UM ESTUDO EXPLORATÓRIO SOBRE SISTEMAS OPERACIONAIS EMBARCADOS[1]

Thomas Tadeu Gallassi[2]
Prof. Luiz Eduardo Galvão Martins (Coautor)[3]

RESUMO

Nos últimos anos têm crescido muito o volume de *software* desenvolvido para sistemas embarcados. Com a diminuição do custo dos processadores de 32 bits, muitas aplicações embarcadas começaram a utilizar este tipo de processador, o que possibilitou o Desenvolvimento de *software* mais complexo e sofisticado. Com os processadores de 32 bits, o uso de sistemas operacionais embarcados se tornou uma realidade, principalmente a partir dos sistemas operacionais padronizados para aplicações embarcadas.
Este artigo aborda uma investigação exploratória sobre sistemas operacionais embarcados, englobando um estudo sobre as principais características dos sistemas operacionais embarcados, uma pesquisa de mercado identificando os sistemas mais utilizados, descrição das características dos sistemas mais populares, um comparativo entre eles, e pequenos experimentos explorando a utilização de um desses sistemas operacionais em uma aplicação de sistema embarcado.

Palavras-chave: Sistema Embarcado, Sistema Operacional, Microcontrolador

ABSTRACT

On recent years, the volume of *software* developed for embedded systems has greatly increased. With the costs decreasing of 32 bits processors, many embedded applications started to utilize that type of processor, that made possible the development of more complex and sophisticated *software*. With the 32 bits processors, the use of embedded operating systems became a reality, especially based on standard operating systems for embedded applications.
This article develops an exploratory investigation about embedded operating systems, including a study about the main features of embedded operating systems, a market study identifying the more utilized systems, description of the characteristics of the more popular systems, a comparison between them, and some case studies exploring the utilization of one of these operating systems in an embedded system application.

Keywords: Embedded System, Operating System, Microcontroller

[1] Artigo baseado em Relatório de Iniciação Científica apresentado pelo Curso Superior de Tecnologia em Análise de Sistemas e Tecnologia da Informação da Faculdade de Tecnologia de Americana – Fatec Americana depositado no 1º semestre de 2011
[2] Discente do 8º Semestre do Curso Superior de Tecnologia em Análise de Sistemas e Tecnologia da Informação da Faculdade de Tecnologia de Americana ; Contato: thomas_tadeu_gallassi@msn.com
[3] Prof.Dr. Fatec – Americana – Graduação em Processamento de Dados e Engenharia de Controle e Automação ; Mestrado em Informática e Doutorado em Engenharia Elétrica ; Contato: martinsleg@hotmail.com

1 INTRODUÇÃO

Sistemas embarcados são sistemas computacionais fisicamente limitados que são embutidos em outros produtos para melhorar seu funcionamento, estender a quantidade de tarefas que realiza ou mesmo permitir que funcione. Estes sistemas estão cada vez mais presentes na vida das pessoas, sendo encontrados desde equipamentos eletrodomésticos até sistemas automotivos de médio e grande porte. A gama de aplicações para este tipo de sistema é muito vasta e o custo do desenvolvimento tem recaído cada vez mais sobre o desenvolvimento do *software*.

Na medida em que os processadores utilizados na construção de sistemas embarcados se tornaram mais sofisticados e potentes, tornou-se viável o uso de sistemas operacionais para auxiliar o desenvolvimento de *software* embarcado.

Este artigo enfoca o uso de sistemas operacionais no desenvolvimento de sistemas embarcados, explorando as particularidades deste segmento de desenvolvimento de *software*. Este artigo também apresenta estudos de sistemas embarcados e suas características, comparativos dos sistemas operacionais embarcados e experimentos para entender melhor o desenvolvimento e utilização de aplicações embarcadas.

1.1 Contextualização

Sistemas embarcados têm despertado interesse em diversos segmentos de mercado nos últimos anos, principalmente através da evolução de tecnologias impulsionadoras deste tipo de sistemas, dentre as quais podemos destacar tecnologia de comunicação sem-fio, interfaces gráficas com o usuário final e acesso à Internet.

Porém, devido à curta vida dos produtos e prazos de desenvolvimento cada vez mais limitados, a padronização do processo de desenvolvimento está sendo essencial, o que leva à utilização de sistemas operacionais embarcados, que ajudam a padronizar e diminuir o tempo de desenvolvimento.

Existem várias denominações para sistemas embarcados, que podem ser encontradas na literatura e no mercado de desenvolvimento de tais sistemas. Sistemas embarcados também são conhecidos pelas denominações: Sistemas embutidos; Sistemas dedicados; Sistemas reativos; Sistemas dinâmicos; Sistemas de tempo-real (embora nem todo sistema embarcado seja um sistema de tempo-real). A definição do que seja um sistema embarcado é controversa, e dada a variedade de aplicações para tais sistemas, e do tipo de tecnologia empregada, é difícil conseguir uma definição geral.

No entanto, existe um conjunto de características que são comuns à maioria dos sistemas embarcados, independentemente do tipo de aplicação. Estas características serão estudadas em capítulos subsequentes.

1.2 Objetivo

O objetivo geral deste artigo é demonstrar um estudo exploratório sobre os principais sistemas operacionais utilizados no desenvolvimento de sistemas embarcados, descrevendo suas principais características e aplicações.

Os objetivos específicos deste artigo são:

- Discutir a importância do desenvolvimento de sistemas embarcados na atualidade;
- Apresentar a diferença entre os sistemas operacionais tradicionais e os sistemas operacionais embarcados;
- Identificar os sistemas operacionais mais utilizados;
- Fazer um comparativo entre os sistemas operacionais embarcados mais populares;
- Apresentar a importância do uso de sistemas operacionais no contexto do desenvolvimento de sistemas embarcados.

1.3 Justificativa

Este artigo insere-se dentro da área de sistemas embarcados, que tem chamado a atenção da comunidade de desenvolvimento de *software* nos últimos anos. Sistemas embarcados são potencialmente ricos, pois fazem parte de produtos extremamente variados, como eletrônicos de consumo, equipamentos automotivos, telecomunicações, equipamentos médicos, sistemas de automação, entre outros.

Os sistemas operacionais embarcados estão sendo cada vez mais utilizados, pois os processadores de 32 bits estão se tornando mais baratos e, portanto, mais acessíveis ao desenvolvimento de tais sistemas. O domínio dos sistemas operacionais embarcados tem se revelado um diferencial de mercado para os profissionais da área de TI que pretendem atuar no nicho.

1.4 Organização do artigo

O capítulo 2 apresenta os sistemas computacionais e sistemas operacionais, suas características e seus componentes. Estas informações são necessários para entender e comparar as diferenças com os sistemas embarcados e os sistemas operacionais embarcados.

O capítulo 3 apresenta os sistemas embarcados, suas características, seus componentes e seus requisitos, além de abordar os sistemas operacionais embarcados e suas características para comparar alguns sistemas operacionais embarcados, enquanto o capítulo 4 apresenta algumas ferramentas de desenvolvimento de sistemas embarcados exploradas durante os experimentos desenvolvidos;

O capítulo 5 apresenta os experimentos propriamente ditos, explicando seus objetivos e demonstrando seu desenvolvimento, e o capítulo 6 apresenta os comentários finais e a conclusão obtidos através da análise das referências bibliográficas e dos experimentos.

2 SISTEMAS OPERACIONAIS

Para que se possa falar sobre sistemas operacionais embarcados, é necessário conhecer um pouco sobre sistemas operacionais tradicionais de informática e seus conceitos, para assim entender as similaridades e diferenças entre ambos os tipos de sistemas. Portanto, neste capítulo são apresentados alguns conceitos sobre sistemas operacionais que embasam o uso de sistemas operacionais embarcados.

2.1 Características

Um sistema computacional moderno consiste em pelo menos um processador, memória principal, teclado, *mouse*, monitor, interfaces de rede, discos, impressoras, e outros dispositivos de entrada e saída de dados. Isso significa que desenvolver programas para manter o controle de todos esses componentes e os utilizar corretamente é extremamente difícil. Por isso foram criados os sistemas operacionais, cujo trabalho é gerenciar esses recursos e fornecer aos programas dos usuários interfaces com o *hardware* simplificadas. Com a evolução e ramificação dos tipos de computadores e de sistemas, foram surgindo também sistemas operacionais específicos e diversificados (TANENBAUM, 2003).

Como exemplificado na Tabela 1, um sistema computacional comum pode ser analisado e estudado em camadas.

Tabela 1 – Camadas de um Sistema Computacional

<table>
<tr><td colspan="3">Programas de aplicação</td><td></td></tr>
<tr><td>Compiladores</td><td>Editores</td><td>Interpretador de comandos</td><td rowspan="2">} Programas do sistema</td></tr>
<tr><td colspan="3">Sistema operacional</td></tr>
<tr><td colspan="3">Linguagem de máquina</td><td rowspan="3">} Hardware</td></tr>
<tr><td colspan="3">Microarquitetura</td></tr>
<tr><td colspan="3">Dispositivos físicos</td></tr>
</table>

Fonte: baseado em TANENBAUM, 2003

A camada superior de um sistema computacional constitui-se dos programas de aplicação, os quais englobam os programas de usuários e os programas utilitários. A segunda camada é formada pelos programas embutidos e dependentes do sistema operacional, mas que não são obrigatórios para o funcionamento do mesmo.

A terceira camada é o sistema operacional em si, que oculta parcialmente a complexidade das camadas de *hardware* e fornece aos usuários e aos programadores, um conjunto de instruções mais convenientes, protegendo assim o *hardware*. O sistema operacional e os programas embutidos, juntos, constituem o nível de programas do sistema.

A quarta camada é a de linguagem de máquina, que move os dados através de instruções recebidas e controla os dispositivos de entrada e saída de dados, carregando valores chamados de registradores de dispositivos. Por ser uma camada com um conjunto de instruções detalhadas, específicas e complexas, o sistema operacional é responsável por simplificá-las e agrupá-las para então fornecer aos programadores e aos usuários.

A quinta camada é a de microarquitetura, na qual os dispositivos físicos, da sexta camada, são agrupados em unidades funcionais. Já a sexta camada, por sua vez, é constituída dos dispositivos físicos, como chips de circuitos integrados, fios, fontes de alimentação, tubos de raios catódicos e dispositivos semelhantes. É importante notar que os chips de circuitos integrados podem ser considerados sistemas

embarcados, que serão estudados posteriormente. As três últimas camadas juntas constituem o nível de *hardware* de um sistema (TANENBAUM, 2003).

2.2 Funções

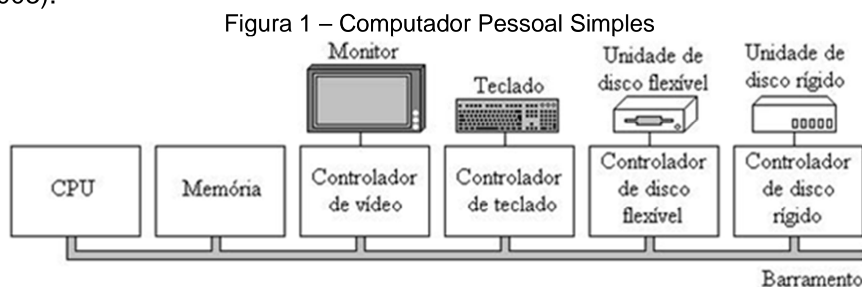
Um sistema operacional pode ser visto como uma máquina estendida, ou seja, um simplificador dos comandos de linguagem de máquina e do *hardware* e fornecedor de serviços, tornando mais fácil a programação, mas pode ser visto também como um gerenciador de recursos, que fornece uma alocação ordenada e controlada de processadores, memórias e dispositivos de entrada e saída de dados, além de assegurar que as requisições de vários programas ou mesmo de vários usuários para um mesmo recurso, sejam feitas de maneira segura e sem interferências (TANENBAUM, 2003).

2.3 *Hardware* de Computadores

Um sistema operacional está intimamente ligado e precisa ter um grande conhecimento sobre o *hardware* do computador no qual é executado, já que estende o conjunto de instruções do computador e gerencia seus recursos.

Conceitualmente, um computador pessoal simples pode ser abstraído para um modelo semelhante ao da Figura 1, onde a *CPU* (Unidade de Processamento Central do inglês *Central Processing Unit*), a memória e os dispositivos de entrada e saída de dados estão ligados por um barramento, o qual proporciona a comunicação entre eles. Computadores mais modernos, porém, podem possuir múltiplos barramentos (TANENBAUM, 2003).

Figura 1 – Computador Pessoal Simples



Fonte: TANENBAUM, 2003

2.3.1 Processadores

O processador é o componente principal do computador, ele possui uma ou mais *CPU*s, que são responsáveis por buscar instruções na memória e as executarem. O ciclo básico para isso é:

- Buscar instrução na memória;
- Decodificar a instrução para determinar seus operandos e qual operação executar;
- Executar a instrução;
- Prosseguir com as instruções subsequentes.

É assim que os programas são executados em um computador. Cada *CPU* possui um conjunto específico de instruções que consegue executar, e é por isso que alguns computadores não conseguem executar certos programas.

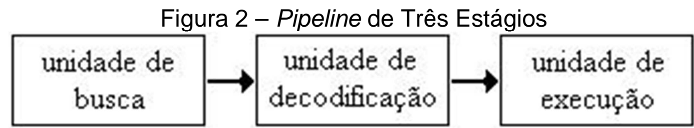
Além disso, como o tempo de acesso à memória para buscar instruções ou operandos é menor do que o tempo para executá-las, as *CPU*s possuem registradores internos para armazenamento de variáveis importantes e para armazenamento de resultados temporários.

Além desses registradores de propósito geral, há também os registradores especiais, que são importantes e visíveis aos programadores. Entre eles temos o contador de programa, que contém o endereço de memória da próxima instrução a buscar, o ponteiro de pilha, o qual aponta para o topo da pilha atual de memória. (As pilhas contêm estruturas para cada procedimento chamado, mas que ainda não encerrou. Uma estrutura de pilha contém parâmetros de entrada e as variáveis locais e temporárias que não são mantidas nos registradores) e a *PSW* (Palavra de Estado do Programa do inglês *Program Status Word*), que contêm os bits do código de condições, os quais podem ser alterados por instruções de comparações, pelo nível de prioridade da *CPU*, e por vários outros bits de controle.

Quando um sistema operacional compartilha o tempo de *CPU*, ele geralmente interrompe a execução de um programa e inicia a de outro. Toda vez que o sistema operacional faz isso, ele precisa salvar todos os registradores para que eles possam ser restaurados quando o programa voltar a ser executado.

As *CPU*s antigas usavam um modelo simples de busca, decodificação e execução de uma instrução por vez, mas as *CPU*s modernas possuem recursos para trabalhar em mais de uma instrução por vez, através de unidades separadas de busca, decodificação e execução de instruções. Um modelo de *pipeline* de três estágios está exemplificado na Figura 2.

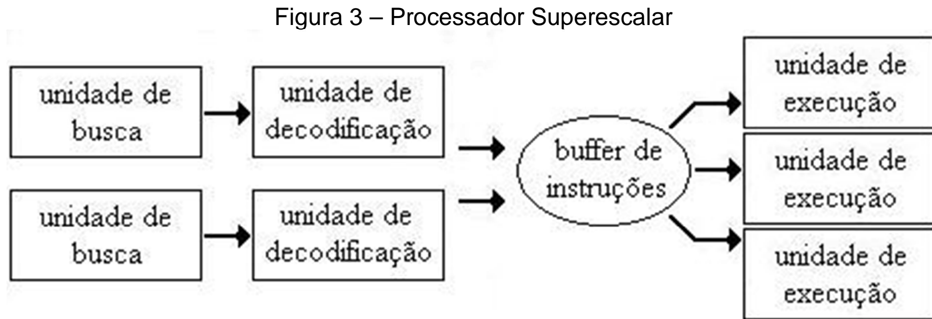
Figura 2 – *Pipeline* de Três Estágios



Fonte: baseado em TANENBAUM, 2003

Há também os processadores superescalares, que possuem múltiplas unidades de execução. Neste tipo de processador, múltiplas instruções são buscadas e decodificadas, e em seguida, temporariamente armazenadas em um *buffer* de instruções, até que possam ser executadas. Quando uma unidade de execução do processador estiver livre, se houver instruções no *buffer*, a próxima instrução será removida do *buffer* e será executada. Pode acontecer de instruções de um mesmo programa serem executadas fora de ordem, geralmente cabe ao *hardware* impedir que isso aconteça. Um modelo de um processador superescalar pode ser encontrado na Figura 3.

Figura 3 – Processador Superescalar



Fonte: baseado em TANENBAUM, 2003

A maioria das *CPU*s possuem dois modos de funcionamento, o modo núcleo, onde os programas possuem acesso a todas as instruções das *CPU*s e podem usar todos os atributos do *hardware*, e o modo usuário, onde o acesso e os atributos são restritos a apenas um subconjunto. O sistema operacional é sempre executado em modo núcleo, e se os programas precisarem de acesso a instruções ou atributos do *hardware* que são restritos, estes podem fazer uma chamada ao sistema, que alterna o modo usuário para modo núcleo e passa o controle para o sistema operacional. Quando a instrução é completada, o sistema operacional alterna novamente a *CPU* para o modo usuário e retorna o resultado para o programa que fez a requisição (TANENBAUM, 2003).

2.3.2 Memória

O segundo principal componente dos computadores é a memória. Idealmente, a memória deveria ser grande, barata e mais veloz que a execução de uma instrução (para que a *CPU* não fosse atrasada), porém nenhuma tecnologia atende esses requisitos, e por isso, a memória dos sistemas é construída baseada em uma hierarquia de camadas, onde as memórias mais rápidas, porém menores e mais caras, estão mais próximas (em termos de acesso) da(s) *CPU*(s), enquanto as que possuem maior capacidade de armazenamento encontram-se mais distantes.

Seguindo a ordem de acesso pela *CPU*, podemos construir em modelo em cinco camadas. No nível superior temos os registradores internos, que, por serem feitos do mesmo material, são tão rápidos quanto a *CPU*. Geralmente possuem menos de 1 *KB* (Kilobytes do inglês *Kylobytes*) de espaço. No segundo nível temos a memória *cache*, que é controlada principalmente pelo *hardware* e guarda as instruções que estão sendo muito usadas. Um computador pode ter múltiplos níveis de memória cache.

Em seguida temos a memória *RAM* (Memória de Acesso Aleatório do inglês *Random Access Memory*), que é a memória principal. As requisições que não são atendidas pela memória *cache* são procuradas na memória *RAM.*

O disco rígido vem em sequência e, por ser um dispositivo mecânico, é um dispositivo de memória não volátil, o que significa que não perde informações quando o computador é desligado (ao contrário das camadas discutidas anteriormente). Por último temos a fita magnética, outro dispositivo mecânico, e por isso, de memória não volátil, que serve como *backup* (cópia de segurança) e pode ser descartado para certos modelos de computadores. Geralmente é comum para computadores de empresas.

Há outros dispositivos de armazenamento que mudam ligeiramente o modelo de camadas específico para cada computador, como discos ópticos, *flash drives* etc.

Há ainda memórias especiais, como a *ROM* (Memória Somente de Leitura do inglês *Ready-Only Memory*), que é uma memória não volátil, programada de fábrica, que é rápida e barata, mas não pode ser

alterada (mas as variações, como a *EEPROM* (Memória Somente de Leitura Programável do inglês *Electrically Erasabel Programmable Read-Only Memory*) e flash *RAM* podem ser apagadas e reescritas, permitindo atualizações e correção de possíveis erros). A *ROM* é bastante usada como carregador, para iniciar um computador, mas pode estar presente em placas de dispositivos de entrada e saída para controle de baixo nível.

2.3.3 Dispositivos de Entrada e Saída de Dados

Os dispositivos de entrada e saída também interagem intensivamente com o sistema operacional. Eles são geralmente constituídos de duas partes: um controlador, e o dispositivo propriamente dito. Os dispositivos em si, possuem interfaces simples, porque não fazem nada muito diferente um dos outros, o que ajuda a padronizá-los.

Já o controlador é um *chip* ou um conjunto de *chips* em uma placa que controla fisicamente o dispositivo. Ele recebe comandos do sistema operacional para interagir com o dispositivo. Como o controle real de um dispositivo pode ser complexo, é tarefa do controlador oferecer uma interface simplificada ao sistema operacional.

Como cada controlador é diferente, diferentes programas são necessários para que o sistema possa controlá-los. Estes programas chamam-se *drivers* de dispositivo, e cada fabricante de controlador deve oferecer um *driver* apropriado para cada sistema operacional suportado. Majoritariamente, para funcionar, os *drivers* devem ser executados em modo núcleo e precisam estar dentro do sistema operacional.

Todo controlador possui um pequeno número de registradores, que são usados na comunicação. Para ativar um controlador, o *driver* recebe um comando do sistema operacional e o traduz em valores apropriados para serem escritos nos registradores do controlador (TANENBAUM, 2003).

2.4 Conceitos sobre Sistemas Operacionais

Não basta conhecer apenas o *hardware* de um sistema esperando entender o que é um sistema operacional, é necessário conhecer também os conceitos básicos de um sistema operacional. Esta seção aborda tais conceitos.

2.4.1 Processo

É um conceito fundamental para todos os tipos de sistemas operacionais. Um processo pode ser definido como um programa em execução.

Associado a cada processo temos seu espaço de endereçamento, uma lista de posições na memória as quais ele pode ler e escrever, e um conjunto de registradores (que inclui o contador de programas, o ponteiro de pilha, alguns registradores do *hardware* e todas as demais informações necessárias para executar o programa). O espaço de endereçamento, por sua vez, contém o programa executável, os dados do programa e sua pilha.

Em sistemas de tempo compartilhados, periodicamente, o sistema operacional interrompe e salva (geralmente em uma tabela de processos) todas as informações de um processo em execução, e começa a executar outro programa, reiniciando a execução do processo salvo mais tarde.

As principais chamadas ao sistema de gerenciamento de processos são aquelas que lidam com a criação e término de processos, mas incluem também requisição de mais memória, liberação de memória e sobreposição de um processo depois de seu término. Um processo pode criar outros processos (chamados de processos filhos), e assim podemos montar uma estrutura de árvore.

Processos que cooperam para realizar uma tarefa precisam frequentemente se comunicar e sincronizar suas atividades, o que é chamado de comunicação interprocessos. Um processo pode pedir ao sistema operacional para ser notificado caso não haja resposta de recebimento de uma mensagem (isso é mais comum entre processos que se comunicam através de uma rede de computadores), assim, quando o sistema operacional enviar o sinal de alarme para o processo interromperá e salvará tudo o que estiver fazendo e inicie um procedimento especial para tratamento desse sinal (geralmente para reenviar a mensagem).

Todo processo possui entre seus registradores, uma *UID* (Identificação de Usuário do inglês *User Identification*), que serve para identificar o usuário sob o qual o processo foi criado. Geralmente é usado como proteção, para impedir que usuários sem permissões especiais interfiram em processos de outros usuários.

Quando um processo está apenas esperando uma resposta de outro processo que, por algum motivo, não existe mais, o chamamos de processo zumbi, já que ele consome tempo do processador e não faz nada de útil ao usuário ou ao sistema (TANENBAUM, 2003).

2.4.2 Gerenciamento de Memória

Todo computador tem uma memória principal que é usada para guardar os programas em execução. Sistemas operacionais simples permitem que apenas um programa por vez ocupe a memória principal.

Sistemas operacionais mais sofisticados permitem que múltiplos programas habitem a memória ao mesmo tempo, mas devem oferecer um mecanismo de proteção para que programas não relacionados não possam acessar, e principalmente modificar, informações de outros programas na memória.

Outro fator importante é o gerenciamento do espaço de endereçamento dos processos. Em computadores antigos, programas que eram maiores que a memória disponível do computador poderiam causar uma série de complicações. Em computadores mais modernos, que utilizam a técnica da memória virtual, quando isso acontece, parte do espaço de endereçamento do programa é guardado no disco, retirando e colocando as informações conforme o necessário (com um alto custo em velocidade, mas evitando complicações) (TANENBAUM, 2003).

2.4.3 Entrada e Saída

Todos os computadores possuem dispositivos de entrada e saída de dados, pois sem eles, os usuários não poderiam dizer o que deve ser feito e/ou não poderiam verificar o resultado obtido. Existem vários tipos de dispositivos de entrada e saída, como *mouses*, teclados, monitores, caixas de som, impressoras, *scanners* etc. Cabe ao sistema operacional gerenciar todos esses dispositivos.

Consequentemente, todo sistema operacional possui um subsistema de entrada e saída para gerenciar seus dispositivos. Alguns dos programas de entrada e saída são independentes de dispositivos (aplicam-se igualmente bem a muitos ou a todos os dispositivos). Outros programas, ou parte deles, como os *drivers*, são específicos para cada dispositivo de entrada e saída (TANENBAUM, 2003).

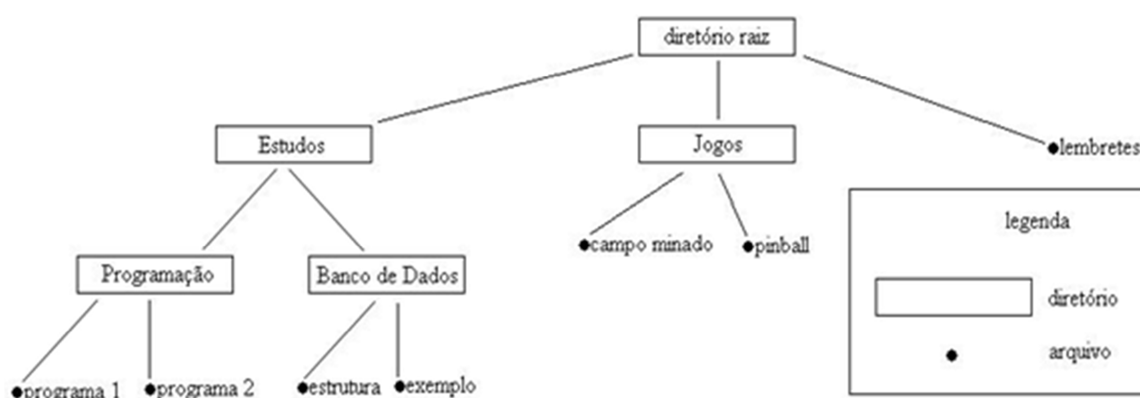
2.4.4 Arquivos

Arquivos e sistemas de arquivos constituem outro conceito fundamental que compõem praticamente todos os sistemas operacionais. Uma das funções de um sistema operacional é ocultar e simplificar para o usuário as peculiaridades do *hardware*, o que inclui os dispositivos de armazenamento.

Chamadas ao sistema são necessárias para criar, remover, ler e escrever arquivos. Para se ler um arquivo, ele deve ser primeiro localizado, aberto e, depois de lido, fechado. Antes de se realizar uma operação com um arquivo ou diretório, ele precisa ser aberto, e nesse momento, as permissões do usuário são verificadas.

Para organizar e guardar os arquivos, a maioria dos sistemas operacionais oferece a tecnologia de diretórios como um modo de agrupar os arquivos. Também são necessárias chamadas ao sistema para criar, renomear e remover diretórios, além de chamadas para colocar e remover arquivos de um diretório. Um diretório pode ter entradas de arquivos ou outros diretórios, o que nos permite representar essa hierarquia em uma estrutura de árvores como no exemplo da Figura 4.

Figura 4 – Hierarquia de Diretórios



Fonte: baseado em TANENBAUM, 2003

Cada arquivo ou diretório pode ser especificado fornecendo-se o caminho a partir do topo da hierarquia (diretório raiz). Isso forma uma lista dos diretórios que devem ser percorridos a partir do diretório raiz para se chegar ao arquivo. Cada processo tem um diretório de trabalho atual, ou seja, suas buscas e/ou especificações de arquivos iniciam desse diretório de trabalho.

Há também arquivos chamados de *pipes*, que são arquivos que servem de comunicação entre dois ou mais processos (um processo escreve no arquivo, e outro processo lê) (TANENBAUM, 2003).

2.4.5 Segurança

Computadores contêm informações que seus usuários, muitas vezes, querem manter confidenciais, e cabe ao sistema operacional gerenciar o sistema de segurança, para que os arquivos e recursos sejam acessíveis apenas a usuários autorizados e para que o computador não seja invadido por usuários externos ou vírus (TANENBAUM, 2003).

2.4.6 Interpretador de Comandos

O sistema operacional serve como código que executa as chamadas ao sistema, assim, editores, compiladores, montadores, ligadores e interpretadores de comandos normalmente não fazem parte do sistema operacional, mesmo que sejam importantes e úteis.

Embora não seja parte de um sistema operacional, um interpretador de comandos utiliza extensivamente os aspectos do sistema operacional, além de poder ser a interface principal entre o usuário e o sistema operacional (a outra possibilidade é uma interface gráfica de usuário) (TANENBAUM, 2003).

3 SISTEMAS EMBARCADOS

Houve muito interesse em tecnologia da informação na última década, mas pouca pesquisa tem sido concentrada em reengenharia de produtos físicos, responsável por incorporar sensores simples em microprocessadores embarcados complexos e sistemas de *software* (tecnologia embarcada da informação) em produtos. Porém, com a competitividade em muitos setores, tem havido um crescente foco em tecnologia embarcada da informação (KONANA, 2007).

Este capítulo aborda diversos aspectos sobre os sistemas embarcados, suas características, seus componentes, sua utilização em projetos, assim como características de sistemas operacionais embarcados usados atualmente no mercado.

3.1 Características

Um sistema embarcado é um sistema computacional fisicamente limitado, geralmente com restrições de memória, tamanho, energia e, consecutivamente, potência, que possui um número limitado, mas específico, de funções. Geralmente está associado ou mesmo embutido em outro produto, como um eletrodoméstico ou um veículo. Muitas vezes possuem características de sistemas de tempo real, como alta velocidade de envio, tratamento e recebimento de dados (TANENBAUM, 2003).

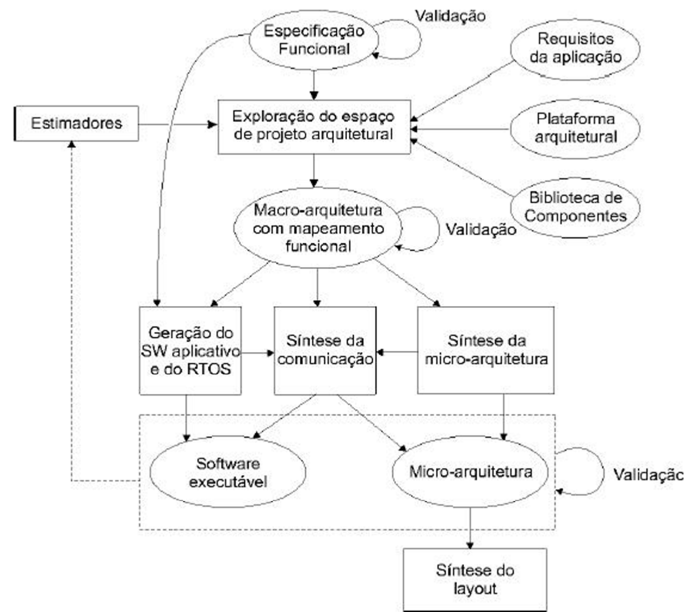
Outras restrições importantes são o custo e o baixo tempo de projeto. Porém, mesmo com todas essas restrições, o desempenho final não pode ser comprometido. Devido aos baixos custos tecnológicos atuais eles podem ser encontrados em produtos das atividades cotidianas, e a tendência é que cada vez mais, eles entrem no cotidiano das pessoas.

O projeto desse tipo de sistema é extremamente complexo por envolver conceitos pouco analisados pela computação de propósitos gerais. Ao projetar um sistema ou programa embarcado, as principais preocupações são, devido aos limites físicos, com portabilidade, baixo limite de potencia, baixa disponibilidade de energia, baixa disponibilidade de memória, necessidade de segurança, confiabilidade, a possibilidade de funcionamento em uma rede maior e o curto espaço de tempo de projeto.

Um projeto de *software* embarcado pode ser muito caro caso ele tenha grande complexidade, pois pode envolver equipes multidisciplinares (como *hardware* digital, *hardware* analógico, *software* e teste) e utilização de ferramentas computacionais de custo elevado. São ainda mais caros quando se tratam de sistemas integrados em uma pastilha, o que obriga empresas a aceitarem projetos que tenham garantidamente volume muito alto de produção. É importante notar que grandes projetos necessitam de uma modelagem melhor definida, com vários níveis de abstração e ferramentas de automação (CARRO, 2003).

Um exemplo de metodologia de um grande projeto de um sistema embarcado pode ser encontrado na Figura 5:

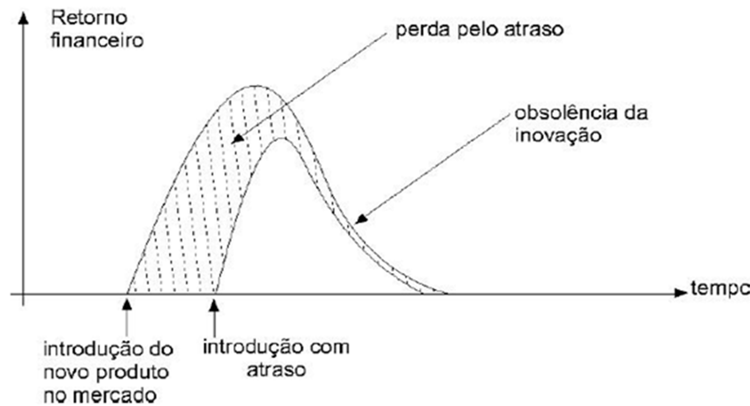
Figura 5 – Metodologia de Projeto



Fonte: CARRO, 2003

Um atraso em um projeto de *software* embarcado pode causar perda considerável de lucros ou pode até causar prejuízos (CARRO, 2003), como exemplificado na Figura 6.

Figura 6 – Retorno Financeiro e Janelas de Tempo



Fonte: CARRO, 2003

O uso de tecnologia embarcada em produtos físicos não é nova. Como exemplo, na década de 80, computadores de controle numéricos obtiveram aumentos drásticos em produtividade e qualidade de processos.

Produtos que usam tecnologia embarcada podem identificar e corrigir problemas antes que ocorra uma falha, capturando e processando eventos (geralmente através de sensores). A tecnologia embarcada torna fácil operar, diagnosticar, servir e personalizar produtos, além de oferecer informações críticas aos fabricantes, o que os permitem oferecer novos serviços, entrar em processo de inovação e construir relações diretas com os clientes. Assim, a tecnologia embarcada tem implicações importantes para o escopo de empresas, competição industrial e aplicações em negócios.

Ela também permite a produtores e prestadores de serviço mapear a saúde de um produto através de seu ciclo de vida, capturando o valor de serviço durante seu ciclo de vida e integrando operações físicas com processos de negócios. Esse mapeamento fornece informações importantes (como problemas de qualidade e estado de manutenção), o que permite melhorar futuros modelos, além de permitir os produtores a prover serviços proativamente, fortalecendo a relação com os clientes.

Esse mapeamento pode ainda servir como guia para decisões de estoque de produtos e partes, especialmente quando há grande quantidade de modelos de um mesmo produto em uma mesma região.

Assim, empresas deveriam ter razões estratégicas para usar tecnologia embarcada, já que ela permite oportunidades para inovação de produtos e processos. Há três tipos de iniciativas para incorporar tecnologia embarcada em um produto:

- Enriquecimento: produtos são redesenhados com tecnologia embarcada para melhorar qualidade, conveniência e desempenho, mas a funcionalidade básica do produto não é alterada e funcionaria mesmo se os sistemas embarcados fossem removidos. Desse modo, o objetivo primário é oferecer novas funções para diferentes produtos. Exemplo: sapatos;
- Digitalização: partes físicas e já existentes do produto são modificadas e ajustadas para prover resultados digitais, mas há partes do produto que ainda são operadas manualmente. Esse tipo de incorporação permite a empresa a oferecer outros produtos e serviços. Exemplo: monitoramento cardíaco;
- Substituição: o produto é totalmente redesenhado e substituído por tecnologia digital. Exemplo: máquinas fotográficas.

Apesar de todas essas vantagens, produtos embarcados também trazem desafios, já que a tecnologia embarcada difere dos modos tradicionais de produtos. Entre esses desafios, temos:

- Projeto modular: produtos tradicionais ainda não seguem o projeto modular, optando pelos modelos mais tradicionais. Para se aplicar tecnologia embarcada em um produto tradicional, o modo como o produto é projetado necessita ser atualizado;
- Especificação: as interfaces com o produto físico devem ser claramente especificadas;
- Capacidades organizacionais: com a tecnologia embarcada e o projeto modular de produtos, a frequência de inovações irá aumentar, o que por sua vez torna os produtos obsoletos mais rapidamente, diminuindo sua vida útil e seu ciclo de vida;
- Conflitos com serviços existentes: poderão haver conflitos com serviços criados quando o produto não era embarcado, além de conflito entre serviços que a empresa necessitara oferecer que antes eram oferecidos por outras empresas;
- Treinamento e desenvolvimento da força de trabalho: desenvolvedores de produtos necessitarão ter conhecimentos de tecnologia da informação para reconhecer oportunidades, e os funcionários que prestam serviços aos clientes terão que ter uma educação básica sobre sistemas embarcados (KONANA, 2007).

3.2 Produtos que usam Tecnologia Embarcada

Empresas têm usado tecnologia embarcada em produtos tradicionais para adicionar novas funcionalidades para melhorar a conveniência, desempenho e segurança dos produtos, incluindo veículos e aparelhos eletrônicos em geral. Como exemplo de produtos, podemos citar aviões, carros, geladeiras, máquinas fotográficas, televisores, telefones, celulares e computadores portáteis entre diversos outros.

3.3 Componentes de um Sistema Embarcado

Projeto de sistemas eletrônicos embarcados enfrentam diversos desafios porque seu espaço de projeto arquitetural a ser explorado é demasiado vasto. A arquitetura de *hardware* pode conter um ou mais processadores ou microcontroladores (tipo de processador com recursos extras como memória e maior suporte a periféricos), memória, interfaces para periféricos e blocos dedicados. Somente a estrutura de comunicação pode variar de um barramento a uma rede complexa.

Além disso, os processadores e microcontroladores disponíveis possuem diversos modelos, os quais devem ser considerados conforme sua aplicação, como *RISC* (Conjunto Reduzido de Construções de Computação do inglês *Reduced Instruction Set Computer*)*, VLIW* (Palavra de Instrução Bem Longa do inglês *Very Long Instruction Word*), *DSP* (Processador de Sinal Digital do inglês *Digital Signal Processor*) e *ASIP* (Processor de Conjunto de Instruções para Aplicações Específicas do inglês *Application-Specific Instruction-Set Processor*), entre outros.

Caso o sistema possua componentes programáveis, o *software* de aplicação pode ser constituído por múltiplos processos, distribuídos entre diferentes processadores e comunicando-se por meio de mecanismos variados.

Um sistema operacional de tempo real também pode ser necessário dependendo da aplicação ou do projeto, pois oferece serviços como comunicação e escalonamento de processos. Em muitas aplicações, é mais adequada a integração do sistema em *SoC* (Sistema em uma Pastilha do inglês *System-on-a-chip*), que é um projeto com custo geralmente mais elevado do que o normal (CARRO, 2003).

Nas situações de criticidade de requisitos de área, potência e/ou desempenho, o projeto em uma única pastilha, na forma de *ASIC* (Circuito Integrado para Aplicação Específica do inglês *Application-Specific Integrated System*), pode ser mandatório. Porém em situações contrárias, é mais indicada a implementação

de um sistema *FPGA* (Arranjo de Portas Programável em Campo do inglês *Field-Programmable Gate Array*), alternativa de customização mais econômica para baixos custos, ou mesmo usar um sistema baseado em famílias de microprocessadores.

Para acelerar o andamento dos projetos, as empresas têm adotado o paradigma de plataformas, que é uma arquitetura de *hardware* e de *software* específica para um domínio de aplicações, porém altamente parametrizável. Essa estratégia viabiliza a reutilização de componentes previamente desenvolvidos e testados, reduzindo o tempo de projeto. Esse reuso pode ser ainda mais reforçado adotando-se padrões para a arquitetura e desenvolvimento (CARRO, 2003).

O projeto de um sistema embarcado consiste então em encontrar um derivativo da plataforma que atenda aos requisitos da aplicação. Partindo-se de uma especificação de alto nível da aplicação, faz-se uma exploração das soluções arquiteturais possíveis, estimando o impacto dos particionamentos de funções. Após isso, é necessária a síntese da estrutura de comunicação, que integrará os componentes e o *hardware*. Também é vital implementar uma metodologia de testes para o sistema.

Nesse modelo de projeto, as inovações dependem mais do *software* do que do *hardware*, já que o último está automatizado. Como a automação do projeto de *hardware* já foi alcançado pelas empresas, o mercado no momento visa o mesmo para o projeto de *software*.

Aprofundando nos componentes de um sistema embarcado, o uso de memória deve ser calculado, pois memórias rápidas ou grandes consomem mais energia, porém não permitem expansões, então sua capacidade de reuso e o tamanho também são importantes, especialmente se houver atualizações planejadas. Há duas soluções para o reuso: descarte dos dados usados, como em um computador comum, e um *buffer* circular, que volta para o começo da memória quando o final encher (só é necessário tomar cuidado para a memória não preencher instruções ainda não resolvidas. Quando um *buffer* ultrapassa seu limite e perde dados, chamamos este evento de *buffer overflow*).

Como as memórias são mais lentas do que os processadores, podemos ter problemas de desempenho caso não sejam devidamente planejadas. Felizmente, o modelo de hierarquia utilizado em sistemas mais comuns funciona bem para sistemas embarcados.

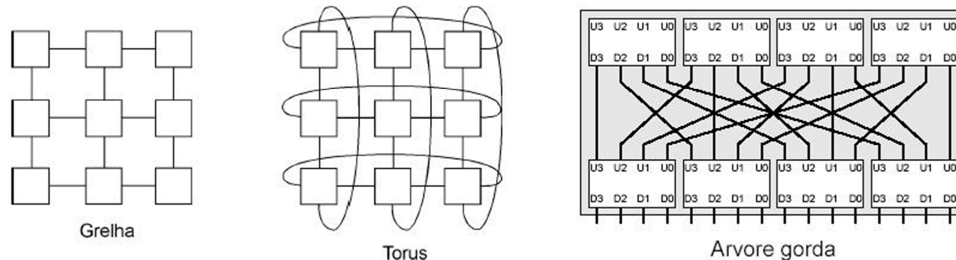
Para adquirir desempenho, o paralelismo e o pseudoparalelismo (uso de *pipeline*s) podem ser necessários. Não é incomum encontrar em grandes projetos, processadores ou microcontroladores com múltiplos núcleos ou mesmo mais de um processador, mas deve se tomar cuidado com a dependência de dados (instruções que dependem do resultado de outra), e a ordem correta de instruções.

O último problema da arquitetura é a comunicação entre componentes. Ligações diretas (ponto-a-ponto) são as mais rápidas, porém devem ser refeitas para cada projeto, o que obviamente recai no tempo de projeto. Barramentos são reusáveis, mas permitem apenas uma transação por vez dentro de seu domínio, o que causa uma enorme perda de desempenho quando o sistema é grande. Uma hierarquia de barramentos também é uma opção, onde há vários sub-barramentos espalhados, mas essa opção ainda pode paralisar diversos recursos de comunicação. Assim, ainda não há um tipo de comunicação ideal.

Alguns trabalhos recentes propõem o uso de uma rede em um chip (*NoC*), que é um conjunto de canais e roteadores dedicados à comunicação. Como as conexões entre roteadores são ponto-a-ponto, a rede pode transmitir mais mensagens ao mesmo tempo. Ela permite paralelismo e é facilmente escalável (basta colocar mais roteadores). A eficiência de uma *NoC* porém depende de vários fatores, como o tamanho do canal de roteamento (quantidade de fios que fazem parte do mesmo), a política de prioridade de mensagens, a topologia da própria *NoC* e a estratégia de chaveamento. Além disso, a distância de certos componentes do *NoC* pode ser crítica, como a memória e o processador. Por último, é importante enfatizar que o custo de um roteador é alto, e uma *NoC* possui vários roteadores.

Na Figura 7, é exemplificado três modelos de *NoC*s (CARRO, 2003).

Figura 7 – Modelos de *NoC*s



Fonte: CARRO, 2003

3.4 Requisitos de Sistemas Embarcados

A maioria dos projetos de sistemas embarcados possuem requisitos bem definidos. Entre os requisitos mais comuns, temos:

1. Computação de Recursos Restritos: como os recursos em um sistema embarcado são escassos, é necessário usá-los eficientemente;

2. Requisitos de Tempo Real: muitas aplicações embarcadas interagem profundamente com o mundo real e por isso têm requisitos estritos de tempo;

3. Portabilidade: para garantir custos menores, os componentes devem ter a possibilidade de serem portados para outras plataformas;

4. Alta Confiabilidade: por serem usados em aplicações críticas, falhas de *software* ou *hardware* são muito problemáticas e extremamente caras;

5. Estabilidade, robustez e confiança: incluem aspectos de confiabilidade e disponibilidade, ou seja, a capacidade de continuar trabalhando sem falhas ou com tolerância ou isolamento delas;

6. Controle de falhas: em tecnologia embarcada, especialmente em redes, podem ocorrer falhas por vários motivos. De um modo geral, as falhas não devem impactar nas funções gerais do sistema, então as falhas devem ser toleradas, controladas, isoladas ou registradas;

7. Proteção: perda de vida, danos severos as pessoas, propriedades e ambiente não devem acontecer. Esse tipo de requisito é para todo o sistema embarcado em que isso pode acontecer, o que inclui equipamentos médicos, veículos, equipamentos industriais e equipamentos militares;

8. Segurança: é a capacidade do sistema de prevenir informações e recursos de serem acessados por usuário não autorizados;

9. Privacidade: usuários geralmente evitam revelar informações confidenciais. Esse requisito se preocupa em não revelar essas informações;

10. Escalabilidade: a capacidade do sistema de ser prontamente expandido ou de conseguir gerenciar quantidades crescentes de trabalho sem que haja muito impacto;

11. Melhoras: tem haver com a capacidade de melhoramentos do sistema, seja em novas funcionalidades ou melhoramento das antigas.

Manter controle sobre todos estes requisitos não é tarefa fácil, especialmente em projetos de grande porte e de orçamento limitado.

3.5 Sistemas Operacionais Embarcados

No caso de sistemas embarcados que contêm componentes programáveis, o *software* de aplicação pode ser composto por múltiplos processos, distribuídos entre diferentes processadores e comunicando-se através de mecanismos variados. Um sistema operacional, oferecendo serviços como comunicação e escalonamento de processos, pode ser necessário (CARRO, 2003).

Um sistema operacional embarcado tais recursos para um sistema embarcado facilita o desenvolvimento de aplicações, mas idealmente deve impactar pouco nas restrições do sistema embarcado (CARRO, 2003).

3.5.1 Características

As características de um sistema operacional embarcado são muito parecidas com as de sistemas embarcados, já que um sistema operacional embarcado é uma extensão do sistema a qual pertence, e não deve interferir em suas funcionalidades básicas, assim, ele deve possuir alto desempenho, baixo custo, confiabilidade, segurança, privacidade, escalabilidade e pouco consumo de energia, memória, processador e espaço. Deve também ajudar os programas que habitam o sistema a desempenhar suas funções, além de tratar das falhas de *software* e *hardware*. Os sistemas operacionais embarcados também devem ser flexíveis e configuráveis. Eles servem ainda para escalonar (organizar) as tarefas, gerenciar a memória e auxiliar nas comunicações.

Diferente de outros sistemas operacionais, os sistemas operacionais embarcados possuem apenas o modo núcleo, e chamadas ao sistema não são necessários, pois todos os programas são executados em modo núcleo e possuem acesso total ao *hardware* (isso poderia ser um problema de segurança, mas gerenciar isso causaria grande impacto no desempenho do sistema). Eles podem fornecer uma interface simplificada ao usuário, mas sua principal função é como extensor e gerenciador do *hardware*

Além disto, um sistema embarcado que possui um sistema operacional obviamente precisa de mais memória principal, maior potencia de processamento e mais energia elétrica. Isso faz com que os custos de *hardware* sejam maiores, mas também pode ajudar a encurtar consideravelmente o tempo despendido no projeto. Por isto, estes sistemas operacionais são geralmente usados em projetos com maior complexidade (CARRO, 2003).

3.5.2 Sistemas Atualmente Disponíveis

Sistemas operacionais embarcados são diversos, e podem ser *GPOSs* (Sistemas Operacionais de Propósito Geral do inglês *General Purpose Operanting Systems*) ou específicos. A maioria dos sistemas embarcados disponíveis no mercado possuem propriedades de sistemas de tempo real, como garantia de resposta e alta velocidade, além de serem flexíveis e configuráveis.

Alguns dos sistemas operacionais embarcados mais tradicionais, como o *VxWorks*, *WinCE* (Sistema Compacto Embarcado Windows do inglês *Windows Embedded Compact*), *QNX*, *PalmOS*, *OS-9*, *LynxOS* e *Symbian* são tipicamente grandes (centenas de *KB*s de memória) sistemas de propósitos gerais, com um rico conjunto de interfaces de programação. São voltados à projetos com mais recursos de memória e processamento, e geralmente suportam multitarefas, proteção de memória, conexões de rede, *API*s (Interfaces de Programação de Aplicações do inglês *Application Programming Interface*) e proteção contra falhas.

O *VxWorks*, da *Wind River*, é o sistema operacional embarcado de propósito geral mais adotado (a Estação Espacial Internacional usa ele). Ele foi criado no começo da década de 80 e fornece uma *API* com 1800 métodos, incluindo suporte a rede, hierarquia de arquivos e gerenciamento de entrada e saída de dispositivos e dados. Ele também traz a bancada de trabalho da *Wind River*, que é uma coleção de ferramentas para acelerar o desenvolvimento com o *VxWorks*. O servidor de desenvolvimento pode ser *Red Hat Linux*, *Solaris*, *SuSE Linux*, *Windows 2000 Professional* ou *Windows XP*. Ele provê também um ambiente de desenvolvimento visual. Ele suporta agendamento, com 256 níveis de prioridades.

O *QNX* é baseado na ideia de executar sistemas operacionais como pequenas tarefas, conhecidas como *servers*. Isso difere dos sistemas operacionais tradicionais, nos quais o sistema é apenas um grande programa composto de várias partes com habilidades especiais. *QNX Neutrino* (2001) foi portado para várias plataformas e funciona em praticamente todos os processadores modernos usados no mercado embarcado. O *QNX* também suporta agendamento, com 256 níveis de prioridades.

O *Windows CE* é um sistema operacional embarcado desenvolvido no final da década de 90 pela Microsoft. Ele é modular, portável e é de tempo real, especialmente projetado para aparelhos com memórias pequenas. Há três principais plataformas de desenvolvimento (*Windows Móbile*, *SmartPhone* e *Portable Media Center*), que permitem o uso de ferramentas ricas em recursos. *Windows CE* suporta até 32 processadores ativos, com múltiplas linhas de execução em cada processo. Também suporta agendamento com 256 níveis de prioridades e possui primitivas de sincronização, semáforos e eventos.

Outros sistemas operacionais embarcados escolheram uma orientação por componentes para configuração de aplicações específicas, como o *eCo* (Sistema operacional Embarcado Configurado do inglês *Embedded Configurable Operating System*) e o *icWORKSHOP*, que têm como objetivo composições leves e estáticas. Eles consistem em um conjunto de componentes que são ligados para formar a aplicação.

VEST é um conjunto de ferramentas propostas para construir sistemas operacionais embarcados baseados em componentes que realizam analises estáticas extensivas, como agendamento, dependências de recursos e interfaces de correção de gramática.

O *eCos* é o sistema operacional embarcado livre e de código aberto mais adotado. Lançado em 1986, ele provê uma ferramenta de configuração gráfica e uma ferramenta de configuração por linhas, para assim adaptar o sistema operacional para realizar requerimentos de aplicações específicas. Este recurso permite ao usuário configurar o sistema operacional para requerimentos específicos de memória e desempenho. O servidor de desenvolvimento pode ser *Windows* ou *Linux*, e ele suporta vários tipos de processadores. Seu agendamento permite até 32 níveis de prioridade. O *eCos* também possui primitivas de sincronização, semáforos, eventos e caixas de correio eletrônico.

Há sistemas operacionais que são especialmente desenvolvidos para sistemas embarcados menores, como *CREEM*, *pSOSytem*, *OSEKWorks* e *Ariel*. Eles têm severas restrições de execução e modelos de armazenamento.

Alguns sistemas operacionais contemporâneos como o Linux possuem extensões que os permitem suportar aplicações de tempo real (como o *RTAI*, o *RT-Linux* e o *uClinux*), mas são adequados apenas para grandes sistemas de tempo real por causa da sua arquitetura. Ainda assim há preocupações com segurança e privacidade. Com algumas soluções propostas, foram criados o *L4 OS* e o *Minix OS*.

Na tabela 2, são apresentados quão bem alguns dos sistemas operacionais embarcados mencionados se adéquam ou fornecem os requisitos estudados na seção 3.4 (FRIEDRICH, 2009).

Tabela 2 – Requerimentos de Sistema Operacionais Embarcados

SO	1	2	3	4	5	6	7	8	9	10	11
VxWorks	Parcial	Parcial	Adequado	Adequado	Parcial	Parcial	Parcial	Adequado	Parcial	Adequado	Parcial
QNX	Parcial	Parcial	Adequado	Adequado	Parcial	Parcial	Parcial	Adequado	Parcial	Adequado	Parcial
WinCE	Parcial	Parcial	Adequado	Parcial	Inadequado	Inadequado	Inadequado	Parcial	Parcial	Parcial	Inadequado
eCos	Adequado	Parcial	Adequado	Parcial	Inadequado	Inadequado	Parcial	Parcial	Parcial	Adequado	Inadequado
pSOSystem	Adequado	Parcial	Parcial	Parcial	Inadequado	Inadequado	Parcial	Parcial	Inadequado	Parcial	Inadequado
RTAI	Inadequado	Adequado	Parcial	Parcial	Inadequado	Inadequado	Inadequado	Parcial	Inadequado	Parcial	Inadequado
uClinux	Adequado	Inadequado	Adequado	Parcial	Inadequado	Inadequado	Inadequado	Parcial	Inadequado	Parcial	Inadequado
L4	Parcial	Parcial	Parcial	Adequado	Adequado	Adequado	Parcial	Adequado	Adequado	Adequado	Parcial
Minix	Parcial	Inadequado	Parcial	Adequado	Adequado	Adequado	Parcial	Adequado	Adequado	Adequado	Parcial

Fonte: FRIEDRICH, 2009

4 FERRAMENTAS DE DESENVOLVIMENTO

Há uma grande gama de ferramentas de desenvolvimento de sistemas embarcados, como compiladores, simuladores de hardware, *design* de hardware e sistemas operacionais. Algumas dessas ferramentas foram estudadas, as quais estão descritas nas seções a seguir:

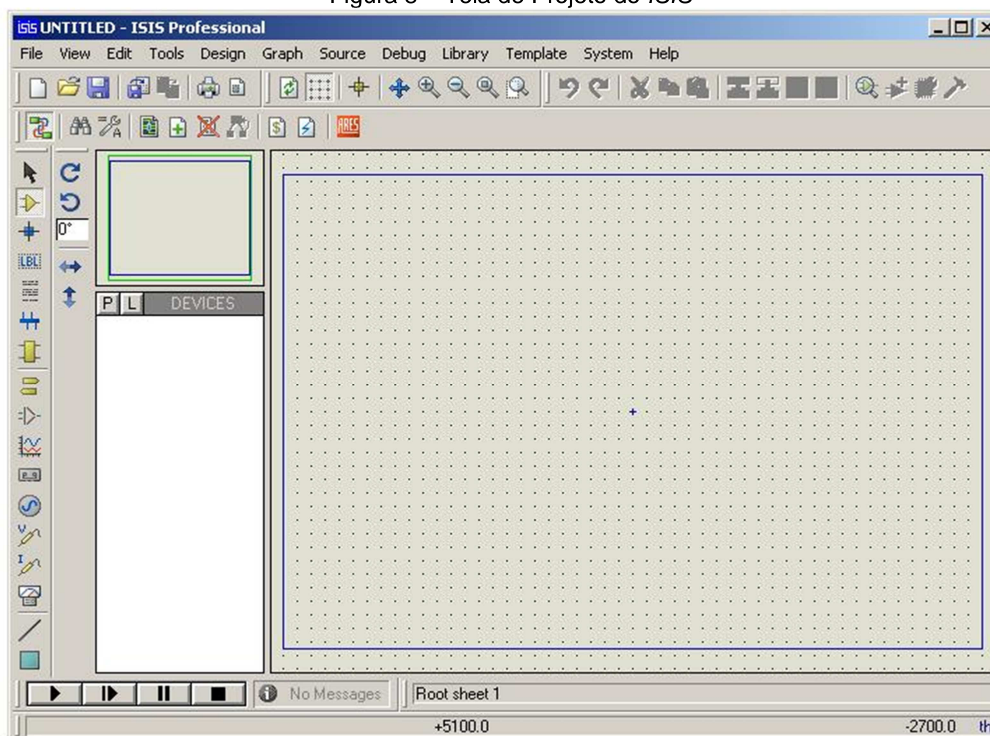
4.1 *Proteus*

Um conjunto de ferramentas para *design*, simulação e esquematização de sistemas embarcados, assim como de componentes para sistemas embarcados, desenvolvida pela *Labcenter Eletronics*. Apesar de ser um conjunto de ferramentas, o módulo mais utilizado é o *ISIS*, que permite o *design* e simulação de sistemas embarcados, além de permitir alteração de componentes caso esses não sirvam especificamente para o sistema em mente.

4.1.1 *ISIS*

O ISIS permite, de forma simples, montar e simular os componentes de hardware de sistemas embarcados e definir atributos para seus componentes. Um exemplo da tela de projeto do *Isis* pode ser encontrada na Figura 8.

Figura 8 – Tela de Projeto do *ISIS*



Os componentes necessitam ser adicionados ao projeto e podem ser ligados através de circuitos, porém cada componente possui atributos específicos, e por isso não podem ser descritos de forma generalizada.

4.1.1.1 Energia

Além de fornecer pilhas, baterias, motores e outras formas de energia, o *ISIS* fornece também força e aterramento elétricos padrões, denominados apropriadamente de *power* e *ground*. Por serem frequentemente usados e por não serem dispositivos propriamente ditos, estes se localizam separados dos componentes do projeto, e sempre estão disponíveis. Há ainda *inputs* e *outputs*, que servem para transferência direta de sinal quando possuem o mesmo nome.

4.1.1.2 Resistores

Os resistores são extremamente importantes para qualquer projeto, pois impedem que dispositivos físicos sejam queimados ou danificados por uma corrente elétrica de tensão muito alta. Isso é evidenciado no *ISIS*, pois se uma tensão muito forte ou muito fraca passa por um componente, especialmente se o componente em questão for um microcontrolador, esse componente não funcionará devidamente. É possível configurar a resistência de cada resistor.

4.1.1.3 Botões

Os botões servem de base para passar sinais através de um circuito apenas quando pressionados. Podem ser usados junto a um microcontrolador que detecte alterações em uma corrente para criar efeitos mais duradores, como desligar certos aparelhos sem desligar o sistema inteiro, ou podem ser usados para iniciar reações que um usuário queira que aconteça apenas quando os botões estejam pressionados.

4.1.1.4 Interruptores

Também conhecidos como *switch*s, os interruptores são parecidos com os botões, porém, por ser um componente que muda de estado após pressionado, o sinal torna-se contínuo, e o interruptor necessita ser pressionado novamente para que volte ao estado anterior e interrompa o fornecimento do sinal.

Por fornecer sinal continuamente, é geralmente usado diretamente com lâmpadas, mas pode ser usado também com microcontroladores que estejam configurados para detectar o estado de um sinal.

4.1.1.5 Portas Lógicas

Um componente emissor de sinal digital, que pode ser facilmente ligado e desligado como um interruptor. Como o sinal já está em formato digital, isso facilita o uso de componentes que possuem pinos ou *ports* que lêem informações digitais, como microcontroladores.

4.1.1.6 Lâmpadas e *LEDs*

Lâmpadas e *LED*s (Diodo Emissor de Luz do inglês *Light Emmiting Diode*) são componentes que emitem luz quando uma corrente elétrica suficientemente forte passa por eles. Geralmente usados para iluminação ou sinalização de algum evento.

4.1.1.7 *Displays*

*Display*s são monitores que mostram algo quando sinais passam por eles. Um *display* de segmentos necessita apenas discernir tensão alta ou baixa para ligar seus segmentos, enquanto um *display* de *LCD* (Display de Cristal Líquido do inglês *Liquid Crystal Display*) possui leitores lógicos para receber a mensagem que deve ser mostrada.

4.1.1.8 Sensores de temperatura

Os sensores de temperatura são dispositivos que lêem a temperatura de determinado objeto e enviam um sinal analógico com as informações para um circuito. Esse sinal analógico necessita ser interpretado, função a qual um microcontrolador é bem útil.

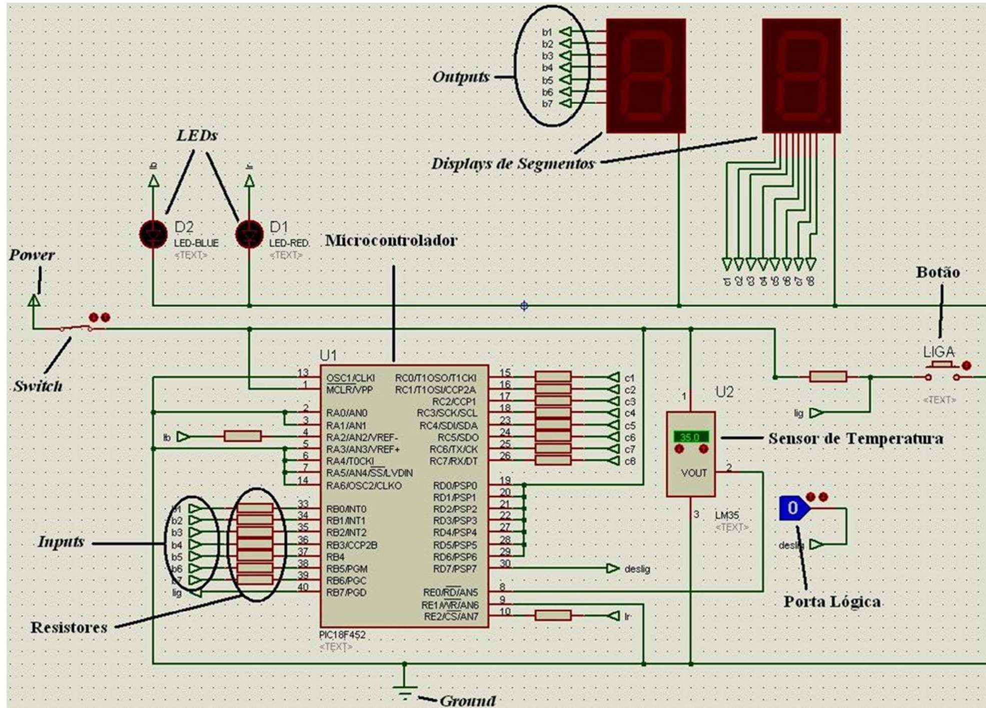
4.1.1.9 Microcontroladores

Microcontroladores são componentes complexos, com dezenas de pinos, cada pino com uma configuração própria dependendo do fabricante do microcontrolador. Eles possuem pinos e *ports* dos tipos digitais e analógicas que podem ser usadas tanto para receber quanto para enviar sinais.

Um de seus parâmetros, para efeito de simulação no *ISIS*, é o caminho do arquivo hexadecimal do programa que o microcontrolador deverá executar. É recomendável que o programa possua *loop* infinito e que o microcontrolador possua energia quase constantemente para que o programa não pare. Exceções podem ser necessárias dependendo da finalidade do sistema e também para poupar energia do sistema, mas estes devem ser avaliados cuidadosamente.

A Figura 9 representa um projeto no ISIS com alguns dos componentes citados.

Figura 9 – Exemplos de Dispositivos no *Isis*

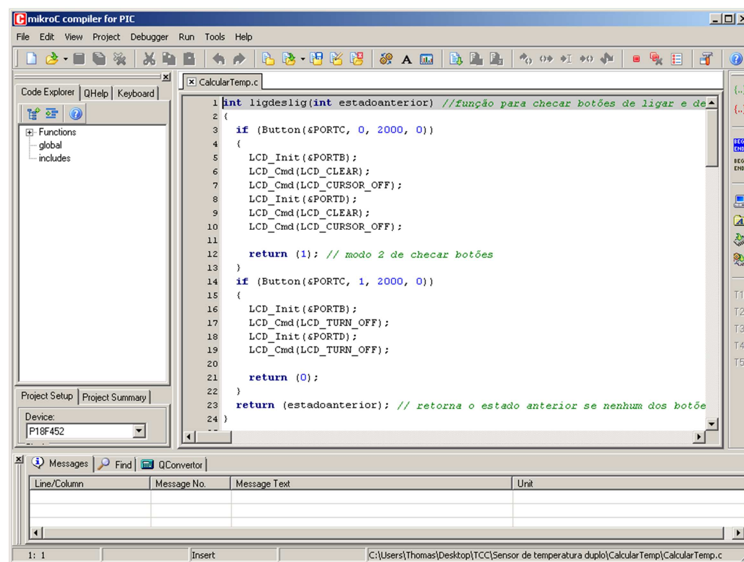


4.2 *MikroC*

Desenvolvido pela *MikroEletronika*, o *mikroC* é um compilador de linguagem C para microcontroladores da família *PIC* (uma série de microcontroladores projetados pela *Microchip Technology*). O MikroC foi desenvolvido visando fornecer uma interface simplificada sem comprometer desempenho e controle das aplicações. Ele possui suporte a bibliotecas de C e também de compilação para arquivos HEX, que podem ser usados pelos microcontroladores do Proteus.

Para que se possa compilar algo, é necessário criar um projeto e definir qual o microcontrolador usado e definir sua freqüência e suas variáveis de sinalização (*flags*). Já para que uma aplicação funcione adequadamente, é necessário iniciar os *ports* do microcontrolador que serão usadas para entrada e saída de dados (MIKROELETRONIKA, 2006). A Figura 10 exemplifica uma tela de projeto do *MikroC*.

Figura 10 – Tela de Projeto do *MikroC*



4.3 *FreeRTOS*

O *FreeRTOS* é um sistema operacional embarcado de código aberto que é ideal para tarefas de tempo real exigentes que usam microcontroladores pequenos ou médios (que possuem entre 16 KB e 256 KB de memória *RAM*). Ele é escrito majoritariamente em C, e permite que aplicações sejam organizadas em coleções independentes de *threads* e permite a categorização de threads por prioridade, facilitando a programação concorrente.

Ele dá suporte a muitas famílias de microcontroladores e é suportado por vários compiladores. Isso é possível porque uma de suas formas de uso é através de código *source* (não compilado) e pode ser integrado facilmente na maioria dos compiladores usando seus arquivos como bibliotecas, através do comando *include*. Isso permite uso de partes específicas do *FreeRTOS*, economizando poder de processamento. Permite também modificação dos arquivos do *FreeRTOS* para atender as necessidades de um sistema embarcado específico

As funções do *FreeRTOS* podem ser classificadas por seu tipo de retorno, já que possuem um prefixo de uma ou mais letras que especificam o tipo de retorno da função, como o prefixo “v”, que significa “*void*” e que significa que a função não retorna nada para a função que a invocou.

Para se aproveitar da maioria das características e até fazer uso de várias funções do *FreeRTOS*, é necessário o uso de tarefas, também conhecidas como *tasks,* uma das funções fundamentais para o entendimento e uso do *FreeRTOS* (BARRY, 2010).

4.3.1 Tarefas

No *FreeRTOS*, as tarefas ou *tasks* são implementadas como funções e cada uma é basicamente um pequeno programa de *loop* (execução) infinito e sem saída. Elas não devem não devem ter um retorno (*return*) e não devem executar além do fim da função. Se uma *task* não for mais necessária, deve ser prontamente excluída. Uma *task*, por outro lado, pode ser usada para criar um número indefinido de outras, desde que tenham nomes diferentes

Cada *task*, seja ela criada diretamente ou seja ela criada por outra, possui sua própria pilha e sua própria cópia de variáveis definidas. Outra característica das *tasks* é que elas possuem dois estados: executando (quando um dos núcleos está executando seu código) e não executando (quando está pronta, porém esperando, para ser executada). O modo de troca de *tasks* no núcleo é feito de modo que as informações e variáveis são salvas quando uma *task* é interrompida e carregada quando chega sua vez de execução. No *FreeRTOS*, para que as *tasks* entrem no processo de execução e espera no microcontrolador, é necessário usar a entidade *scheduler*, uma entidade responsável por gerenciar esse processo.

Tasks são criadas usando a função de *API* “FreeRTOSxTaskCreate()”. Essa função é provavelmente a mais complexa de todas as funções de API do *FreeRTOS*, mas também é o componente fundamental de todos para sistemas de multitarefas que usam o *FreeRTOS*, e por isso é o primeiro a ser estudado no manual de referências. Seus parâmetros são os seguintes:

1. Nome da função de *loop* infinito que será a *task*;
2. Nome descritivo da função (para fins de depuração do código);
3. Tamanho da pilha em palavras. O valor real da pilha é calculado a partir desse valor e da largura de pilhas (outro valor configurado separadamente que define o tamanho das palavras);
4. Parâmetros que devem ser passados para a *task* quando ela for criada (usar NULL se nenhum valor for necessário);
5. Prioridade da *task* (quanto menor o valor, maior a prioridade);
6. Nome de um manuseador (para que a *task* possa ser alterada por outras *tasks* e vice-versa).

Se uma *task* for criada com sucesso, ela retornará uma mensagem durante a execução do código, e outra mensagem caso contrário (BARRY, 2010).

É importante notar que as *tasks* funcionam como *threads*, pois apesar de trabalharem em conjunto para um aplicativo e compartilhar informações entre si, elas concorrem para usar os núcleos de um microcontrolador, usando sua prioridade para obter vantagem nessa competição. Além disso elas possuem estados, algo comum entre processos e *threads*.

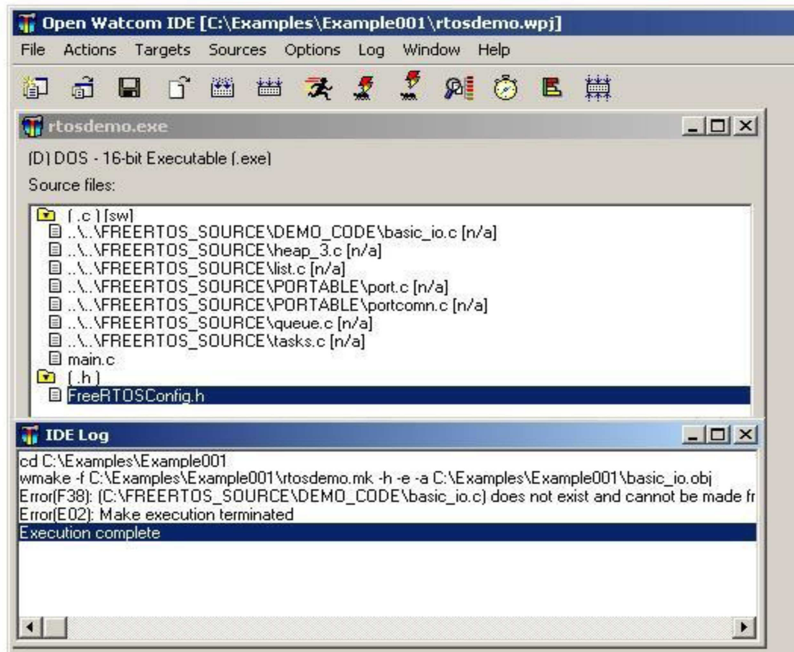
4.4 *Open Watcom*

O *Open Watcom* é o sucessor de código aberto dos conjuntos de compiladores e ferramentas de desenvolvimento comercializados pela *Sybase*, *Powersoft* e *WATCOM International Corp* (OPEN WATCOM COMMUNITY, 2010). Ele é desenvolvido por uma comunidade, e está licenciado sob licença pública pela Sybase.

É uma ferramenta útil para executar os exemplos do *FreeRTOS* que vêm juntos ao manual de referência, útil em especial para entender o funcionamento de *tasks* no *FreeRTOS* (BARRY, 2010). Infelizmente, o *Open Watcom* não possui compatibilidade com o *Proteus*.

A Figura 11 demonstra uma tela de projeto do *Open Watcom*.

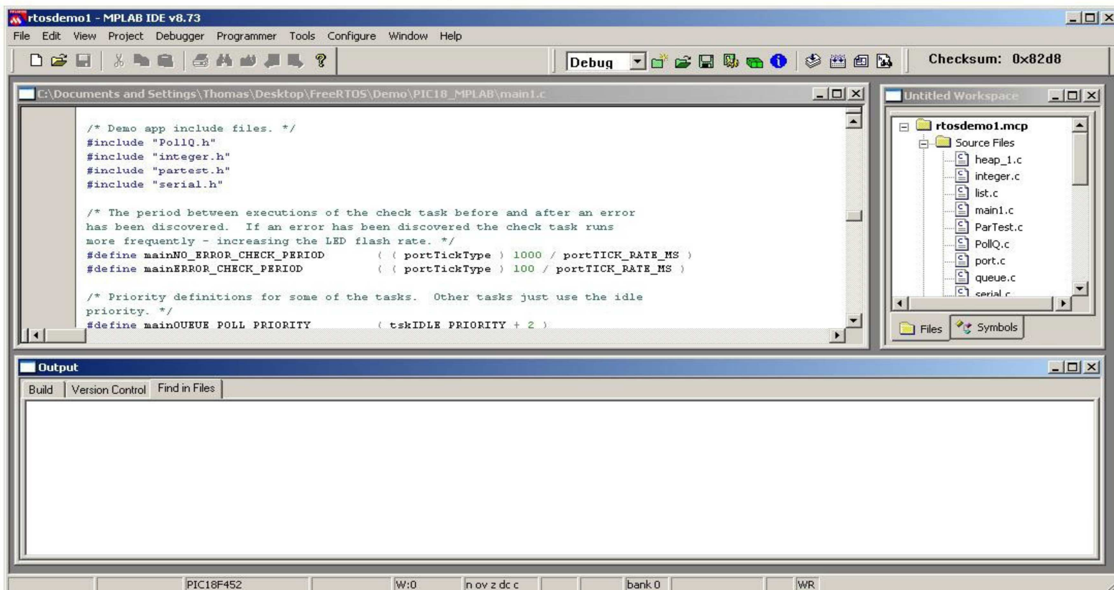
Figura 11 – Tela de Projeto do *Open Watcom*



4.5 *MPLAB*

O *MPLAB* é uma *IDE* (Ambiente Integrado de Desenvolvimento do inglês *Integrated Development Environment*) para microcontroladores da *Microchip Technology* que facilita o desenvolvimento de projetos com múltiplos arquivos, e possui compatibilidade com o uso do *FreeRTOS*. Por ser apenas um *IDE*, o *MPLAB* precisa dos compiladores para funcionar (MICROCHIP TECHNOLOGY, 2005), como o *MPLAB C18*, essencial para se trabalhar com microcontroladores *PIC18*. A Figura 12 demonstra a *IDE* do *MPLAB*.

Figura 12 – *IDE* do *MPLAB*



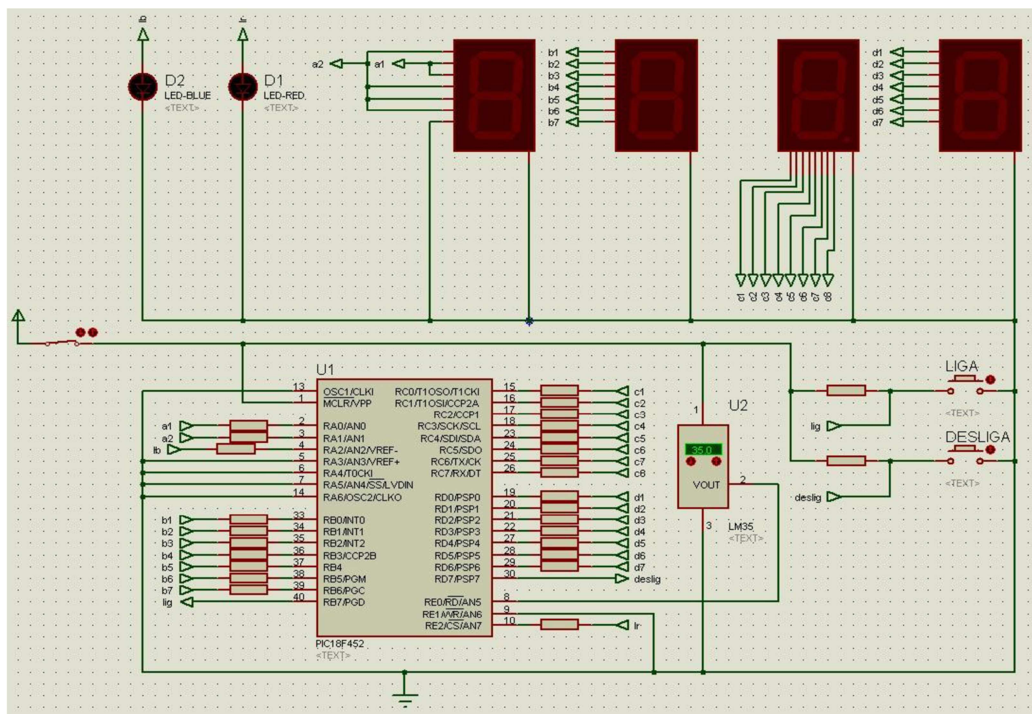
5 EXPERIMENTOS

No decorrer do projeto, foram propostos alguns experimentos, cada um com uma finalidade específica. A descrição de cada um, assim como seus resultados, podem ser encontrados nas seções a seguir. As partes em cinza nos quadros não fazem parte do programa propriamente dito, mas explicam os códigos acima delas com mais detalhes do que os comentários dentro dos programas.

5.1 Desenvolvimento Gradual de um Sistema Embarcado

Durante o segundo experimento, foram usados o *Proteus* 7.4 SP3 (Build 6792) e *MikroC* 8.1.0.0. A proposta deste experimento foi desenvolver um sistema embarcado para ler a temperatura informada por um sensor e mostrar em *displays* de segmentos, evoluindo periodicamente esse sistema. As implementações incluíram botões de ligar e desligar, um interruptor de emergência e *LED*s para indicar se a temperatura está muito alta ou muito baixa. Para simplificação dos circuitos, foram usados *inputs* e *outputs*, que basicamente transferem o sinal obtido entre si. Uma representação da versão mais recente do sistema pode ser encontrada na Figura 13, enquanto a codificação do programa pode ser encontrada na subseção 5.2.1.

Figura 13 – Sistema Embarcado Desenvolvido Gradualmente



5.1.1 Código do Programa Desenvolvido Gradualmente

O Quadro 1 apresenta o código do programa desenvolvido no *MikroC* durante esse experimento. Como a proposta para o experimento foi desenvolver gradualmente, e não apenas explorar um sistema embarcado, procurou-se utilizar maneiras menos repetitivas e mais legíveis de desenvolvimento, como o emprego de funções extras. Em especial, foi usada uma única função para converter algarismos para cada *display* de sete segmentos, e outra função para converter os dados capturados e enviados pelos sensores de temperatura:

Quadro 1 – Código de um sistema embarcado desenvolvido gradualmente

int ligdeslig(int estadoanterior) { //função para checar botões de ligar e desligar
Uma função que deve receber um valor numérico do tipo inteiro e retornar outro valor numérico do tipo inteiro. Ela foi feita com a intenção de checar se um entre dois botões definidos foi ou está pressionado, retornando um valor de sinalização usado pela função principal do programa para ligar ou desligar os *ports* do microcontrolador. O valor recebido por essa função é o valor de sinalização que representa o estado atual dos *ports*, e é retornado caso nenhum botão tenha sido pressionado. Isto é feito para evitar e facilitar a manipulação dos *ports*.
F (Button(&PORTB, 7, 2000, 0)) return (1); // um modo de checar o estado de botões
Esse comando verifica se o primeiro botão, conectado no sétimo pino do *port* “B”, está ou foi pressionado, retornando um valor usado para sinalização. Na função principal, o valor retornado é verificado e usado para ligar os ports desligados do microcontrolador.
F (Button(&PORTD, 7, 2000, 0)) return (0);

Esse comando verifica se o segundo botão, conectado no sétimo pino do *port* "D", está ou foi pressionado, retornando um valor usado para sinalização. Na função principal, o valor etornado é verificado e usado para desligar os ports ligados do microcontrolador.

return (estadoanterior); // retorna o estado anterior se nenhum botão tiver sido
pressionado
}

Caso nenhum botão tenha sido pressionado, está função deve retornar o valor de sinalização recebido, simulando o pressionamento de um dos botões, para que o programa possa continuar executando.
float capturartemp(int pinoan) { // função para capturar o valor de um pino analógico e converte-lo para formato digital. Nesse caso ele está sendo usado para capturar um valor de um sensor de temperatura
Uma função que deve receber um valor numérico do tipo inteiro e retornar um valor numérico do tipo real. Ela serve para capturar e converter para o formato digital um valor que está sendo enviado para um pino analógico de um microcontrolador. Para tanto, essa função deve receber um valor que representa o pino que deve ser verificado. Essa função foi construída com o objetivo de verificar os dados emitidos por um sensor de temperatura .

return ((float)((5. * Adc_read(pinoan) * 100.) / 1024)); //fórmula de conversão de dados
analógicos
}

Esse comando usa uma fórmula para converter um valor analógico (nesse caso recebido do sensor de temperatura) para o formato digital e retorna o valor para a função que pediu a verificação (nesse caso, a função principal).
int disp7seg(int unidade) { // função de ativação de pinos para displays de 7 segmentos
Uma função que deve receber um valor numérico do tipo inteiro e retornar outro valor numérico do tipo inteiro. Ela tem como objetivo verificar um algarismo e retornar a conversão desse algarismo para um valor que o represente em um *port* conectado apropriadamente a um *display* de sete segmentos do tipo cátodo.

switch(unidade) // verifica qual o digito obtido
{

Esse comando é um comando condicional, ou seja, toma decisões baseados em cálculos, e comparações de valores. O valor que esta sendo verificado nesse caso é o valor recebido como algarismo.

case 0: return(63); // cada valor representa um conjunto de pinos sequenciais que devem estar ativos para mostrar o número adequado
case 1: return(6);
case 2: return(91);
case 3: return(79);
case 4: return(102);
case 5: return(109);
case 6: return(125);
case 7: return(7);
case 8: return(127);
case 9: return(111);

Cada um desses comandos é uma condição a qual o algarismo pode assumir. Para cada caso possível, é feita uma conversão para um formato legível para um *display* de sete segmentos do tipo cátodo, e depois enviada como resposta à chamada da função.

default: return(0);
}
}

Caso o algarismo não seja legível ou válido, é retornado um valor que não mostra nada em um *display* de sete segmentos do tipo cátodo.
int tempalta(int tempalt, int valoralt) { // função usada para comparar a temperatura atual e a temperatura superior de alerta
Uma função que recebe dois valores numéricos do tipo inteiro e retorna outro valor numérico do tipo inteiro. Ela foi feita com o objetivo de verificar se a temperatura ultrapassou um limite definido.

F (tempalt >= valoralt) return (1); // temperatura acima do limite
else return (0); // temperatura abaixo do limite
}

Esse comando condicional está verificando se a temperatura atual informada ultrapassou ou não o limite informado, retornando um valor de sinalização que deve ser interpretado pelo programa.
int tempbaixa(int tempabx, int valorabx) { // função usada para comparar a temperatura atual e a temperatura inferior de alerta
Outra função que recebe dois valores numéricos do tipo inteiro e retorna um valor numérico do tipo inteiro. Similar a anterior, essa função foi feita com o objetivo de comparar se a temperatura ultrapassou um limite definido, porém essa função verifica o limite de temperatura mínima.

F (tempabx <= valorabx) return (1); // temperatura acima do limite inferior
else return (0); // temperatura abaixo do limite inferior
}

Esse comando condicional está verificando se a temperatura atual informada é inferior ou não ao limite informado, retornando um valor de sinalização que deve ser interpretado pelo programa.
void desligar() { // função para desligar/hibernar os ports
Essa é uma função apenas de ação, ela não recebe e nem retorna nenhum valor para a função que a invocou. Ela foi feita para desligar ou hibernar os *ports* do microcontrolador, mas não o microcontrolador em si.

PORTA = 0; // todas os ports usadas são hibernados, funcionando apenas os pinos de
entrada de dados
PORTB = 0;
PORTC = 0;
PORTD = 0;
PORTE = 0;

}
Esse conjunto de comandos faz com que os *ports* do microcontrolador emitam sinal de baixa freqüência, efetivamente hibernando-os e impedindo os displays de sete segmentos do tipo cátodos conectados de emitir qualquer informação. Porém, devido a natureza das configurações dos pinos que recebem dados, esses não são hibernados e continuam funcionando normalmente.
void main() { // função principal
A função principal e essencial do programa. Devido às mudanças no estilo de programação desde o último experimento, nesse programa ela majoritariamente invoca as outras funções.
int inttemp10, centena, ligado = 0; // declaração de variáveis
Declarações de variáveis para manipulação das funções. A primeira variável é usada para guardar o valor da temperatura multiplicado por 10 para fins de cálculos e comparações, a segunda variável é usada para calcular e manipular a centena dos valores de temperatura recebidos, já que esse necessitou um tratamento especial devido ao limite de pinos do microcontrolador, e a terceira variável é usada para verificar e dizer se os *ports* estão hibernando ou não (o programa começa com os *ports* hibernando).

TRISA = 0, PORTA = 0; //inicia e configura os ports do microcontrolador
TRISB = 0; PORTB = 0;
TRISC = 0; PORTC = 0;
TRISD = 0; PORTD = 0;
TRISE = 0; PORTE = 0;

Esse conjunto de variáveis serve para configurar os *ports* do microcontrolador, definindo o estado inicial e as permissões de recebimento e envio de dados para cada *port*. Nesse caso, os *ports* “A”, “B”, “C”, “D” e “E” são configurados para envio de dados, mas isto não impede que eles também recebam dados.
while(1) { // laço infinito
O comando *while* é um comando de retorno condicional, ou seja, o programa ou função retorna quando determinada condição é atingida, e prossegue com o programa ou função caso não seja atingida. Nessa situação, a condicional é “1”, o que significa “verdadeiro” e deixa o programa com uma condicional sempre verdadeira, criando o *loop* infinito recomendado para programas embarcados.

ligado = ligdeslig(ligado); // verificar botões de ligar e desligar
if (ligado) { // o dispositivo está ligado

É atribuído a uma variável o retorno da função que verifica se algum botão (de ligar ou hibernar o microcontrolador) foi pressionado, e logo após é verificado, através do valor obtido, se o microcontrolador deve ser religado ou hibernado.

inttemp10 = (int)(capturartemp(5)*10); // captura a temperatura e multiplica por 10
para efeito de comparação de dados

Caso o microcontrolador não esteja hibernando, É atribuído a uma variável e multiplicado por 10 (para fins de cálculos posteriores) o valor de temperatura que um dos pinos analógicos está recebendo.

F ((inttemp10 / 1000) < 1) centena = 3; // calcula centenas
else centena = 1;

Esse comando calcula o valor necessário para demonstrar a centena no *display* de sete segmentos do tipo cátodo, já que, devido ao limite de temperatura que o sensor de temperatura aquenta e a necessidade de compartilhar pinos no *port* A devido ao limite de pinos no microcontrolador, foi necessário um tratamento especial.
PORTA = centena + tempbaixa(inttemp10/10, 10)*4; // verifica se a temperatura está baixa. Como compartilha pinos com o display de centenas e alterar apenas um pino envolve o port inteiro, ajustes foram necessários
Devido ao *LED* que representa se a temperatura está muito baixa estar ligado ao *port* que requer tratamento especial, foi necessário um calculo para que não houvesse conflito. Esse código não somente demonstra o valor certo da centena da temperatura, como também liga o *LED* se a temperatura estiver abaixo de 10ºC.
PORTE.RE2 = tempalta(inttemp10/10, 60); // verifica se a temperatura está alta
Essa chamada de função em um pino em um pino do *port* “E” deve ligar o *LED* de temperatura alta se a temperatura estiver acima de 60ºC
PORTB = disp7seg(inttemp10 % 1000 / 100); // calcula dezenas
Essa chamada de função em um pino do port “B” deve converter o algarismo de dezena da temperatura e demonstrar no *display* de sete segmentos apropriado.
PORTC = disp7seg(inttemp10 % 100 / 10) + 128; // calcula unidades
Essa chamada de função em um pino do port “C” deve converter o algarismo de unidade da temperatura e demonstrar no *display* de sete segmentos apropriado. É adicionado o número 128 ao cálculo, pois o *display* de sete segmentos usado nesse caso possui um ponto, usado para números fracionados e acionado pela adição de 128 no cálculo quando ligado apropriadamente no *port* do microcontrolador.
PORTD = disp7seg(inttemp10 % 10); // calcula decimais
Essa chamada de função em um pino do port “D” deve converter o algarismo do decimal da temperatura e demonstrar no *display* de sete segmentos apropriado
}
Essa chave indica o fim da condicional para caso o microcontrolador esteja ativo. É importante notar que toda a coleta de dados, cálculos e comparações relacionados ao sensor de temperatura são feitos apenas nesse caso.
else desligar(); // o dispositivo está desligado
Caso o microcontrolador deva estar hibernando, toda a coleta de dados, cálculos e comparações relacionados ao sensor de temperatura são pulados, e é chamado uma função para hibernar e para ter certeza de que os *ports* do microcontrolador estejam hibernando.
}
Essa chave marca o fim do comando while definido anteriormente, e a parte final do loop infinito. Esse programa deve voltar ao inicio do comando while em condições normais, já que a condicional definida será sempre verdadeira.
}
Essa chave marca o fim do programa. O programa não deve chegar até essa marca em condições normais.

5.2 Sensores de Temperatura com *Displays* de LCD

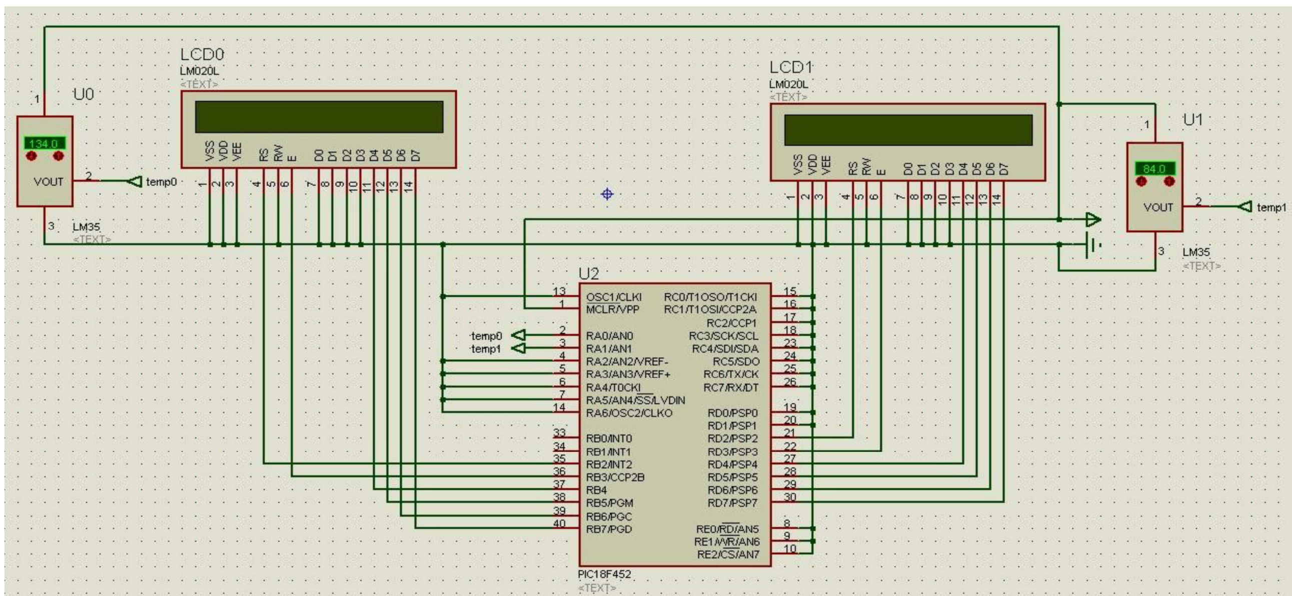
Durante este experimento, foram usadas as ferramentas *Proteus* 7.4 SP3 (Build 6792), *MikroC* 8.1.0.0, *Open Watcom* 1.9, MPLAB IDE 8.73 e *MPLAB C18* 3.39. Foi proposto o desenvolvimento de um sistema embarcado sem sistema operacional que gerencia dois sensores de temperatura sequencialmente, e depois disso, a evolução do sistema para um sistema operacional embarcado de tarefas concorrentes com o *FreeRTOS* e o *MPLAB*

A primeira etapa do experimento ocorreu sem muitos problemas, porém houve problemas na segunda etapa com o *MPLAB*. Nem mesmo os exemplos do *FreeRTOS* com o *MPLAB* executaram corretamente, o que indica que seria necessário um tempo consideravelmente maior para o estudo da mesma ferramenta.

O motivo pelo qual não foi usado o *MikroC na implementação do sistema embarcado* é porque este não possui compatibilidade com o *FreeRTOS*.

O projeto eletrônico para ambos os casos pode ser encontrado na Figura 14, enquanto a codificação para o programa sem o sistema embarcado pode ser encontrado na subseção 5.3.1, e com o sistema embarcado (com as limitações descritas anteriormente) na subseção 5.3.2.

Figura 14 – Sensores de Temperatura com *Displays* de LCD



5.2.1 Código de Leitura de Sensores de Temperatura sem SO

O Quadro 2 apresenta o código do programa sem sistema operacional embarcado desenvolvido no *MikroC* durante esse experimento. Foram usadas apenas duas funções além da função principal, uma função para facilitar a coleta e a conversão dos dados dos sensores de temperatura, e outra função para imprimir textos nos *displays* de *LCD*, cabendo à função principal fazer a conversão de valores numéricos para textos:

Quadro 2 – Código para dois displays de LCD

int calctemp(int port) { // função para capturar temperatura
Uma função que deve receber um valor numérico do tipo inteiro e retornar outro valor numérico do tipo inteiro. Ela serve para capturar e converter para o formato digital um valor que está sendo enviado para um pino analógico de um microcontrolador. Para tanto, essa função deve receber um valor que representa o pino que deve ser verificado. Essa função foi construída com o objetivo de verificar os dados emitidos por um sensor de temperatura.
return ((int)((5. * Adc_read(port) * 100.) / 1024 + 0.5)); }
Esse comando usa uma fórmula para converter um valor analógico (nesse caso recebido do sensor de temperatura) para o formato digital e retorna o valor para a função que pediu a verificação (nesse caso, a função principal).
void printlcd(char text) { // função para imprimir em um display de LCD
Essa função não retorna nenhum valor, apesar de receber um texto. Ela tem como objetivo imprimir o texto enviado no *display* de *LCD* ativo da função.
LCD_Cmd(LCD_CURSOR_OFF);
Esse comando desliga e esconde o *cursor* do *display* de *LCD* ativo. Reusar esse comando garante que o *cursor* continuará desligado e escondido.
LCD_Out(1,1,text);

Esse comando imprime no *display* de *LCD* ativo, começando da primeira posição, o texto contido na variável recebida. O motivo para não usar um comando para limpar o *display* de *LCD* previamente é porque o simulador estava religando o *cursor*, porém não houve problemas com a demonstração dos resultados porque os textos possuem sempre o mesmo tamanho e sobrescrevem os anteriores.
Delay_ms(200); }
Esse comando faz com que o processador espere 200 ciclos de processamento antes de continuar a execução do programa.
void main() { // função principal
A função principal e essencial do programa.
char temp0[17], temp1[17]; // variáveis para guardar e transferir texto para os displays
Declarações de variáveis para manipulação das funções. Ambas as variáveis são usadas para guardar as temperaturas dos sensores em formato de texto, para facilitar a impressão delas.

TRISA = 0, PORTA = 0; // configuração dos ports do microcontrolador
TRISB = 0; PORTB = 0;
TRISC = 0; PORTC = 0;
TRISD = 0; PORTD = 0;
TRISE = 0; PORTE = 0;

Esse conjunto de variáveis serve para configurar os *ports* do microcontrolador, definindo o estado inicial e as permissões de recebimento e envio de dados para cada *port*. Nesse caso, os *ports* “A”, “B”, “C”, “D” e “E” são configurados para envio de dados, mas isto não impede que eles também recebam dados.
LCD_Init(&PORTB); // inicia o LCD do port B
Esse comando ativa o *display* de *LCD* ligado ao *port* “B”.
LCD_Cmd(LCD_CLEAR); // limpa o conteúdo do LCD
Esse comando limpa qualquer texto que possa estar contido no *display* de *LCD* ativo, que deve ser o *display* ligado ao *port* “B”. O uso desse comando é o motivo pelo qual os *displays* de *LCD* já estão sendo ativados.
LCD_Cmd(LCD_CURSOR_OFF); // esconde o cursos do LCD
Esse comando desliga e esconde o *cursor* do *display* de *LCD* ativo.
LCD_Init(&PORTD); // inicia o LCD do port D
Esse comando ativa o *display* de *LCD* ligado ao *port* “D”, desligando qualquer outro *display* de *LCD* ativo
LCD_Cmd(LCD_CLEAR); // limpa o conteúdo do LCD
Esse comando limpa qualquer texto que possa estar contido no *display* de *LCD* ativo, que deve ser o *display* ligado ao *port* “D”. O uso desse comando é o motivo pelo qual os *displays* de *LCD* já estão sendo ativados.
LCD_Cmd(LCD_CURSOR_OFF); // esconde o cursos do LCD
Esse comando desliga e esconde o *cursor* do *display* de *LCD* ativo.
while(1) { // função de loop infinito
O comando *while* é um comando de retorno condicional, ou seja, o programa ou função retorna quando determinada condição é atingida, e prossegue com o programa ou função caso não seja atingida. Nessa situação, a condicional é “1”, o que significa “verdadeiro” e deixa o programa com uma condicional sempre verdadeira, criando o *loop* infinito recomendado para programas embarcados.

IntToStr(calctemp(0), temp0); // transforma um valor numérico em um valor de texto.
IntToStr(calctemp(1), temp1);

Cada um desses comandos serve para converter um valor numérico do tipo inteiro contido em seu primeiro parâmetro, em um valor de texto, que é transferido para o segundo parâmetro (que por sua vez necessita ser uma variável). É importante notar que o primeiro parâmetro usado em ambos os comandos foi uma função, o que significa que essa função será invocada e seu retorno, desde que seja válido, será convertido.
LCD_Init(&PORTB); // reinicia o LCD do port B
Esse comando reinicia e ativa o display de *LCD* conectado ao *port* “B”, desligando o conectado ao *port* “D”.
printlcd(temp0); // imprime o conteúdo da variável no LCD
Esse comando invoca a função para escrever um texto no *display* de *LCD* ativo, enviando o valor guardado na primeira variável de texto.
LCD_Init(&PORTD); // reinicia o LCD do port D, desabilitando o do port B
Esse comando reinicia e ativa o display de *LCD* conectado ao *port* “D”, desligando o conectado ao *port* “B”.
printlcd(temp1); // imprime o conteúdo da variável no LCD
Esse comando invoca a função para escrever um texto no *display* de *LCD* ativo, enviando o valor guardado na primeira variável de texto.
}
Essa chave marca o fim do comando *while* definido anteriormente, e a parte final do *loop* infinito. Esse programa deve voltar ao inicio do comando *while* em condições normais, já que a condicional definida será sempre verdadeira.
}
Essa chave marca o fim do programa. O programa não deve chegar até essa marca em condições normais.

5.3 *LED*s Piscantes

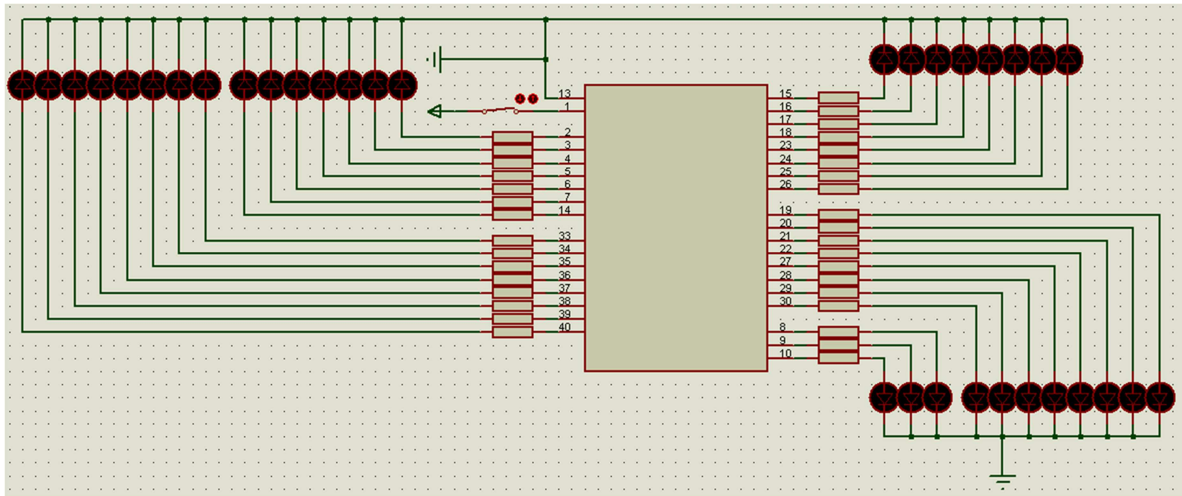
Como houve alguns problemas em utilizar o *MPLAB* para realizar experimentos com o *FreeRTOS* no *Isis*, algumas alternativas foram pesquisadas. Uma delas levou às modificações de Milinkovic (2012) no *FreeRTOS* para possibilitar o funcionamento sob o *mikroC PRO for ARM*. As suas modificações incluem alterações em certas bibliotecas do *FreeRTOS*, como *queue.c* e *tasks.c*, seguido de um programa de testes que está na versão 0.0.0.4.

Neste experimento foi estudada a versão 0.0.0.1 do programa de testes desenvolvido por Milinkovic (2012) e replicados os resultados para um projeto sem sistema operacional embarcado, utilizando o *mikroC*

8.1.0.0. O *mikroC PRO for ARM 2.50* foi necessário para compilar o projeto de Milinkovic, já que este foi desenvolvido para uso no microcontrolador *ARM STM32 M3*.

Em nota especial, o *Isis* do *Proteus* 7.4 SP3 (*Build* 6792) não possui suporte para este microcontrolador (porém algumas bibliotecas de terceiros podem ser utilizadas para adicionar suporte a diversos microcontroladores, incluindo este). A Figura 15 apresenta o projeto implementado, independente de compilador ou de microcontrolador (mas vale lembrar que podem haver restrições dependendo da configuração de cada pino para cada microcontrolador).

Figura 15 – Projeto de *LED*s Piscantes



O Quadro 3 demonstra o código para este projeto em um microcontrolador da família *PIC*, sem a utilização de um sistema operacional embarcado.

Quadro 3 – *LED*s Piscantes sem Sistema Operacional Embarcado

void main() {
A função principal e essencial do programa.
PORTA = 0; TRISA = 0; PORTB = 0; TRISB = 0; PORTC = 0; TRISC = 0; PORTD = 0; TRISD = 0; PORTE = 0; TRISE = 0;
Este conjunto de variáveis serve para configurar os *ports* do microcontrolador, definindo o estado inicial e as permissões de recebimento e envio de dados para cada *port*. Neste caso, os *ports* “A”, “B”, “C”, “D” e “E” são configurados para envio e recebimento de dados. Também inicia todos os *port*s em “0”, o que significa que todos os pinos estarão desligados e consecutivamente os *LED*s.
while(1) {
O comando *while* é um comando de retorno condicional, ou seja, o programa ou função retorna quando determinada condição é atingida, e prossegue com o programa ou função caso não seja atingida. Nessa situação, a condicional é “1”, o que significa “verdadeiro” e deixa o programa com uma condicional sempre verdadeira, criando o *loop* infinito recomendado para programas embarcados.
PORTA = ~PORTA; PORTB = ~PORTB; PORTC = ~PORTC; PORTD = ~PORTD; PORTE = ~PORTE;
Estes comandos utilizam o ~ para realizar uma inversão binária, acendendo os pinos do *port* caso ele esteja desligado e desligando caso ele esteja aceso. Isto permite acender e apagar os *LED*s conectados aos pinos.
Delay_ms(100);
Este comando faz com que o núcleo de processamento espere 100 ciclos antes de continuar a executar o programa. Neste caso, ele é utilizado para que um usuário humano possa observar o resultado antes que seja alterado novamente.
}
Esta chave marca o fim do comando *while* definido anteriormente, e a parte final do *loop* infinito. Este programa deve voltar ao inicio do comando *while* em condições normais, já que a condicional definida será sempre verdadeira.
}
Esta chave marca o fim do programa. O programa não deve chegar até essa marca em condições normais.

O Quadro 4 demonstra o programa de Milinkovic (2012), que foi desenvolvido para um microcontrolador *ARM STM32 M3*.

Quadro 4 – *LED*s Piscantes com Sistema Operacional Embarcado

#include "FreeRTOS.h" #include "task.h"
As bibliotecas necessárias do *FreeRTOS* para o funcionamento deste programa. São usadas as duas mais essenciais, sendo a biblioteca *task* responsável pelo gerenciamento das tarefas.
#define ledSTACK_SIZE configMINIMAL_STACK_SIZE #define mainBLINKING_TASK_PRIORITY (tskIDLE_PRIORITY + 1) #define TaskDelay_ms(x) vTaskDelay(x/portTICK_RATE_MS)
Algumas contantes definidas para facilitar a manipulação da aplicação.
static portTASK_FUNCTION(vLedBlinkingTask, pvParameters) {
Está função na verdade é uma tarefa que será criada.
for (;;) {
O comando de *loop* infinito necessário para a *task*.

GPIOA_ODR = ~GPIOA_ODR; // Toggle PORTA
GPIOB_ODR = ~GPIOB_ODR; // Toggle PORTB
GPIOC_ODR = ~GPIOC_ODR; // Toggle PORTC
GPIOD_ODR = ~GPIOD_ODR; // Toggle PORTD
GPIOE_ODR = ~GPIOE_ODR; // Toggle PORTE

Estes comandos utilizam o ~ para realizar uma inversão binária, acendendo os pinos do *port* caso ele esteja desligado e desligando caso ele esteja aceso. Isto permite acender e apagar os *LED*s conectados aos pinos.
TaskDelay_ms(100);
Este comando faz com que o núcleo de processamento espere 100 ciclos antes de continuar a executar o programa. Neste caso, ele é utilizado para que um usuário humano possa observar o resultado antes que seja alterado novamente.
}
Esta chave marca o fim do comando de *loop* infinito definido anteriormente. Esta tarefa deve voltar ao inicio do *loop* em condições normais, já que a condicional definida será sempre verdadeira.
}
Esta chave marca o fim da *task*. O programa não deve chegar até essa marca em condições normais.
void main(void) {
A função principal e essencial do programa.

GPIO_Digital_Output(&GPIOA_BASE, _GPIO_PINMASK_ALL); // Set PORTA as digital output
GPIO_Digital_Output(&GPIOB_BASE, _GPIO_PINMASK_ALL); // Set PORTB as digital output
GPIO_Digital_Output(&GPIOC_BASE, _GPIO_PINMASK_ALL); // Set PORTC as digital output
GPIO_Digital_Output(&GPIOD_BASE, _GPIO_PINMASK_ALL); // Set PORTD as digital output
GPIO_Digital_Output(&GPIOE_BASE, _GPIO_PINMASK_ALL); // Set PORTE as digital output

Este conjunto de variáveis serve para configurar os *ports* do microcontrolador, definindo o estado inicial e as permissões de recebimento e envio de dados para cada *port*. Neste caso, os *ports* “A”, “B”, “C”, “D” e “E” são configurados para envio dados no formato digital.
GPIOA_ODR = 0; GPIOB_ODR = 0; GPIOC_ODR = 0; GPIOD_ODR = 0; GPIOE_ODR = 0;
Inicia todos os *port*s em “0”, o que significa que todos os pinos estarão desligados e consecutivamente os *LED*s.
xTaskCreate(vLedBlinkingTask, (signed char *) "LED", ledSTACK_SIZE, NULL, mainBLINKING_TASK_PRIORITY, (xTaskHandle *) NULL);
Prepara a única *task* ou tarefa do programa, mas ainda não a inicia.
vTaskStartScheduler();
Inicia o agendador de tarefas, responsável por iniciar e gerenciá-las, utilizando para isso as configurações de prioridades de cada *task*.
}
Esta chave marca o fim do programa. O programa não deve chegar até essa marca em condições normais.

Comparando os Quadros 3 e 4, é perceptível que o sistema embarcado com sistema operacional é mais complexo e necessita de mais memória, porém é mais maleável para mudanças, o que facilita extensão, como Milinkovic (2012) fez com versões subsequentes, e a utilização do aplicativo como base à outros aplicativos, facilitando a padronização de projetos futuros. Ainda assim, o sistema embarcado sem o sistema operacional não é tão exigente em relação à memória e poder de processamento do microcontrolador, evidenciado não apenas pelo código, mas também por não necessitar da inclusão de bibliotecas adicionais, e isto é um diferencial importante em projetos de baixo custo.

Também é notável que desenvolver sistemas embarcados utilizando o *FreeRTOS* é consideravelmente diferente do que desenvolver sem um sistema operacional, especialmente devido ao uso de *tasks*, o que significa que, em uma equipe de desenvolvimento ou em empresa em migração para utilização de sistemas operacionais embarcados, um treinamento talvez seja necessário.

Além disso, Milinkovic (2012) precisou fazer alterações no *FreeRTOS* para funcionar adequadamente, implicando que um conhecimento adicional sobre os sistemas operacionais embarcados é necessário para a sua utilização correta, e também demonstra que os sistemas operacionais embarcados possuem incompatibilidades com alguns ambientes de desenvolvimento.

6 COMENTÁRIOS FINAIS

Devido diferenças e restrições dos sistemas embarcados se comparados com outros modelos de sistemas mais clássicos, há uma dificuldade em utilizá-los corretamente, mas eles podem trazer benefícios não só para o mercado, mas para os clientes também, especialmente considerando que a maior parte dos sistemas computacionais produzidos são sistemas embarcados. Porém, é uma tecnologia que requer planejamento para seu uso, pois seus produtos tendem a se tornar obsoletos rapidamente.

Por outro lado, os sistemas operacionais embarcados podem ajudar a padronizar e encurtar a fase de desenvolvimento, aumentando a vida útil do produto no mercado, especialmente de projetos complexos, mas a utilização de sistemas operacionais embarcados requer um estudo sobre as vantagens e desvantagens, assim como a utilização dos mesmos. Sem um estudo ou conhecimento prévio, desenvolvedores de *software* que utilizam tais sistemas operacionais podem desenvolver produtos mais lentos do que deveriam ser.

Os objetivos do artigo, que incluíram desenvolver um estudo exploratório e discutir a importância atual de sistemas embarcados, identificar e comparar os principais sistemas operacionais embarcados utilizados detalhando suas características e descrevendo suas aplicações, diferenciar sistemas operacionais tradicionais de sistemas operacionais embarcados e apresentar a importância de seu uso, foram todos cumpridos.

Este artigo traz ainda a oportunidade aos leitores de conhecer um pouco sobre sistemas embarcados e sistemas operacionais embarcados.

Os resultados obtidos durante o desenvolvimento dos experimentos não foram muito diferentes dos obtidos durante a revisão bibliográfica, especialmente considerando as limitações físicas de um sistema embarcado. Os principais resultados obtidos foram:

- Sistemas tradicionais diferem consideravelmente de sistemas embarcados;
- Sistemas embarcados são fisicamente bem limitados;
- Sistemas embarcados possuem funções bem especificadas;
- Sistemas embarcados geralmente são rápidos;
- Produtos embarcados possuem uma janela de tempo de desenvolvimento e de venda curtos devido a constante evolução tecnológica;
- Sistemas operacionais embarcados são melhores para aplicações complexas devido a suporte à programação multitarefas;
- Sistemas operacionais embarcados desempenham bem mesmo em aplicações simples;
- Sistemas operacionais embarcados ajudam a padronizar projetos de sistemas embarcados independentes de complexidade;
- Sistemas operacionais embarcados modificam consideravelmente o estilo de programação;
- Padronização é essencial para o desenvolvimento de sistemas embarcados, pois diminuem o tempo gasto com desenvolvimento;
- Há muitas ferramentas para o uso e simulação de tecnologia embarcada;

Com os resultados obtidos e com os estudos da revisão bibliográfica, pode-se concluir que:

- Há um pouco de confusão ou discórdia entre diferentes autores sobre a definição correta de sistemas embarcados, especialmente sobre seus nomes alternativos, mas ainda assim, todos concordam que sistemas embarcados são fisicamente limitados e desempenham poucas funções, mas ainda assim, são precisos e rápidos;
- O mercado está parcialmente ciente e interessado nesta tecnologia devido aos vários benefícios que ela traz quando seus desafios e dificuldades são conquistados;
- Sistemas tradicionais diferem consideravelmente de sistemas embarcados, mas é necessário estudar o primeiro para que se possa estudar corretamente o segundo. O mesmo pode ser dito entre sistemas operacionais tradicionais e sistemas operacionais embarcados e,

- Os sistemas operacionais embarcados, apesar de úteis, trazem ainda mais desafios para seu uso correto, mas não deixam de ser essenciais em projetos muito complexos.

REFERÊNCIAS

BARRY, R. **Using the FreeRTOS**™ Real Time Kernel, A Practical Guide. Manual técnico digital, Versão 1.3.2, Real Time Engineers Ltd., 2010. Disponível em: http://www.freertos.org/Documentation/FreeRTOS-documentation-andbook.html

CARRO, L.; WAGNER, F. R.; **Sistemas computacionais embarcados**. Jornadas de Atualização em Informática da SBC, 2003.

FRIEDRICH, L. F.; A **Survey of operating systems infrastructure for embedded systems**. Relatório Técnico, Faculdade de Ciências da Universidade de Lisboa, Lisboa, 2009.

KONANA, P.; RAY G.. Physical product reengineering with embedded information technology. Artigo Técnico, **Revista Communications of the ACM**, vol. 50, nº 10. Outubro, 2007.

MICROCHIP TECHNOLOGY. **MPLAB® IDE** user's Guide. Manual técnico digital, 2005 Disponível em: http://ww1.microchip.com/downloads/en/devicedoc/51519a.pdf

MILINKOVIC, S. Porting **FreeRTOS to mikroC for STM32 M3**. Disponível em: <http://www.libstock.com/projects/view/370/porting-freertos-to-mikroc-for-stm32-m3>. Acesso em: 15 nov. 2012.

MIKROELETRONIKA. **MikroC**: user's manual. Manual Técnico Digital, 2006. Disponível em: http://www.mikroe.com/pdf/mikroC/mikroC_manual.pdf

OPEN WATCOM COMMUNITY , SYBASE. **Open Watcom 1.9 C/C++:** getting started help, Seção "Introduction to Open Watcom C/C++", Manual de *Software*, 2010. Disponível em: instalação do Open Watcom 1.9

TANENBAUM, A. S. **Sistemas operacionais modernos**. 2.ed. São Paulo: Livro Técnico, Prentice Hall, 2003.

Leituras Complementares

CARRO, L.; WAGNER, F. R. **Metodologias e técnicas de engenharia de software para sistemas embarcados**, Jornadas de Atualização em Informática da SBC, 2003.

LABCENTER ELETRÔNICS. **Proteus Design Suíte Product Guide**. Guia Online. Disponível em: http://downloads.labcenter.co.uk/proteus7brochure.pdf

TAURION, C. **Software embarcado**: a nova onda da informática. Rio de Janeiro: Livro Técnico, Brasport: 2005.

ANTI ESSAYS Mobile **Os History Essay**. Disponível em: <http://www.antiessays.com/free-essays/189371.html>. Acesso em: 26 out. 2012.

BRAIN, M. **How microcontrollers work**. Disponível em: <http://www.howstuffworks.com/microcontroller.htm>. Acesso em: 26 out. 2012.

BURNSIDE, K. **The History of Embedded Systems**. Disponível em: <http://www.ehow.com/info_12030725_history-embedded-systems.html>. Acesso em: 30 out. 2012.

LABSPACE **Computers: bits & bytes**. Seção 4.2. Disponível em: <http://labspace.open.ac.uk/mod/oucontent/view.php?id=426285§ion=1.4.2>. Acesso em: 25 out. 2012.

LAKATOS, E. M.; MARCONI, M. de A. **Técnicas de pesquisa**. São Paulo: Atlas, 1996.

MIKROELETRONIKA **Creating the first project in mikroC PRO for ARM**. Disponível em: <http://www.mikroe.com/downloads/get/1767/ctfp_mikroc_pro_for_arm.pdf>. Acesso em: 15 nov. 2012.

NATURAL INSTITUTE OF STANDARDS TECHNOLOGY **Pervasive Computing Program**. Disponível em: < http://www.itl.nist.gov/pervasivecomputing.html>. Acesso em: 30 out. 2012.

SIFAKIS, J. **A Brief History of Informatics and Embedded Systems**. Disponível em: <http://www.embedded-systems-portal.com/CTB/History,-8.html/>. Acesso em: 5 jun. 2012.

TYSON, J. **How Video Game Systems Work**. Disponível em: < http://electronics.howstuffworks.com/video-game3.htm>. Acesso em: 26 out. 2012.

WEBOPEDIA **Mobile Operating System**. Disponível em: <http://www.webopedia.com/TERM/M/mobile_operating_system.html>. Acesso em: 26 out. 2012.

Thomas Tadeu Gallassi

Graduando como Bacharel em Análise de Sistemas e Tecnologia da Informação na Faculdade de Tecnologia de Americana,tendo realizado uma iniciação científica sob a mesma instituição com o título de "Um Estudo Exploratório Sobre Sistemas Operacionais Embarcados". Participou também da XVII Maratona de Programação, atingindo o 35º lugar na fase nacional.
Contato: thomas_tadeu_gallassi@msn.com
Fonte: CNPQ – Currículo Lattes

Prof. Luiz Eduardo Galvão Martins

Professor Adjunto do Instituto de Ciência e Tecnologia da Universidade Federal de São Paulo (UNIFESP), lotado no “campus” de São José dos Campos. Tem experiência na área de Engenharia de Software, com ênfase em Engenharia de Requisitos, atuando principalmente nos seguintes temas de pesquisa e desenvolvimento: Engenharia de Software para Sistemas Embarcados, Sistemas Ciberfísicos, Sistemas Robóticos e Sistemas Adaptativos.
Contato: martinsleg@hotmail.com
Fonte: CNPQ – Currículo Lattes