tarefas. Isto pode significar que, se um processo tiver um número maior de tarefas, ele poderá receber uma parcela maior do tempo do microcontrolador (isto acontece no caso do Windows CE, por exemplo).

1.3.3. Concorrência entre Tarefas

A eficiência na implementação de concorrência através de múltiplas tarefas, faz com que este mecanismo seja o mais utilizado em sistemas embarcados para tratar questões de simultaneidade. Mesmo para software embarcado que não dispõe de suporte para multitarefa, é comum a implementação de rotinas que simulam a concorrência entre módulos do software. Entretanto, quando a complexidade das tarefas e de suas interações cresce, o emprego de um sistema operacional torna-se recomendável, senão obrigatório. O uso de um sistema operacional permite também separar a lógica da camada de aplicação da infra-estrutura de implementação da concorrência.

Os sistemas operacionais para aplicações embarcadas freqüentemente disponibilizam recursos mais avançados para controle das concorrências entre tarefas do que o sistemas operacionais para PCs. Isto se deve ao fato de que as necessidades de concorrência são

maiores e mais complexas em sistemas embarcados. Alguns destes recursos são descritos a seguir.

- **Configuração do *Quantum* das Tarefas**

Através do escalonamento por tempo compartilhado, o uso do processador é disponibilizado para execução de cada tarefa durante um determinado período de tempo, chamado ***thread quantum***. Caso a tarefa interrompa suas atividades (seja pelo término de suas ações, pela necessidade de algum recurso indisponível naquele momento ou pela necessidade de aguardar determinada temporização) antes do tempo disponibilizado a ela se esgotar, o escalonador irá efetuar um chaveamento de contexto e disponibilizará o processador para outra tarefa. Se, entretanto, a tarefa continuar executando ações até o esgotamento do tempo disponível, ela será interrompida e a sua execução continuará na próxima vez que ela for escalonada. O processo de interrupção da execução de uma tarefa chama-se **preempção**.

Por padrão, atribui-se um valor de *quantum* igual para todas as tarefas, por exemplo, 100 ms. Entretanto, pode-se alterar o *quantum* de determinadas tarefas, para menor ou para maior, conforme se deseje disponibilizar menor ou maior tempo do processador. Esta configuração do *quantum* afeta, portanto, o desempenho das tarefas na execução de suas ações permitindo priorizar tarefas mais importantes ou que operem a taxas mais elevadas. A Figura 12 ilustra, através de um gráfico de Gantt, um cenário com quatro tarefas de quantos diferentes sendo executadas. Nota-se que, embora todas as tarefas tenham a mesma oportunidade de execução, as tarefas 2 e 4 poderão executar mais ações pois possuem um *quantum* de 100ms, maior que os das tarefas 1 e 3. As linhas verticais tracejadas mostram os momentos de preempção das tarefas.

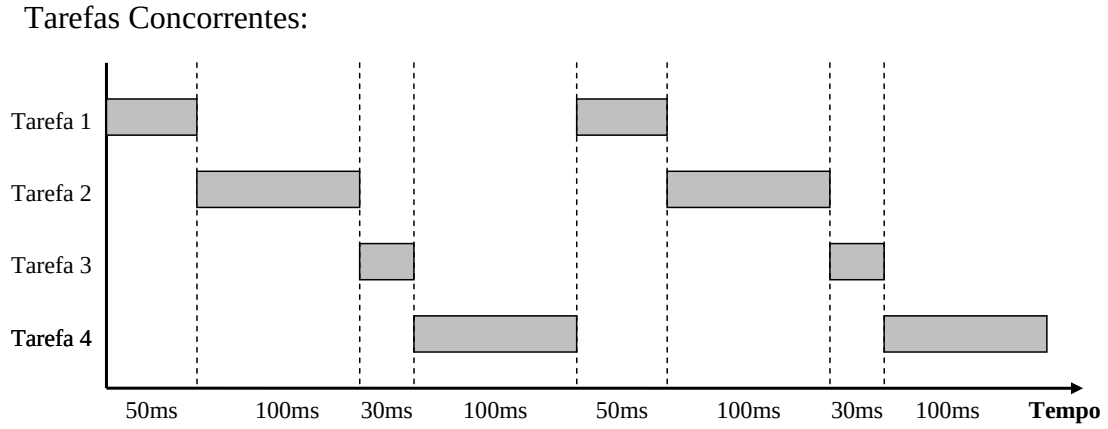


Figura 12 – Exemplo de execução concorrente de quatro tarefas com quantum diferentes.

- **Definição de Níveis de Prioridade**

Outro mecanismo para privilegiar determinadas tarefas é o estabelecimento de prioridades às tarefas. Comumente, os sistemas operacionais permitem utilizar até 256 níveis de prioridade, sendo 0 o nível mais alto e 255 o nível mais baixo. Os níveis de

prioridade são um mecanismo forte de definição da importância relativa das tarefas. Quando se atribui um determinado nível de prioridade a uma tarefa, isto significa que ela sempre terá privilégio de execução sobre todas as demais tarefas de níveis inferiores. Assim, sempre que esta tarefa desejar (ou puder) ser executada, imediatamente qualquer outra de nível inferior será interrompida ("preemptada") para ceder o processador. Nenhuma outra tarefa de nível inferior será executada enquanto esta tarefa ocupar o processador. Tarefas com o mesmo nível de prioridade são, tipicamente, escalonadas seqüencial e ciclicamente usando um algoritmo chamado Round-Robin.

O fato de uma tarefa ter um nível de prioridade menor não significa que ela nunca ou dificilmente será executada. As tarefas em sistemas embarcadas são muito dinâmicas e reativas, de forma que mesmo as tarefas de maior prioridade liberam, freqüentemente, o processador a outras tarefas.

A Figura 13 ilustra uma estrutura de níveis de prioridade de tarefas. Para cada nível podem existir diversas tarefas que são escalonadas seqüencialmente. Determinados níveis podem não ser usados. As tarefas de mais alta prioridade são, tipicamente, relacionadas com ações do próprio sistema operacional e com rotinas de tratamento de interrupções.

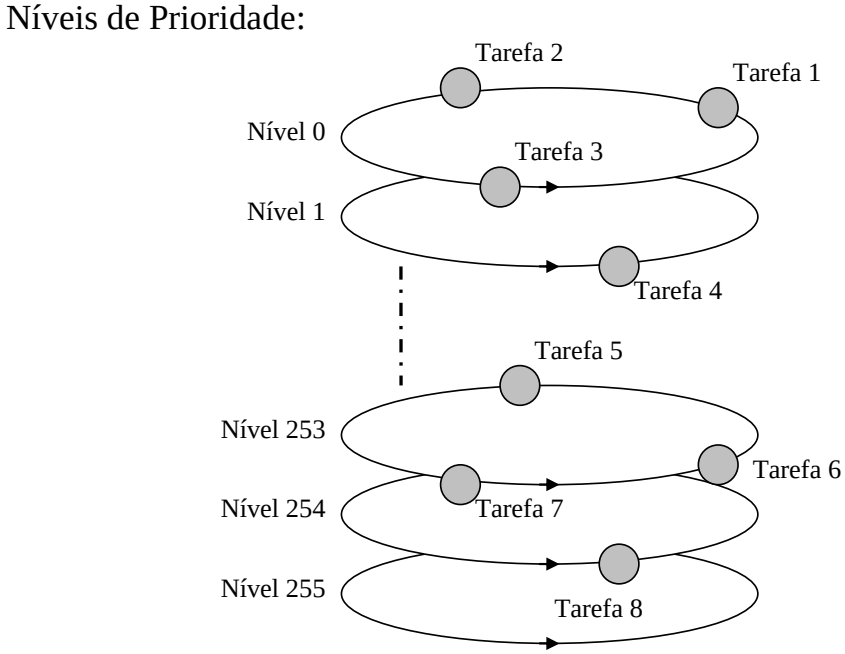


Figura 13 – Organização de tarefas em níveis de prioridade.

- **Comunicação e Sincronização entre Tarefas**

Durante a execução das tarefas em um sistema embarcado, pode ser necessária a troca de mensagens de uma tarefa para outra. Uma troca de mensagem pode significar que uma tarefa está enviando algum dado para outra tarefa. Assim, através das interações usando mensagens, pode-se estabelecer um fluxo de dados entre as tarefas. Uma tarefa poderia, por exemplo, ler dados de uma porta serial, verificar a validade dos dados e

enviá-los a outra tarefa através de uma mensagem. A segunda tarefa poderia receber estes dados, efetuar uma análise e gravá-los em uma memória. Estas ações realizadas pelas duas tarefas podem parecer seqüenciais, porém deve-se notar que enquanto a segunda tarefa analisa e grava determinados dados, poderia haver a recepção de novos dados vindos da porta serial. Daí o interesse de implementação na forma de duas tarefas concorrentes. Deve-se notar também que a mensagem enviada pela primeira tarefa deve ser assíncrona de forma que a execução da primeira tarefa não seja interrompida pelo tratamento da mensagem feito pela segunda tarefa.

Além da comunicação entre tarefas através da troca de mensagens, freqüentemente as tarefas realizam ações complementares e precisam de mecanismos de sincronização. Estes mecanismos definem dependências causais entre as tarefas e são necessários para garantir que uma determinada ordem de execução das ações entre as tarefas seja respeitada. A sincronização entre tarefas pode ser feita também através de mensagens. Neste sentido uma tarefa pode enviar uma mensagem indicando a outra tarefa que já concluiu determinada ação ou que já alcançou determinado estado. Da mesma forma, uma tarefa pode suspender suas ações até que receba uma notificação (mensagem) informando que outra tarefa já alcançou determinado estado. Esta interação com o objetivo de ajustar o andamento das ações entre tarefas é dito sincronismo entre tarefas.

Considerando ainda o exemplo anterior, a segunda tarefa, após terminar a análise e gravação de um grupo de dados recebidos da primeira tarefa, poderia passar à análise de um novo grupo de dados. Porém este novo processamento depende da primeira tarefa ter concluído a recepção e validação de novos dados. Há, portanto, uma dependência da segunda tarefa em relação ao estado da primeira. A segunda tarefa deverá suspender suas ações aguardando a chegada de uma nova mensagem da primeira tarefa. Esta mensagem, portanto, não representa uma comunicação, mas também um sincronismo entre as tarefas. A Figura 14 ilustra este cenário.

Comunicação e Sincronismo:

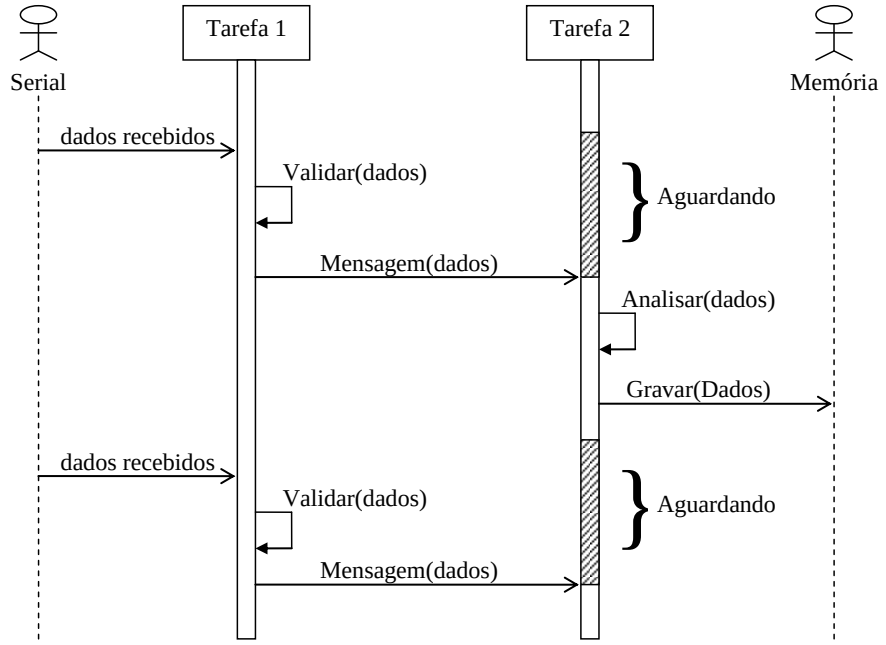


Figura 14 – Comunicação e sincronismo entre duas tarefas.

Comunicação e Sincronismo:

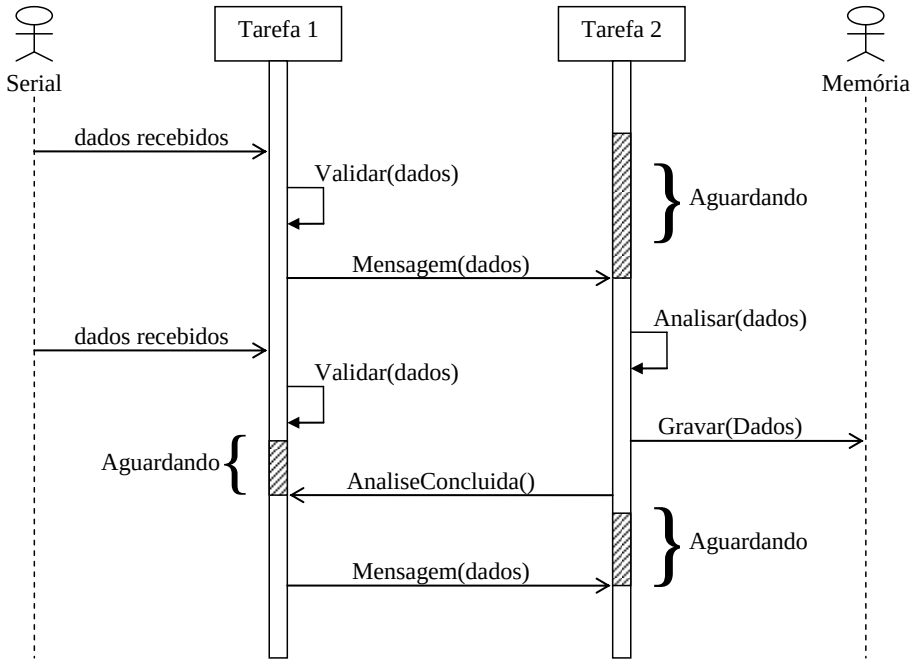


Figura 15 – Segunda versão da comunicação e sincronismo entre duas tarefas.

Observando-se o cenário apresentado na Figura 14, percebe-se que a Tarefa 2 aguarda (partes hachuradas) a Tarefa 1 terminar o recebimento e validação dos dados e o envio destes dados para, então, efetuar sua análise e gravação. Porém, a questão é: o que aconteceria se a taxa de recepção de mensagens pela Tarefa 1 fosse bem maior? Haveria um acúmulo de mensagens com dados enviadas à Tarefa 2, podendo chegar ao ponto de poder esgotar os *buffers* de mensagens. Assim, sabendo-se que, por exemplo, somente uma mensagem pode ser armazenada, seria necessário que a Tarefa 1 soubesse quando a Tarefa 2 já terminou o processamento de uma mensagem anterior. Assim, cria-se também uma dependência da Tarefa 1 em relação à Tarefa 2 conforme ilustrado na Figura 15.

- **Semáforos e Mutex**

Outro mecanismo de sincronismo comumente utilizado em software embarcado são os semáforos e uma variação denominada Mutex. Um semáforo, como o nome sugere, é um sinalizador que indica se determinado recurso está ou não disponível para utilização por determinada tarefa. Este mecanismo é utilizado quando há um recurso específico que é de interesse a mais de uma tarefa. Como o recurso só pode ser usado por uma tarefa por vez, deve-se evitar que duas ou mais tarefas façam uso deste recurso simultaneamente. Assim, a primeira tarefa a solicitar o recurso sinaliza o semáforo, o que irá indicar às demais que ele está em uso. As demais tarefas interessadas neste mesmo recurso podem bloquear aguardando sua liberação ou executar outras ações e fazer uma nova tentativa mais tarde. Quando a tarefa não precisar mais do recurso, ela irá retirar a sinalização do semáforo e, assim, outras tarefas poderão utilizá-lo.

Mutex (*Mutual Exclusion*) é um mecanismo de sincronismo entre tarefas semelhante aos semáforos. Ele possui, entretanto, um contador que permite o controle de acesso a mais de um recurso. Imaginando que haja três unidades de um determinado tipo de recurso de uso compartilhado, poderia ser empregado um mutex com contador igual a 3. O mutex será sinalizado quando três tarefas solicitarem estes recursos simultaneamente. Uma quarta tarefa encontraria o mutext sinalizado e teria que aguardar pela sua liberação.

Como afirmado anteriormente, as tarefas são fluxos de execução sobre uma mesma área de código de um processo. Tipicamente, cada o fluxo de execução de cada tarefa percorre uma região própria do código. Entretanto, ocorre que, em determinadas situações, uma ou mais tarefas percorrem uma mesma região de código. Quando isto ocorre, cria-se uma situação denominada **reentrância**. A reentrância pode não trazer dificuldade alguma à execução do software, porém há situações em que é necessário impedir que ela ocorra.

Pode-se considerar, por exemplo, uma situação na qual duas tarefas desejem acessar uma mesma função (ou método) para a leitura de um registrador. Esta função poderia, por exemplo, ler o valor de um registrador, armazenar seu valor em uma variável global (ou atributo de um objeto) e, a seguir, efetuar uma ação dependendo do valor lido. Caso a primeira tarefa inicie a execução da função e seja interrompida exatamente após armazenar o valor do registrador na variável global, poderia ocorrer da segunda tarefa executar a mesma função obtendo um novo valor do registrador e sobrescrevendo o valor armazenado pela primeira tarefa. Quando a primeira tarefa retornar sua execução, a decisão que ela irá tomar sobre o valor da variável global será diferente da prevista.

Para proteger determinadas regiões do código contra a possibilidade de reentrância utiliza-se o mecanismo de seções críticas (também chamadas regiões críticas). Marca-se um ponto de início e de fim da seção crítica, de forma que quando uma tarefa estiver em execução naquele trecho, ela nunca será preemptada.

As seções críticas podem, também, ser usadas com o propósito de dar prioridade máxima a uma determinada região do código. Como a tarefa nunca será preemptada naquela região, ela passa a ter a prioridade máxima de execução naquele ponto. Como este mecanismo afeta a ordem de prioridades das tarefas, ele deve ser usado de forma cautelosa e limitada a regiões que sejam realmente críticas no software.

1.3.4. Aspectos Temporais

Uma parcela significativa dos sistema embarcados possuem restrições temporais, ou seja, são sistemas embarcados que operam em tempo real, ou simplesmente sistemas em tempo real. Para esta classe de sistemas, uma falha temporal é tão severa quanto uma falha lógica, portanto, se um sistema em tempo real produzir um resultado cujo valor é correto mas este resultado foi produzido com atraso então o sistema falhou. Como muitos sistemas em tempo real controlam equipamentos críticos (como controle de vôos, por exemplo), uma falha temporal pode resultar em uma catástrofe.

As tarefas de um sistemas em tempo real são classificadas em tarefas com restrições temporais rígidas (*hard real-time*) , tarefas com restrições temporais firmes (*firm real-time*) e tarefas com restrições temporais flexíveis (*soft real-time*). Uma tarefa com retrições temporais rígidas jamais pode perder um prazo para gerar resultados. Se tal fato ocorrer o sistema como um todo falhou. As outras duas categorias permitem algum atraso, posto que estes resultados, mesmo atrasados, ainda tem algum valor ao usuário.

Os sistemas em tempo real devem ser planejados, desde o início, para atender as restrições temporais. É uma falácia desenvolver um sistema em tempo real, como se fosse um sistema convencional, e medir os tempos em que os resultados são gerados apenas no final do processo de desenvolvimento, e, então, tentar fazer correções visando atender aos requisitos temporais. Esta forma de tratar o problema das restrições temporais raramente produz o resultado desejado.

Um sistema em tempo real deve ser estruturado na forma de um conjunto de tarefas com restrições temporais conhecidas desde o início do projeto. Estas tarefas devem ter suas características temporais, em particular o seu tempo de execução (pior caso) e seu período (ou menor tempo entre execuções consecutivas) bem definidas. A execução das tarefas deve ser controlada por um escalonador próprio para sistemas em tempo real. Este tipo de escalonador está presente nos RTOS (ou então o núcleo não deveria ser classificado como *real-time operating system*). A maioria dos RTOS atuais baseia-se em níveis de prioridade para definir a ordem em que as tarefas do sistema são escalonadas.

Os núcleos operacionais podem ser preemptivos ou não-preemptivos. No primeiro caso, o núcleo tem total controle sobre qual tarefa utiliza o processador. A qualquer momento o núcleo pode interromper a tarefa em execução e substituí-la por outra. A interrupção da tarefa em execução é denominada preempção e a substituição é denominada chaveamento de contexto. Já o escalonador dos núcleos não-preemptivos tem o papel de selecionar a tarefa que irá utilizar o processador, mas, uma vez em execução, cabe exclusivamente à tarefa a liberação do processador.

Considerando que num sistema embarcado as tarefas cooperam entre si para produzir o resultado desejado, o escalonamento não-preemptivo (ou cooperativo) é uma alternativa válida, que se adequadamente utilizada também pode garantir o atendimento às restrições temporais. Um benefício do escalonamento não-preemptivo é sua relativa facilidade de programação, quando comparado aos sistemas que utilizam escalonamento preemptivo, pelo fato dos pontos de chaveamento de contexto serem definidos pelo programador.

Um modelo de estruturação de sistemas em tempo real está apresentado na Figura 16 [Renaux 93]. O leitor reconhecerá a semelhança óbvia com o modelo da estrutura dos sistemas embarcados.

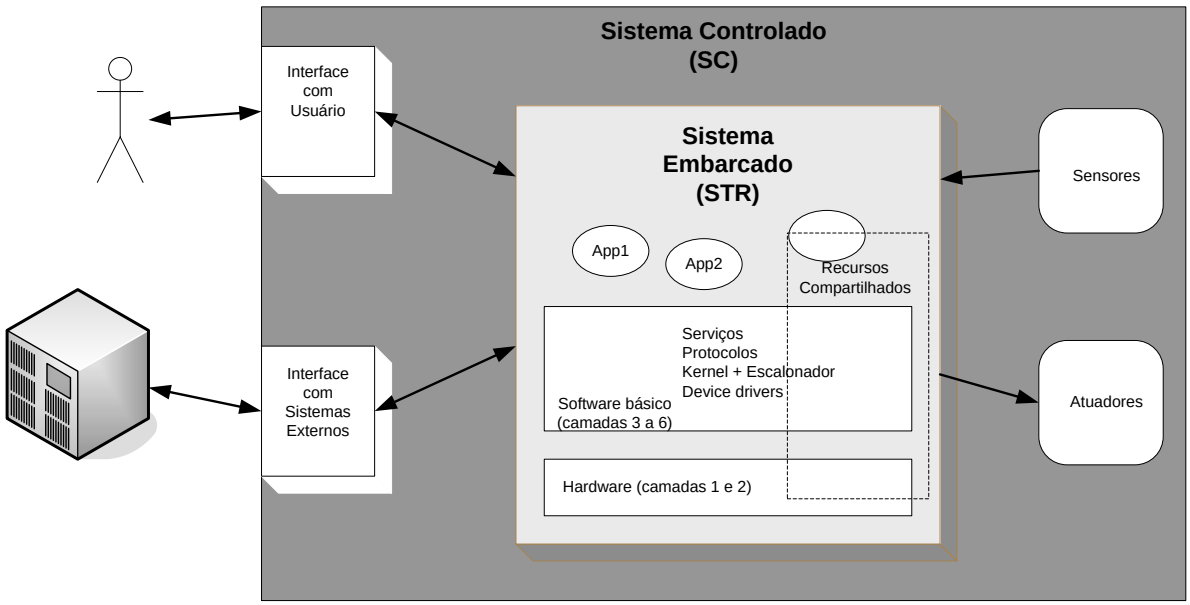


Figura 16 – Estrutura de um Sistema em Tempo Real

O sistema embarcado operado em tempo real (STR) é o responsável pelo controle do sistema no qual está inserido (SC). Este por sua vez, está inserido em um contexto sócio-técnico e interage com usuários e com sistemas técnicos através das interfaces apropriadas. Para interagir com o SC, o STR faz uso de sensores, que informam a situação corrente do SC e de atuadores, que permitem ao STR agir sobre o SC. A estrutura interna do STR é baseada no modelo de 7 camadas (Figura 3). O *hardware* consiste de processador, memória e dispositivos de entrada e saída, estes conectados aos sensores, atuadores, e interfaces externas. O *software* básico inclui os *device drivers*, o *kernel*, com seu respectivo escalonador de tarefas, protocolos e serviços. As tarefas de aplicação são as responsáveis por implementar a funcionalidade desejada do sistema e em atendimento às restrições temporais impostas a este. Nas diversas camadas do STR existem recursos compartilhados, sejam dispositivos físicos (*hardware*), estruturas de dados, ou tarefas servidoras de determinado serviço. O núcleo deve oferecer mecanismos que permitam acessar estes recursos compartilhados de forma organizada, respeitando o princípio da exclusão mútua.

Para ilustrar a utilização do modelo acima, considere o caso de uma impressora laser. O SC consiste de todo o mecanismo da impressora, incluindo o toner, o cilindro sobre o

qual um feixe laser projeta a imagem a ser impressa, e todo o conjunto de partes mecânicas, sensores, motores, engrenagens e eixos que compõem uma impressora. Inclui também o circuito eletrônico e o software que formam o STR. Os sensores (chaves ópticas, chaves mecâncias, etc.) e os atuadores (motores, controladores do feixe laser, etc.) dão acesso do STR à mecânica. A interface com o usuário, tipicamente, é simples nas impressoras laser, seus componentes podem inclui um display (LCD), alguns botões e LEDs. A interface com sistemas externos é formada pelos canais de comunicação que permitem a conexão da impressora a um computador, p. ex. Ethernet, USB, Serial e Paralela (Centronix).

1.3.4. Sistemas Operacionais / Kernels para Sistemas Embarcados

No contexto dos computadores pessoais (*notebooks, desktops e workstations*) existem poucas opções de sistemas operacionais. Já no contexto dos sistemas embarcados, há centenas de opções. Esta diversidade se justifica pela grande variedade de aplicações e de plataformas de hardware disponíveis.

Nesta seção serão apresentados três sistemas operacionais para uso em sistemas embarcados: o X Real-Time Kernel desenvolvido pela eSysTech (Brasil), o Windows CE da Microsoft (EUA) e o RTAI, uma variante do Linux desenvolvido pela comunidade de software livre. Estes três sistemas são representativos dos sistemas disponíveis.

1.3.4.1 X Real-Time Kernel

O núcleo operacional **X Real-Time Kernel** foi desenvolvido pela eSysTech – Embedded Systems Technologies (Brasil). Este núcleo tem características típicas dos núcleos desenvolvidos para uso nos chamados *deeply embedded systems*, ou seja, sistemas embarcados que são profundamente inseridos em equipamentos e que, portanto, não são percebidos pelo usuário como um sistema computacional. Exemplos de sistemas embarcados nesta categoria incluem: impressoras, computadores de bordo para veículos, sistemas de rastreamento, sistemas de alarme e segurança patrimonial, controladores de máquinas ferramenta, incluindo CNC (Controle Numérico Computadorizado), controle de equipamentos agrícolas e muitos outros.

Os núcleos operacionais para sistemas profundamente embarcados visam acomodar características que competem entre si: por um lado se quer oferecer uma grande variedade de serviços para as tarefas da aplicação, por outro, se deseja excelente desempenho, tanto no que se refere ao uso reduzido de espaço em memória como, também, no tempo reduzido de execução dos serviços. Um parâmetro de grande relevância neste caso é o tempo de chaveamento de contexto, já que este está associado ao mecanismo que permite apresentar às aplicações uma plataforma concorrente, porém, o chaveamento de contexto em si não é uma atividade relevante do ponto de vista da aplicação embarcada e, portanto, deve consumir uma parcela mínima do tempo do processador. A título de exemplo, o **X Real-Time Kernel** realiza chaveamentos de contexto entre tarefas em cerca de 5 microsegundos quando executando num processador ARM7TDMI à 66 MHz.

A tabela abaixo lista os serviços oferecidos pelo **X Real-Time Kernel** que são representativos dos núcleos na sua categoria.

Tabela 1 - Serviços do X Real-Time Kernel

Classif.	Método	Descrição
Thread Management and Scheduling	**CreateThread**	Criação de uma tarefa
	KillThread	Término de uma tarefa
	GetTId	Consulta ao identificador de uma tarefa
	Yield	Liberação do processador para outra tarefa
	SuspendThread	Mudança de estado de uma tarefa para suspenso
	ResumeThread	Mudança de estado de uma tarefa suspensa para pronto
Time Services	**GetTime**	Leitura do relógio do kernel
	MsgAt	Solicitação de envio de mensagem em hora programada
	PeriodicMsg	Solicitação de envio de mensagem periódica
	ClearMsg	Cancelamento de solicitação de envio de mensagem
	SleepFor / SleepUntil	Suspensão temporário da tarefa
Serviços de Comunicação Síncrona e Assíncrona	**Send**	Envio (síncrono) de mensagem
	Receive	Recepção de mensagens síncronas e assíncronas
	Reply	Resposta à mensagem síncrona
	Put	Envio (assíncrono) de mensagem
	CheckForMsg	Verificação de recebimento de mensagem
Serviços de Interrupção	**RegisterISR**	Cadastra uma rotina como *handler* de interrupção
	UnregisterISR	Cancela o cadastro de *handler* de interrupção
	MaskHWInt	Mascara um pedido de interrupção
	UnmaskHWInt	Desmascara (libera) um pedido de interrupção
	ConfigHWInt	Configura um pedido de interrupção
	InterruptLock	Bloqueia as interrupções
	InterruptUnlock	Desbloqueia as interrupções
Serviços de Sincronização	**Wait**	sincronização via semáforo
	Signal	sincronização via semáforo
	Get	consulta ao valor corrente do semáforo

As aplicações desenvolvidas com o **X Real-Time Kernel** seguem o modelo de 7 camadas sendo que os serviços do kernel ocupam a camada 4 – Microkernel e tarefas associadas (Figura 3). Um exemplo de tarefa associada é o gerente de temporização, uma tarefa interna do **X Real-Time Kernel** responsável por gerenciar as solicitações temporais (serviços SleepFor, SleepUntil, PeriodicMsg e MsgAt).

A Figura 17 apresenta a estrutura dos módulos que compõem o X Real-Time Kernel.

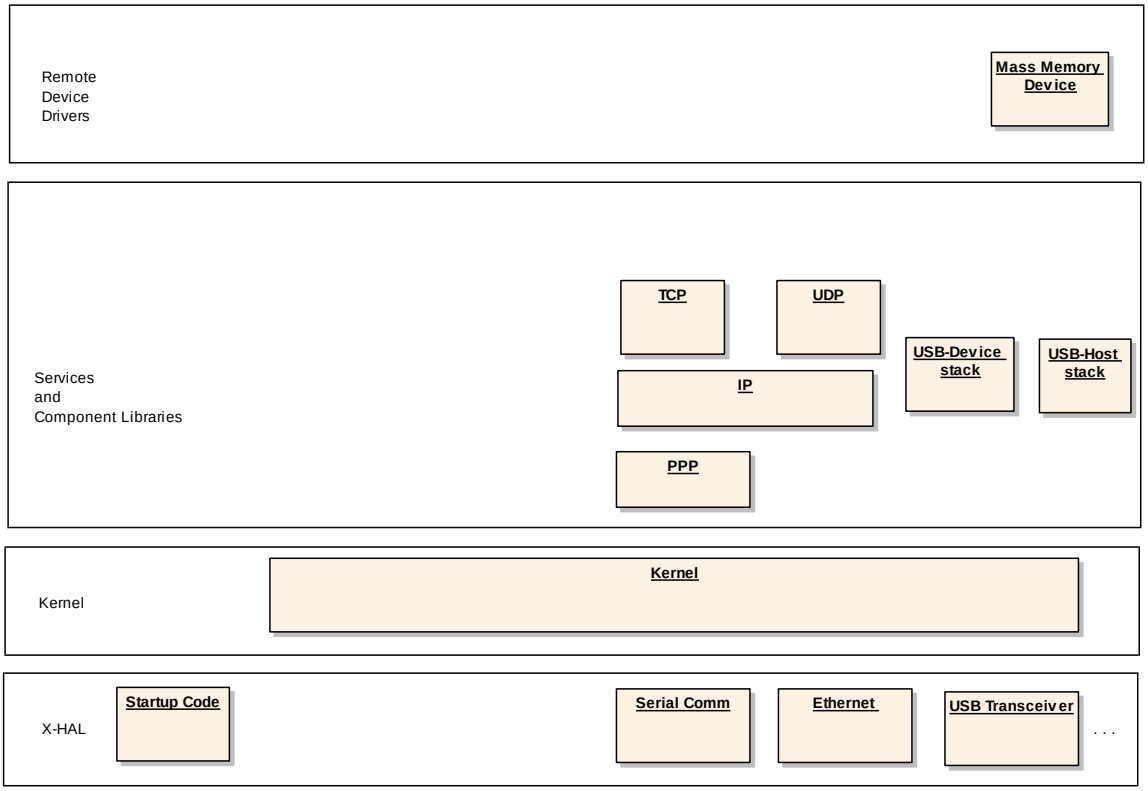


Figura 17 – Estrutura do X Real-Time Kernel.

A camada X-HAL (*Hardware Abstraction Layer*) implementa os *device drivers* dos dispositivos de *hardware* existentes no sistema. Esta camada, além de abstrair o *hardware*, também tem o importante papel de apresentar uma visão homogênea do *hardware* para as camadas superiores. Desta forma, as diferenças entre os dispositivos de *hardware* de diferentes sistemas são tratadas de forma localizada, sem interferir nas demais camadas.

A camada Kernel implementa os serviços do **X Real-Time Kernel**, listados anteriormente. Já os módulos opcionais do **X Real-Time Kernel**, que incluem uma pilha TCP/IP, uma pilha USB e um sistema de arquivos compatível com FAT, estão nas camadas superiores à do kernel.

```
int main( )
{
      os.Init( );
      os.CreateThread(SensorData,3,0,"SensorGas",1024,0, 7);
      // criação de outras tarefas
      os.Start( );
}

void SensorData(int sensor_id, int arg2)
{
      const int n_samples = 5;
      int s_data[n_samples];
      int i = 0;
      while (true) {
          os.SleepFor(20 * MSEC);
          s_data[i] = GetSensorValue( sensor_id );
          i = i < n_samples ? i+1 : 0;
          int res = Filter(s_data);
          os.Put(ProcessSensorData_TId,&res);
      }
}
```

Este exemplo apresenta de forma simplificada algumas das funcionalidade do **X Real-Time Kernel**. O programa principal deve, pelo menos, inicializar o *kernel*, criar algumas tarefas e dar início a execução do escalonador. A chamada os.Start( ) passa o controle do processador para o escalonador do *kernel* sem jamais retornar. Quando a tarefa SensorData é criada ela recebe até dois parâmetros. Neste exemplo apenas o primeiro parâmetro é utilizado, no caso para identificar o sensor a ser lido.

A tarefa utilizada neste exemplo executa periodicamente, em intervalos de 200 ms. A cada execução, coleta dados de um sensor e armazena este valor em uma fila circular de tamanho cinco. Estas cinco amostras são utilizadas pela rotina de filtragem para gerar um valor de sensor, que é então transferido para uma tarefa, cujo identificador é ProcessSensorData_TId, através do envio de uma mensagem assíncrona.

1.3.4.2 Windows CE

Windows CE, ou apenas WincCE, é um sistema operacional desenvolvido pela Microsoft para sistemas embarcados. Embora haja semelhanças com outros sistemas Windows produzidos pela Microsoft para PCs, o Windows CE é um sistema inteiramente desenvolvido para dispositivos de menor porte. As primeiras versões do Windows CE (em especial até a versão 2) focaram aplicações em dispositivos de mão como *hand-*

helds. As versões posteriores passaram a disponibilizar uma estrutura de *microkernel* própria para customização em sistemas embarcados, permitindo um emprego muito mais amplo. O Windows CE tornou-se, então, um sistema operacional em tempo real, modular e baseado em componentes.

Um aspecto importante do Windows CE, se comparado com as versões do Windows para PCs, é que este sistema operacional não é disponibilizado pronto para instalação. O desenvolvedor do sistema embarcado deve realizar sua customização gerando uma versão própria da imagem (versão customizada e compilada) deste sistema operacional. Para esta customização, também chamada configuração, o desenvolvedor emprega uma ferramenta da Microsoft denominada Platform Builder (até a versão 5.0 do Windows CE) ou Visual Studio (a partir da versão 6.0). Através da ferramenta de configuração, o desenvolvedor define a plataforma de hardware de seu sistema embarcado e quais componentes formarão a imagem final. As plataformas de hardware suportadas atualmente são X86, ARM, SH (Super Hitachi) e MIPS. Com relação aos componentes, existe um catálogo com uma extensa lista de *drivers*, aplicativos, utilitários, APIs, protocolos e servidores que podem ser selecionados para compor a configuração desejada.

A mesma ferramenta de configuração inclui, também, editores, compiladores e depuradores que permitem o desenvolvimento de programas para a imagem do Windows CE a ser produzida. Através desta ferramenta é feita também a construção (*build*) da imagem gerando um único arquivo final denominado NK.BIN que contém toda a imagem com seus componentes. Este único arquivo é, então, gravado em uma unidade de armazenamento (HD, memória *flash*, *compact flash*, ou outra) do sistema embarcado a partir da qual será feita a carga do sistema (*boot*) quando ele for ligado.

Deve-se notar, portanto, que cada imagem produzida do Windows CE é única no sentido de conter uma combinação exclusiva de configurações, componentes e programas de terceiros. A própria Microsoft desenvolve versões de imagens do Windows CE para propósitos particulares como o AutoPC, Pocket PC e Windows Mobile.

A ferramenta Platform Builder e, mais recentemente, o Visual Studio possuem também um ambiente de teste integrado. Este ambiente permite automatizar a verificação de funcionamento e robustez do sistema, exercitando a chamada de *drivers* e APIs. É possível, também, a criação de rotinas de teste adicionais que são incluídas nas seqüências de teste executadas pela ferramenta.

Um aspecto distintivo do Windows CE, se comparado aos demais sistemas operacionais do Microsoft, é o fato de que a maior parte de seu código fonte é disponibilizada ao desenvolvedor em modelo denominado *Shared Code*. Ou seja, o desenvolvedor adquire a ferramenta de desenvolvimento e recebe o código fonte de grande parte do sistema. O desenvolvedor pode acessar o código, alterá-lo e comercializar a imagem produzida com as alterações feitas. O código fonte, entretanto, não pode ser divulgado ou distribuído livremente.

A Figura 18 ilustra a arquitetura modular do Windows CE na versão 6.0 [MSDN, 2007]. Na camada superior encontram-se os programas de aplicação e eventuais *device drivers*

criados pelo desenvolvedor. Os programas de aplicação podem ser desenvolvidos em C/C++ e compilados para a plataforma de hardware específica do projeto em questão. Pode-se também desenvolver aplicações gerenciadas em .NET e em Java. O desenvolvedor pode também criar *driver* para novos dispositivos não suportados pelo Windows CE. Assim como para as aplicações, estes *drivers* são executados em Modo Usuário para permitir que as ferramentas de depuração e teste possam ser empregadas livremente. Os programas de aplicação e os *drivers* criados pelo desenvolvedor acessam os servidos do *kernel* do Windows CE através de chamadas à API do sistema implementada em uma biblioteca chamada Coredll. Através desta biblioteca as chamadas são encaminhadas aos módulos do sistema operacional.

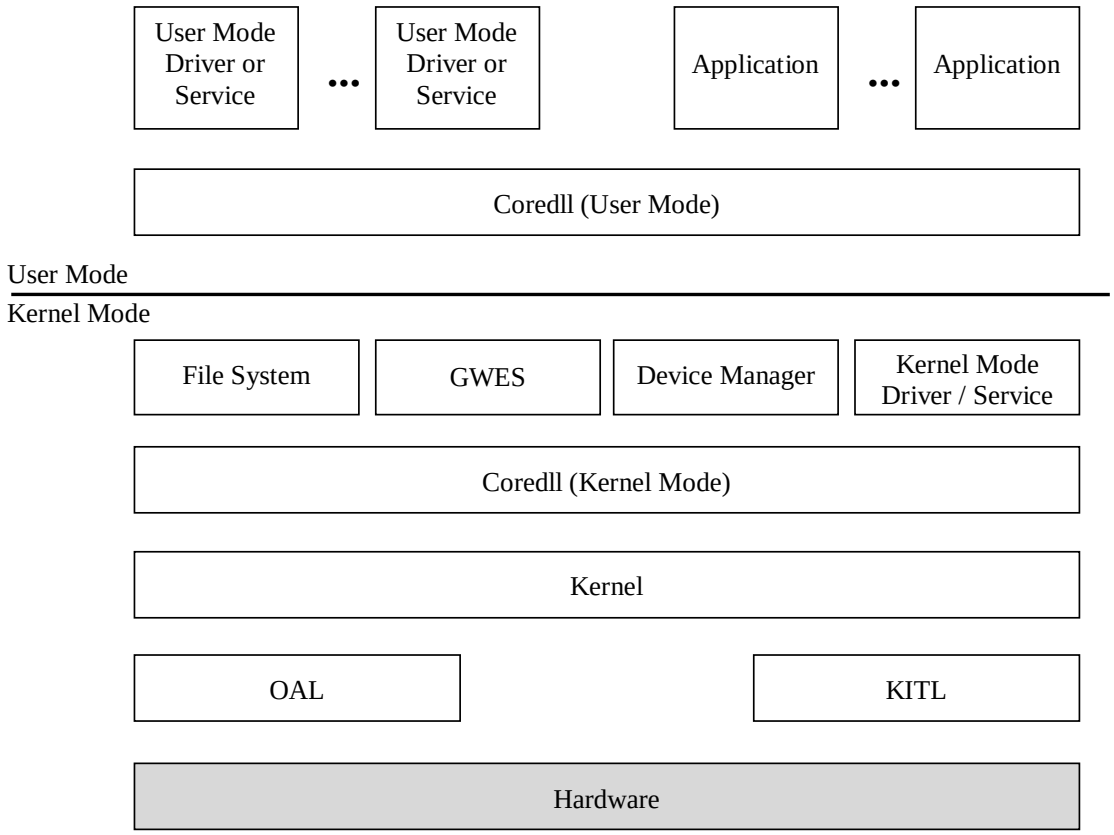
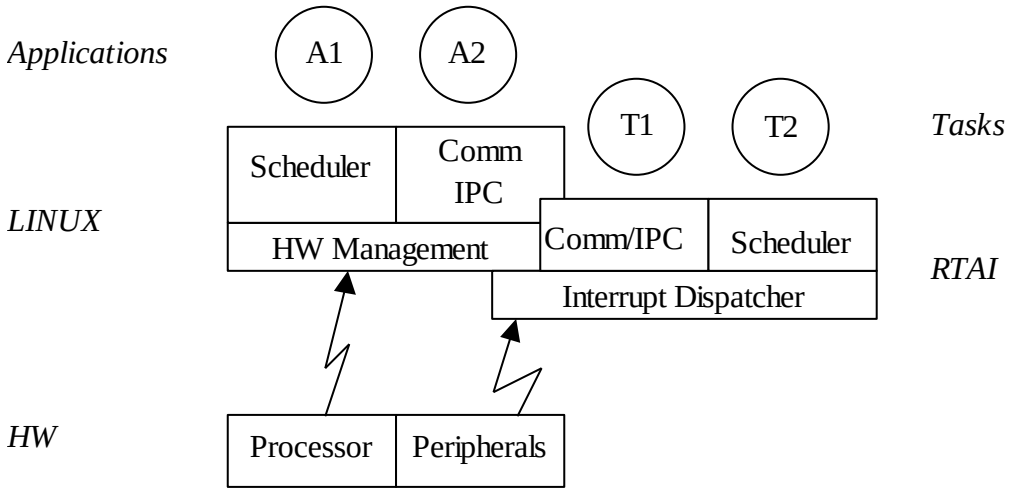


Figura 18 – Arquitetura do Sistema Operacional Windows CE 6.0 da Microsoft.

A parte central do Windows CE é composta pelo Kernel (responsável por tarefas como a gerência de memória e gerência de processos), pela Coreddl (biblioteca dinâmica do Kernel), pelo módulo File System (reponsável pelos sistemas de arquivo), pelo GWES (*Graphical, Windowing, and Event Subsystem*), pelo Device Manager (responsável pelo gerenciamento de dispositivos periféricos e carga de bilbiotecas e drivers) e pelos *device drivers* e serviços do sistema.

1.3.4.3 Linux (RTAI)

O RTAI (*Real-Time Application Interface*) [Bianchi, 1999] é uma das variantes do Linux desenvolvido para utilização em sistemas embarcados com restrições temporais. Considerando que o sistema operacional Linux não é apropriado para uso em sistemas de tempo real, devido principalmente a latência elevada no atendimento às interrupções e variabilidade temporal dos seus serviços, a solução adotada no desenvolvimento do RTAI, e de diversas variantes do Linux para uso em tempo real, foi a implementação de um *micro-kernel* abaixo do núcleo convencional do Linux. Desta forma, todo o Linux é tratado como uma tarefa deste *micro-kernel* e pode ser interrompido a qualquer momento. As tarefas com restrições temporais são executadas sobre o *micro-kernel* (T1 e T2 na Figura 19). Já as tarefas convencionais (A1 e A2 na Figura 19), que fazem uso dos serviços do Linux, executam sobre a tarefa Linux.



Fonte: http://www.aero.polimi.it/~rtai/documentation
Figura 19 – Arquitetura do RTAI

O tratamento de interrupções é realizado inicialmente pelo *micro-kernel*, que, quando necessário, encaminha o restante do tratamento para a tarefa Linux. Para permitir a integração entre as tarefas de tempo real, que executam sobre o *micro-kernel*, e as tarefas convencionais, que executam sobre o Linux, foram introduzidos uma série de mecanismos de comunicação entre tarefas. O RTAI disponibiliza ainda serviços para a gerência de memória, compatíveis com o padrão POSIX, e uma interface para que processos em modo usuário (ou seja, não em modo *kernel*) possam *executar* com restrições temporais.

O RTAI foi originalmente desenvolvido no Departamento de Engenharia Aeroespacial do Politécnico de Milano (Itália) em 2000 e desde então tem recebido colaborações da comunidade de software livre.

1.4. Desenvolvimento de Software Embarcado

Historicamente, os softwares para sistemas embarcados foram sempre desenvolvidos por projetistas de hardware (tipicamente, técnicos e engenheiros em eletrônica), uma vez que somente eles eram capazes de compreender e lidar com a intricada e específica forma de

operação dos circuitos por eles projetados. Com o aparecimento do microcontrolador, passou a existir uma maior padronização do software embarcado organizados na forma de programas contendo seqüências de instruções. Entretanto, pela própria carência de uma formação melhor em engenharia de software, código produzido pelos engenheiros em sistemas digitais era, freqüentemente, composto de milhares de linhas compondo seqüências intermináveis de instruções de difícil entendimento e manutenção [Ganssle, 1992].

Mais recentemente, com o aumento crescente da demanda e da complexidade dos sistemas embarcados percebe-se um crescimento correspondente na especialização das funções dentro das equipes de desenvolvimento. A tarefa de desenvolvimento do software do sistema embarcado começa a ser realizada por engenheiros com maior especialidade em engenharia de software e capazes de empregar metodologias e técnicas mais apropriadas. Neste sentido, o engenheiro de computação é um bom exemplo de profissional capaz de assumir este papel nos times de projeto de sistemas embarcados. O desenvolvedor de software embarcado, em todos os casos, deverá sempre ter conhecimento tanto de software quando de aspectos mais ou menos aprofundados do hardware do sistema.

O desenvolvimento de software embarcado, de uma forma geral, se baseia nos mesmos princípios de desenvolvimento de outros sistemas de informação. Considerando-se, entretanto, as especificidades dos sistemas embarcados, o desenvolvimento de software embarcado também possui alguns aspectos próprios, conforme discutido a seguir.

1.4.1. Processo de Desenvolvimento de Software Embarcado

Assim como para a engenharia de software em geral, não há um modelo de processo de desenvolvimento padrão para software embarcado. Apesar de datar dos anos 70, o próprio modelo clássico de processo de desenvolvimento, denominado modelo em cascata, pode ser empregado para desenvolver software embarcado. Ele define as fases ou atividades incontornáveis que devem ser conduzidas para o desenvolvimento de um software e determina as dependências causais diretas entre estas atividades, conforme ilustrado na Figura 20.

O modelo em cascata descreve um processo seqüencial no qual o resultado obtido em cada fase é utilizado como entrada para a próxima fase, criando-se assim dependências entre elas. Embora simples de ser compreendido, implementado e gerenciado, o modelo em cascata foi criticado ao longo dos anos e novas abordagens foram e têm sido propostas com o objetivo melhorar a eficiência do projeto de software e a qualidade dos produtos. Entre os métodos desenvolvidos estão: o modelo por prototipação, o modelo em espiral ou modelo por refinamentos sucessivos [Pressman, 1995], o modelo em V e, mais recentemente, o modelo ou processo unificado [Jacobson, 1999].

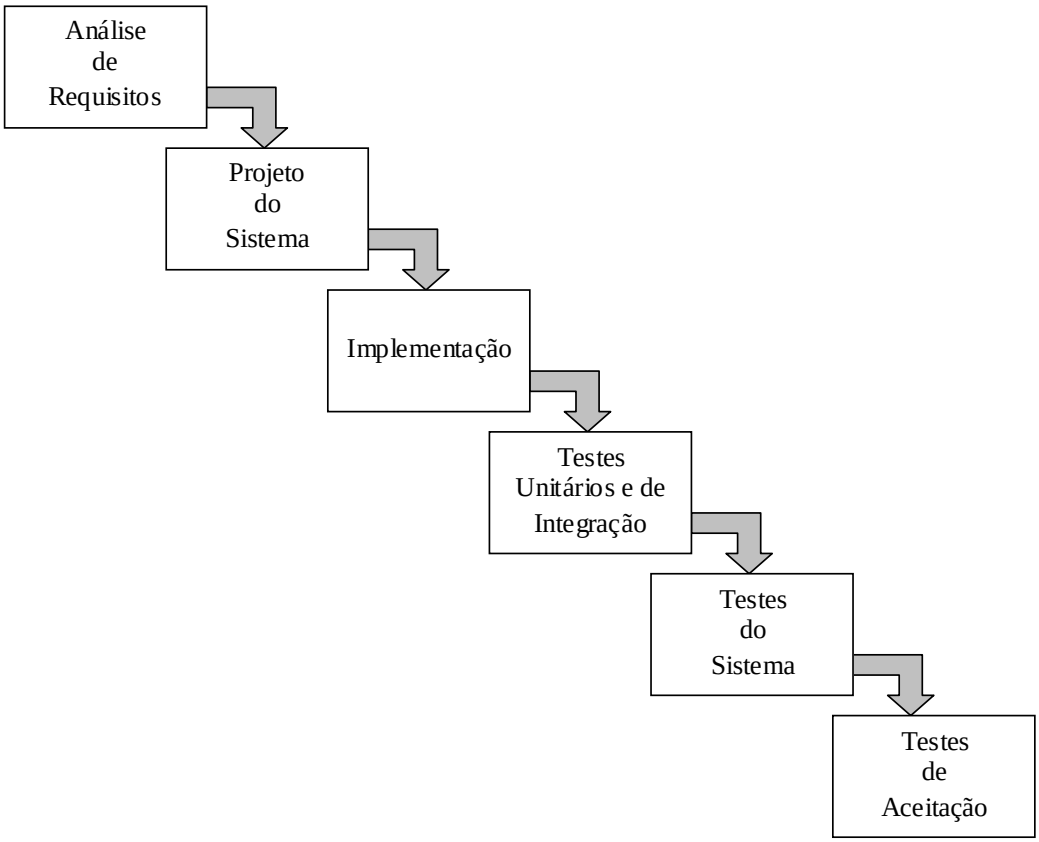


Figura 20 – Processo clássico de desenvolvimento empregando o modelo em cascata.

O **Modelo em V** foi proposto nos anos 80 e tem sido utilizado por empresas de desenvolvimento de software embarcado. Este modelo deriva do modelo em cascata com um foco maior nos testes e validações do software desenvolvido. A intenção é reduzir as dificuldades de reatividade do modelo em cascata em face de problemas ou falhas encontradas ao longo do desenvolvimento.

No modelo em V, há uma fase de validação para cada fase de desenvolvimento, conforme ilustrado na figura a seguir. Esta atenção maior com a validação das fases de desenvolvimento permite atender melhor as necessidades de garantia de robustez existente em sistemas embarcados.

A Análise de Requisitos representa um estudo do domínio de aplicação do software com o objetivo de identificar as necessidades do cliente. A intenção é determinar o que o software embarcado deverá fazer ou produzir como resultados. A descrição dos requisitos nesta fase é feita em termos da perspectiva do cliente, sem considerar explicitamente a tecnologia que será empregada para atender suas necessidades.

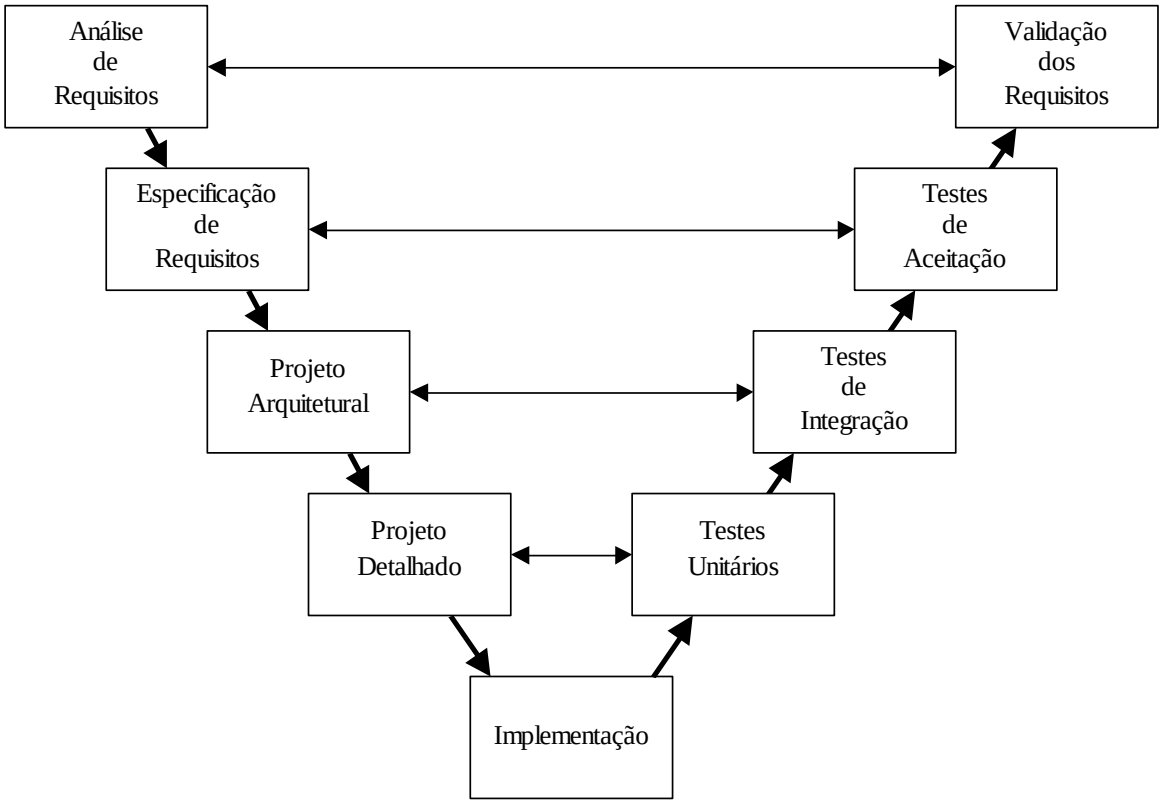


Figura 21 – Processo de desenvolvimento empregando o modelo em V.

Na fase de Especificação de Requisitos, a descrição dos requisitos em termos de necessidades do cliente é traduzida em uma descrição melhor estruturada das exigências impostas ao software empregando uma notação ou linguagem mais formal. São também diferenciados os requisitos funcionais (exigências em termos de operações ou serviços a serem executados) e os requisitos não funcionais (restrições relacionadas com as funcionalidades ou com o software como um todo).

O Projeto Arquitetural representa o desenvolvimento de uma solução de alto nível (solução abstrata ou genérica) para o software levando em conta todos os requisitos impostos. Nesta atividade, a solução é concebida e descrita apenas em termos de sua arquitetura, ou seja, em termos de seus componentes e da organização estrutural destes componentes.

Na fase de Projeto Detalhado, a arquitetura do software é refinada de forma a se descrever em detalhes todos os aspectos tanto da organização estrutural quando do comportamento dinâmico do software.

Após a fase de Implementação, inicia-se a fase de Testes Unitários cujo objetivo é testar os componentes do software isoladamente de maneira a validar a fase de Projeto Detalhado em termos das unidades que constituem o software. A seguir, na fase de Testes de Integração, são realizados testes de componentes maiores do sistema que representam agrupamentos das unidades menores já testadas na fase anterior. Através dos testes de integração é feita a validação do projeto arquitetural do software. Na fase seguinte são realizados os Testes de Aceitação através dos quais verifica-se o atendimento dos requisitos funcionais que foram impostos ao software. Por fim, na fase

de Validação dos Requisitos, o software é testado junto ao cliente no sentido de verificar que o software atende as necessidades do cliente.

Um exemplo de processo de desenvolvimento criado propriamente para sistemas concorrentes e em tempo real é o **COMET** – *Concurrent Object Modeling and Architectural Design with UML*, porposto por Hassan Gomaa [Gomaa, 2000]. Este é um processo de desenvolvimento de software orientado que é compatível com Processo Unificado desenvolvido por Ivar Jacobson, Grady Booch e James Rumbaught [Jacobson, 1999].

COMET é um processo iterativo baseado no conceito de casos de uso. Os requisitos funcionais são definidos em termos de atores e casos de uso, de forma que cada caso de uso descreve uma seqüência de interações entre os atores e o sistema. A Figura 22 apresenta uma visão geral deste processo.

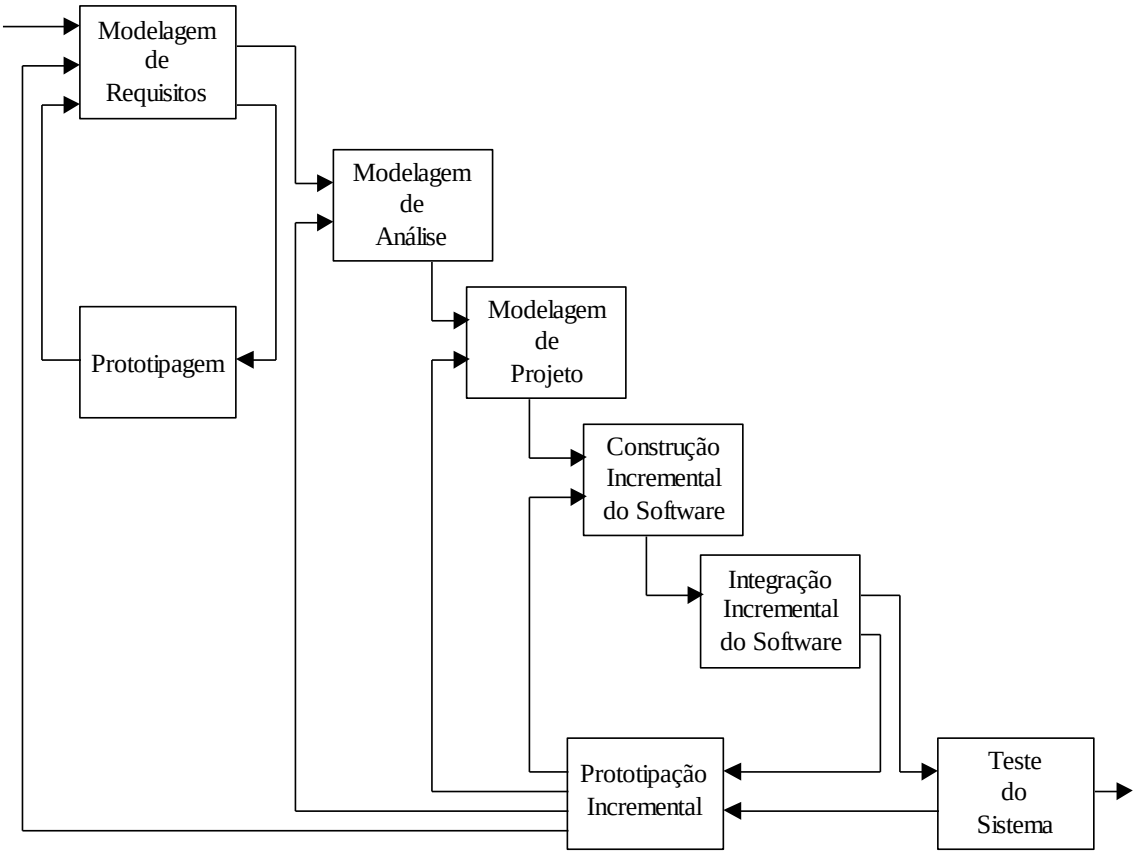


Figura 22 – Processo de desenvolvimento COMET.

Determinados projetos de software embarcado podem envolver atividades particulares em função das características ou requisitos do projeto como, por exemplo, em sistemas críticos. Em razão da área de aplicação, podem existir normas que impõem determinadas atividades dentro do processo de desenvolvimento. Um exemplo é o desenvolvimento de software para aviônica que, para homologação do produto, deve atender a norma DO-178 (*Software Considerations in Airborne Systems and Equipment Certification*) do RTCA (*Radio Technical Commission for Aeronautics*) [LinuxWorks, 2007]. Esta norma

é extremamente severa exigindo que diversos documentos e resultados sejam produzidos ao longo do processo de desenvolvimento. Estes documentos e resultados visam garantir que todas as recomendações tenham sido atendidas.

Em termos de planejamento, os documentos exigidos são:

- *Plan for software aspects of certification* (PSAC)
- *Software development plan* (SDP)
- *Software verification plan* (SVP)
- *Software configuration management plan* (SCMP)
- *Software quality assurance plan* (SQAP)
- *System requirements*
- *Software requirements standard* (SRS)
- *Software design standard* (SDS)
- *Software code standard* (SCS)

Em termos de desenvolvimento do software, os documentos e resultados exigidos são:

- *Software requirements data* (SRD)
- *Software design description* (SDD)
- *Source code*
- *Executable object code*

Em termos de verificação do software, os documentos exigidos são:

- *Software verification cases and procedures* (SVCP)
- *Software verification results* (SVR)

Em termos de garantia de qualidade, os documentos exigidos são:

- *Software quality assurance records* (SQAR)
- *Software conformity review* (SCR)
- *Software accomplishment summary* (SAS)

Em termos de gerência de configuração, os documentos exigidos são:

- *Software configuration index* (SCI)
- *Software life cycle environment configuration index* (SECI)

Embora a norma DO-178B não estabeleça, formalmente, um processo de desenvolvimento de software, os documentos exigidos e as dependências estabelecidas desenham um arcabouço de processo que pode ser acomodado a processos de desenvolvimento existentes.

1.4.2. Linguagens e Ferramentas de Desenvolvimento

Algumas peculiaridades dos sistemas embarcados precisam ser consideradas quando se seleciona uma linguagem de programação:

- restrição de área de memória disponível tanto para código como para dados;
- acesso ao hardware, em particular aos registradores dos dispositivos de entrada e saída;
- restrições temporais impostas ao software embarcado;

Conseqüentemente, as linguagens de programação para sistemas embarcados precisar ser muito eficientes no que tange ao volume de código gerado e ter um comportamento conhecido com relação a alocação de memória durante a execução. Precisam prover mecanismos de acesso a endereços específicos de memória (para dispositivos mapeados em memória) ou na região de endereçamento de entrada / saída (para dispositivos mapeados em E/S). Precisam, ainda, ter tempos de execução conhecidos ou determináveis.

Estas características são típicas não apenas dos sistemas embarcados, mas, do software básico de forma geral. Portanto, é natural que as linguagens utilizadas com mais freqüência no desenvolvimento de sistemas embarcados sejam justamente as linguagens desenvolvidas para programação de software básico: C e *assembly*.

A linguagem *assembly* é praticamente obrigatória no desenvolvimento de qualquer sistema embarcado. Sendo formada pelo conjunto de instruções do processador utilizado, sempre será a linguagem que dá ao programador acesso total a todos os recursos oferecidos pelo processador. As principais desvantagens da linguagem *assembly* são a baixa portabilidade e a complexidade de programação. A linguagem *assembly* é específica de cada arquitetura, portanto, código escrito em *assembly* só pode ser utilizado em projetos de sistemas embarcados que utilizam aquela arquitetura. Por outro lado, a linguagem *assembly* é uma linguagem de baixo nível, no sentido de não oferecer uma abstração do *hardware*, na verdade ela expõem o *hardware* como ele efetivamente é. Desta forma, programas desenvolvidos em *assembly* tem seu código fonte naturalmente mais extenso e mais complexo do que códigos equivalentes escritos em linguagens de alto nível.

Nas primeiras décadas de desenvolvimento de sistemas embarcados era comum encontrar sistemas totalmente desenvolvidos em *assembly*. A funcionalidade exigida do software era suficientemente restrita e de complexidade suficientemente reduzida de forma a permitir este desenvolvimento. Além disso, compiladores de linguagens de alto-nível para arquiteturas comumente utilizadas em sistemas embarcados eram raros, relativamente custosos e freqüentemente geravam código com desempenho (tanto em velocidade como em espaço de memória) inferior ao dos programas desenvolvidos em *assembly*.

Atualmente, apenas uma pequena parcela do código de um sistema embarcado é desenvolvido em *assembly*. A maior parte do código é desenvolvida em linguagens de alto-nível, em particular, C e C++ (ver Figura 11).

A linguagem C tem sua história bastante ligada ao desenvolvimento do sistema operacional UNIX. Sendo uma linguagem desenvolvida especificamente para a área de

software básico é natural que tenha os recursos necessários para acessos ao hardware. Desta forma, a linguagem C se transformou na principal linguagem utilizada para o desenvolvimento de sistemas embarcados. Além do benefício de ser uma linguagem de alto-nível, permitindo ganhos significativos de produtividade de programação, a linguagem C também traz ganhos no que se refere a portabilidade de código. Atualmente é impensável que uma nova arquitetura seja lançada sem que haja compiladores C desenvolvidas para ela.

A linguagem C++ vem ganhando espaço entre os desenvolvedores de sistemas embarcados. Por um lado traz todos os benefícios de C, e acrescenta os benefícios da orientação a objetos, resultado em vantagens tanto do ponto de vista de produtividade, como de reuso de código.

Duas outras linguagens que merecem destaque são Java e Ada. O número de desenvolvedores de sistemas embarcados que utilizam Java tem aumentado ano a ano. Java é uma linguagem que se adapta muito bem para sistemas embarcados. A principal restrição ainda é o uso intensivo que os programas em Java fazem da memória de alocação dinâmica (Heap), o que nem sempre é adequado para sistemas com restrições de espaço em memória.

A linguagem Ada foi desenvolvida como uma linguagem para desenvolvimento de sistemas embarcados, principalmente para os que possuem restrições temporais (sistemas em tempo real). Diferente de C e C++, a linguagem Ada permite explicitar as tarefas concorrentes que compõem um sistema embarcado, assim como os seus aspectos temporais. O fato de Ada ter nascido no mundo dos sistemas militares (seu desenvolvimento foi fomentado pelo Departamento de Defesa do governo americano) acabou restringindo o uso desta linguagem a um certo nicho do mercado.

O ambiente de desenvolvimento de sistemas embarcados se distingue do ambiente de desenvolvimento de software para computadores pessoais por uma razão determinante: é formado por uma máquina hospedeira (*host*) e uma plataforma alvo (*target*) que é o sistema embarcado ou uma variante deste. Na máquina hospedeira ocorre a edição, compilação e montagem (*link*) do código. O código binário gerado neste processo é transferido para a plataforma alvo e executado nesta, sob gerenciamento da ferramenta de depuração (*debugger*) que executa na máquina hospedeira. A justificativa para este ambiente composto (host + target) é que a plataforma alvo em geral não possui os recursos necessários (capacidade de processamento, capacidade de memória, sistema de arquivos, interface homem-máquina) necessários ao processo de edição, compilação e montagem de código.

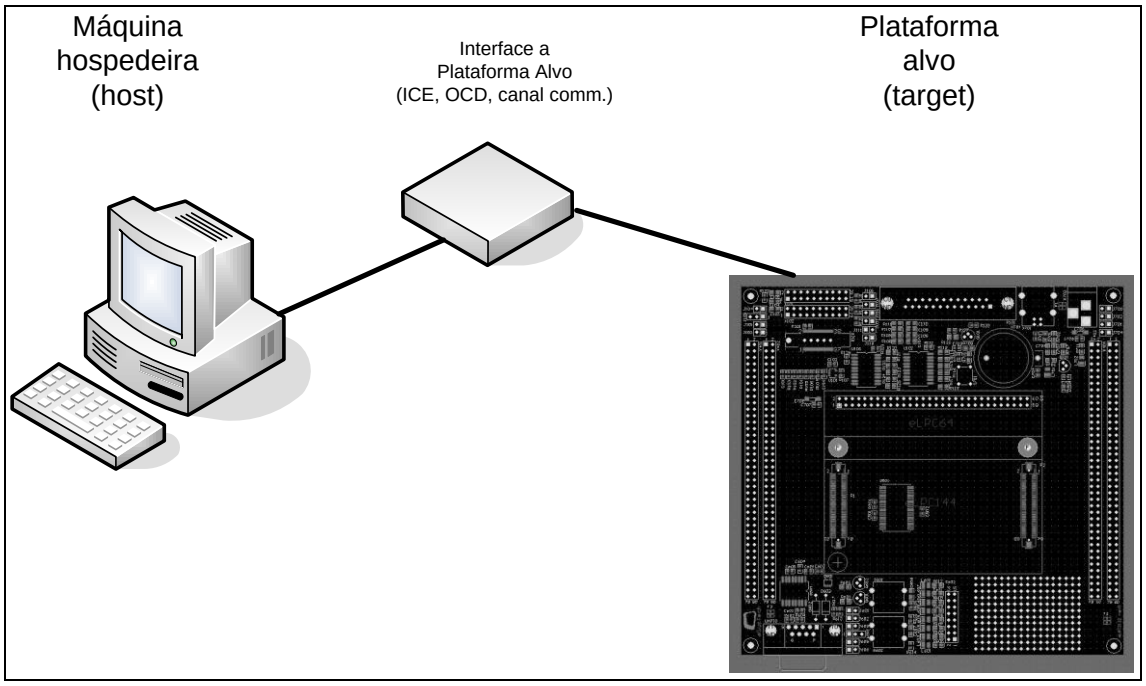


Figura 23 – Ambiente de Depuração

Na máquina hospedeira se faz necessário o uso de *cross-compilers* e *cross-assemblers*, ou seja, compiladores que executam na máquina hospedeira (por exemplo um computador Pentium e portanto com arquitetura x86) mas que geram código binário para uma outra arquitetura, por exemplo ARM. Os códigos objeto resultantes dos processos de compilação e montagem (*assembly*) são então ligados (*linked*) formando um código binário que deve ser transferido para a plataforma alvo.

Existem diversas formas de transferência do código binário da máquina hospedeira para a plataforma alvo, estas se dividem em transferência através de uma conexão física entre a máquina hospedeira e a plataforma alvo, e a transferência sem conexão física.

A conexão física é a forma preferida por dar muito mais flexibilidade e eficiência no processo de depuração. As formas de conexão físicas entre máquina hospedeira e plataforma alvo mais comuns são:

- ICE – In-Circuit Emulator, um equipamento que substitui o processador da plataforma alvo permitindo uma interação deste com a máquina hospedeira.
- OCD – On-Chip Debug, é a principal forma de conexão física utilizada atualmente, por seu baixo custo (comparada ao ICE) e grande flexibilidade; muitos processadores utilizam a interface JTAG para dar acesso a funcionalidade de OCD.
- canal de comunicação serial (RS-232, USB, Ethernet, ...) – permitem a transferência de dados entre a máquina hospedeira e a plataforma alvo. Tem a desvantagem de exigir que um software (programa monitor) esteja em execução na plataforma alvo para que a comunicação possa se realizar.

A transferência sem conexão física só é utilizada quando a infra-estrutura necessária para as opções acima não está disponível. Neste caso, conecta-se à máquina hospedeira um dispositivo de armazenamento, por exemplo um disco rígido, ou um cartão de memória

(PC Card ou Compact Flash) ou ainda um programador de EPROM para permitir a gravação do código gerado na máquina hospedeira em algum dispositivo que possa ser fisicamente conectado a plataforma alvo. Terminada a gravação do dispositivo este é transferido à plataforma alvo que é então energizada e testada. A ausência de um canal de comunicação impede o uso de uma ferramenta de depuração executando na máquina hospedeira. A depuração precisa ser realizada sem uma ferramenta de apoio, o que dificulta significativamente o processo de depuração.

É comum se utilizar os ambientes integrados de desenvolvimento (IDE – *Integrated Development Environment*) que contém o editor, compilador, montador (*assembler*), ligador (*linker*), depurador (*debugger*) além de várias outras ferramentas de apoio, como por exemplo um navegador (*browser*) que permite encontrar rapidamente o código fonte de uma função específica, ou a definição de uma classe, ou uma referência a um objeto.

1.4.3. Depuração e Teste de Software Embarcado

As atividades de depuração e teste normalmente ocorrem concorrentemente e sucedem a codificação. No processo de desenvolvimento segundo o modelo em V (Figura 21), assim que módulos de software são codificados eles são submetidos ao teste de unidade. Neste teste, um módulo de software é exercitado por código desenvolvido especificamente com a finalidade de teste. Para a geração deste código de teste pode-se fazer uso de ferramentas de apoio, que, baseadas no planejamento do teste de unidade, geram casos de teste adequados. Para cada caso de teste espera-se um determinado resultado, gerado pelo módulo em teste. A verificação do resultado real com o resultado esperado também pode ser feita com apoio de ferramentas de teste com tal finalidade. Quando o resultado real diferir do resultado esperado, ou seja, quando se identifica uma falha (*failure*) então tem início o processo de depuração, que tem por objetivo identificar a falta (*fault*) que causadora desta falha. Para tal, é de fundamental importância a disponibilidade de uma ferramenta de apoio a depuração, ou seja, o depurador (*debugger*).

A funcionalidade básica de um depurador inclui a possibilidade de controlar a execução das instruções e a possibilidade de examinar o conteúdo das variáveis em memória. Para controle da execução utiliza-se um modo de execução denominado passo-a-passo (*single step*) onde uma única instrução do processador é executada a cada comando do usuário. Também é possível inserir ponto de parada (*breakpoints*) ao longo do código. Sempre que a execução está parada é possível examinar o conteúdo dos registradores do processador bem como o conteúdo da memória e desta forma identificar em qual trecho de código está a falta.

Os depuradores atualmente em uso possuem uma variedade de funcionalidades adicionais, incluindo: examinar estruturas de dados complexas em memória, seguir ponteiros, acessar o ponto do código fonte contendo determinada estrutura ou código, executar seqüências de comandos armazenados em *scripts* ou *macros* e apresentar de forma gráfica informações sobre o programa em execução.

Quando se trata do teste de sistemas embarcados que operam em tempo real, e que, portanto, tem restrições temporais, o teste e, particularmente, a depuração se tornam bem mais complexos. Geralmente, o sistema não pode mais ser executado em modo passo-a-passo ou com pontos de parada. Até mesmo a inserção de mensagens (por

exemplo utilizando um **printf**) interfere no comportamento temporal a ponto de invalidar o processo de depuração.

Nestes casos, o núcleo operacional pode oferecer uma ferramenta de apoio que consiste em coletar dados históricos sobre a execução para análise posterior. Os três RTOS apresentados anteriormente dispõem deste tipo de apoio.

1.5. Conclusões

Os sistemas computacionais embarcados estão cada vez mais presentes na sociedade moderna. Sua ubiqüidade é óbvia, mas o fato de maior relevância é a dependência que nossa sociedade tem dos serviços prestados por estes sistemas, muitos deles vitais.

A cada nova geração de microprocessadores, sempre oferecendo mais desempenho e funcionalidades a custos menores, novas aplicações se tornam viáveis. Desta forma, o mercado de sistemas embarcados está em constante crescimento, a cada ano produzindo sistemas com maior grau de complexidade e de funcionalidades.

O software embarcado é um elemento essencial destes sistemas, já que implementa a maior parte da sua funcionalidade. O aumento da complexidade dos sistemas embarcados reflete-se diretamente no aumento da complexidade do software embarcado, e, desta forma, em imposições sobre o processo de desenvolvimento. As exigências sempre são no sentido de produzir mais em menos tempo, a menor custo e com maior qualidade e robustez.

Algumas tendências são claras. No que se refere ao software, os processos envolvendo reuso tem ganho destaque, em particular o desenvolvimento de software baseado em componentes. O objetivo é gerar componentes de software reutilizáveis que possam ser integralmente utilizados em um novo projeto evitando seu desenvolvimento e teste. No campo do hardware, a tendência tem sido da utilização de sistemas microprocessados utilizando mais de um processador, os chamados sistemas *multicore*. Estes, tanto podem ser homogêneos, quando todos os processadores são idênticos, como heterogêgeos, que congregam processadores especializados para diferentes papéis.

Tanto o desenvolvimento baseado em componentes, como os sistemas *multicore*, como as novas funcionalidades exigidas dos sistemas embarcados impactam sobre o perfil exigido dos profissionais atuando na área, que devem estar preparados para atender mais estas exigências em um área que está permanentemente em evolução rápida.

Referências

Bianchi E., Dozio L., Ghiringhelli G.L., Mantegazza P., (1999), “Complex Control Systems, Applications of DIAPM-RTAI at DIAPM”, Realtime Linux Workshop, Vienna.

Jacobson, I, Booch, G., Rumbaugh, J., (1999), The Unified Software Development Process, Addison Wesley.

Ganssle, J., Barr, M., (2003), Embedded Systems Dictionary, CMP Books, São Francisco.

Ganssle, J., (2006), “What processor is in your product?”, Embedded.com, http://www.embedded.com.

Gomaa, H., (2000), Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison Wesley.

Lee, E. A., (2002), “Embedded Software”, Advances in Computers, M. Zelkowitz, editor, Vol. 56, Academic Press, London.

LinuxWorks, (2007), “What is DO-178”, http://www.lynuxworks.com

Hennessy, J. L. and Patterson, D. A., (1992) Computer Architecture: A Quantitative Approach, Morgan Kaufmann.

Pressman, R. S., (1995), Engenharia de Software, Makron Books.

Renaux, D. P. B (1993), “RTX-Parlog: a Concurrent Logic Programming Language for Real-Time Systems”, Universidade de Waterloo, Canadá.

Renaux, D., (2005), Design of an Object-Oriented Operating System for Embedded Systems, LIT - Laboratório de Inovação e Tecnologia em Sistemas Embarcados, CEFET-PR, Curitiba, 2005.

Roman, D., (2006), “Software: what gets embedded”, Electronic Engineering Times, April, 2006.

Turley, J., (2005), “Embedded systems survey: Operating systems up for grabs”, Embedded.com, http://www.embedded.com.

Turley, J., (2006), “Operating Systems on the Rise”, http://www.embedded.com

Wolf, W. H., (2005), Computers as Components: Principles of Embedded Computer System Design, Elsevier.

MSDN Microsoft, (2007), Microsoft Development Network, http://msdn2.microsoft.com/en-us/embedded/aa731407.aspx