CENTRO PAULA SOUZA

GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso de Bacharelado em Análise de Sistemas e Tecnologia da Informação

ANÁLISE SOBRE O DESENVOLVIMENTO DE APLICAÇÕES EMBARCADAS UTILIZANDO SISTEMAS OPERACIONAIS

THOMAS TADEU GALLASSI

Americana, SP
2012

CENTRO PAULA SOUZA GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso de Bacharelado em Análise de Sistemas e Tecnologia da Informação

ANÁLISE SOBRE O DESENVOLVIMENTO DE APLICAÇÕES EMBARCADAS UTILIZANDO SISTEMAS OPERACIONAIS

THOMAS TADEU GALLASSI

thomas_tadeu_gallassi@msn.com

Monografia de conclusão apresentada à Faculdade de Tecnologia de Americana, como parte dos requisitos para a obtenção do título de Bacharel em Análise de Sistemas e Tecnologia da Informação.

Área: Computação Ubíqua

Americana, SP
2012

BANCA EXAMINADORA

Professor José Luiz Zem
(Orientador)

Professor Clerivaldo José Roccia
(Presidente)

Professor Rogério Nunes de Freitas
(Convidado)

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Jose Fernandes Gallassi e Sueli Guerreiro Gallassi, os quais sempre me incentivaram e apoiaram em meus estudos e me educaram, ensinaram e aconselharam durante minha vida, além de ter provido por mim em momentos de dificuldades.

Agradeço também aos meus irmãos, Diego Hernandes Gallassi e Ricardo Gallassi, os quais foram decisivos para a minha escolha em seguir a área de Tecnologia da Informação como profissão.

Agradeço ainda aos meus diversos amigos, que aturam a mim e a meus defeitos sem reclamar.

Por último, agradeço a todos os professores que já lecionaram para mim durante minha vida, pois acredito que aprendi algo com cada um, mesmo com aqueles com os quais entrei em conflito direto ou indireto.

DEDICATÓRIA

Dedico este trabalho a Deus e aos meus pais, pelo apoio e paciência, pois sem eles eu nunca chegaria onde estou.

RESUMO

Aplicações embarcadas, sistemas embarcados ou sistemas embutidos possuem nomenclatura vasta e referem-se ao uso de pequenos sistemas computacionais e programas dentro de outros produtos de forma sutil e por vezes imperceptível, a fim de melhorar funcionalidades existentes ou oferecer novas.

Estão presentes em praticamente todas as áreas de atuação humana, e estima-se que mais de 95% dos sistemas computacionais produzidos atualmente são sistemas embarcados, o que demonstra sua importância para o mercado e para a tecnologia moderna. Ainda assim, seus projetos são restritos fisicamente, o que acarreta em restrições de consumo de energia, potencia de processamento e disponibilidade de memória, além das exigências de velocidade, segurança e confiabilidade serem altas devido às exigências do mercado e as evoluções tecnológicas lhes tornarem obsoletos rapidamente.

Para melhorar essa dinâmica, algumas empresas têm adotado sistemas operacionais embarcados, que ajudam na padronização dos projetos e aumentam a vida útil dos produtos, mas também requerem mais recursos computacionais, o que acarreta em maiores custos com dispositivos físicos e pode não compensar lucrativamente às empresas em certos casos.

Por outro lado, trazem características normalmente ausentes mas úteis em aplicações embarcadas, como a concorrência entre processos e a política de prioridades. Isso facilita a programação de aplicações complexas.

Complementarmente, um sistema operacional embarcado é complexo, e suas implicações devem sempre ser estudadas antes da implantação em um projeto, caso contrário, o sistema operacional pode ser mal implantado, com consequências como desperdício de recursos físicos, financeiros e computacionais como velocidade de processamento e disponibilidade de memória.

Palavras Chave: sistema embarcado; sistema operacional; sistema operacional embarcado.

ABSTRACT

Embedded applications or embedded systems possess a wide nomenclature, which refer to the use of little computational systems and programs subtly and sometimes imperceptibly inside other products to improve existing features or add new ones.

They are present in almost all human activity areas, and it's estimated that more than 95% of computational systems presently produced are embedded applications, which demonstrates its importance to the modern market and technology. Still, the projects have physical limitations which leads to restrictions for energy consumption, processing power and available memory, beyond the high speed, security and reliability requirements made by the market and the technological evolutions making them obsolete quickly.

To improve this dynamic, some companies have adopted embedded operating systems, which help in project standardization and increase the shelf life of products, but also require more computational resources, resulting in higher costs with physical devices and which might not profitably compensate in certain cases.

Moreover, they bring features normally absent but useful in embedded applications, such as competition between processes and the priorities policy. This facilitates the programming of complex applications.

In addition, an embedded operating system is complex, and its implications should always be considered before deployment on a project, otherwise the operating system may be poorly deployed, with consequences such as wasting physical, financial and computational resources such as processing speed and memory availability.

Keywords: *embedded system; operating system; embedded operating system.*

SUMÁRIO

INTRODUÇÃO **1**

1 LEVANTAMENTO BIBLIOGRÁFICO **5**

1.1 SISTEMAS COMPUTACIONAIS 5

1.1.1 PROCESSADORES **7**

1.1.2 MEMÓRIA **10**

1.1.3 DISPOSITIVOS DE ENTRADA E SAÍDA DE DADOS **12**

1.2 SISTEMAS OPERACIONAIS 13

1.2.1 CONCEITOS DE SISTEMAS OPERACIONAIS **14**

1.2.1.1 PROCESSO **14**

1.2.1.2 DEADLOCK **15**

1.2.1.3 GERENCIAMENTO DE MEMÓRIA **16**

1.2.1.4 ENTRADA E SAÍDA **16**

1.2.1.5 ARQUIVOS **17**

1.2.1.6 SEGURANÇA **18**

1.2.1.7 INTERPRETADOR DE COMANDOS **18**

1.2.2 TIPOS DE SISTEMAS OPERACIONAIS **19**

1.2.2.1 COMPUTADORES DE GRANDE PORTE **19**

1.2.2.2 SERVIDORES **19**

1.2.2.3 PARALELOS OU MULTINÚCLEOS **20**

1.2.2.4 COMPUTADORES PESSOAIS **20**

1.2.2.5 TEMPO REAL **20**

1.2.2.6 EMBARCADOS **21**

1.2.2.7 DISPOSITIVOS MÓVEIS **21**

1.2.2.8 VIDEOGAMES **22**

1.2.2.9 CARTÕES INTELIGENTES **22**

1.3 SISTEMAS EMBARCADOS 22

1.3.1 HISTÓRIA DE SISTEMAS EMBARCADOS **22**

1.3.2 CARACTERÍSTICAS DE SISTEMAS EMBARCADOS **24**

1.3.3 DESENVOLVIMENTO DE SISTEMAS EMBARCADOS **25**

1.3.4 ARQUITETURA DE SISTEMAS EMBARCADOS **31**

1.3.5 REQUISITOS DE SISTEMAS EMBARCADOS 36
1.4 SISTEMAS OPERACIONAIS EMBARCADOS 38
1.4.1 SUAS CARACTERÍSTICAS 38
1.4.2 SISTEMAS ATUALMENTE DISPONÍVEIS 40
2 DESENVOLVIMENTO 44
2.1 FERRAMENTAS UTILIZADAS E ESTUDADAS 44
2.1.1 PROTEUS **... 44**
2.1.1.1 ISIS **.. 44**
2.1.1.1.1 ENERGIA 45
2.1.1.1.2 RESISTORES 46
2.1.1.1.3 BOTÕES 46
2.1.1.1.4 INTERRUPTORES 46
2.1.1.1.5 PORTAS LÓGICAS 47
2.1.1.1.6 LÂMPADAS E *LED*S 47
***2.1.1.1.7 DISPLAY*S** **.. 47**
2.1.1.1.8 SENSORES DE TEMPERATURA 47
2.1.1.1.9 MICROCONTROLADORES 48
2.1.2 MIKROC **... 49**
2.1.3 FREERTOS **.. 51**
2.1.3.1 *TASK*S (TAREFAS) 53
2.1.4 OPEN WATCOM **... 57**
2.1.5 MPLAB **.. 58**
2.2 EXPERIMENTOS REALIZADOS 59
2.2.1 EXPERIMENTOS SEM SISTEMA OPERACIONAL EMBARCADO ... 60
2.2.1.1 TESTE DE MICROCONTROLADOR 60
2.2.1.1.1 CODIGO DO PROGRAMA ORIGINAL 61
2.2.1.1.2 PROPOSTAS DE MELHORIAS 65
2.2.1.2 SENSOR DE TEMPERATURA 67
2.2.1.2.1 CÓDIGO DO PROGRAMA ORIGINAL 68
2.2.1.2.2 PROPOSTAS DE MELHORIAS 73
2.2.1.3 SENSOR DE TEMPERATURA DUPLO 73
2.2.1.3.1 CÓDIGO DO PROGRAMA ORIGINAL 74
2.2.1.3.2 PROPOSTAS DE MELHORIAS 76

2.2.2 EXPERIMENTOS COM SISTEMA OPERACIONAL EMBARCADO...83
2.2.2.1 *LED*s Piscantes...83
3 CONSIDERAÇÕES FINAIS...87
3.1 CONCLUSÃO ...88
3.2 PROPOSTAS DE TRABALHOS FUTUROS ...90
REFERÊNCIAS BIBLIOGRÁFICAS...92
BIBLIOGRAFIA ...94

ÍNDICE DE FIGURAS

2.1 – MICROCOMPUTADOR SIMPLES..7
2.2 – *PIPELINE* COM TRÊS ESTÁGIOS..9
2.3 – PROCESSADOR SUPERESCALAR..10
2.4 – HIERARQUIA DE DIRETÓRIOS 18
2.5 – JANELAS DE TEMPO E RETORNO FINANCEIRO 27
2.6 – METODOLOGIA DE PROJETO EMBARCADO..28
2.7 – COMPARAÇÃO ENTRE PROJETO TRADICIONAL E DE REUSO 30
2.8 – MODELOS DE *NOC*S 36
3.1 – TELA DE PROJETO NO *ISIS*...45
3.2 – EXEMPLOS DE COMPONENTES NO *ISIS* 49
3.3 – TELA DE UM PROJETO NO *MIKROC*..50
3.4 – TELA DE UM PROJETO NO *OPEN WATCOM* 58
3.5 – *IDE* DO *MPLAB* 59
3.6 – PROJETO DE TESTE DE *DISPLAY*S E MICROCONTROLADOR 60
3.7 – SISTEMA EMBARCADO COM SENSOR DE TEMPERATURA 68
3.9 – SISTEMA EMBARCADO COM DOIS SENSORES DE TEMPERATURA.......73
3.9 – SISTEMA COM DOIS SENSORES DE TEMPERATURA APRIMORADO 76
3.10 – PROJETO DE *LED*S PISCANTES 83

ÍNDICE DE TABELAS

2.1 – CAMADAS DE UM SISTEMA COMPUTACIONAL..5
2.2 – REQUERIMENTOS DE SISTEMAS OPERACIONAIS EMBARCADOS.........43

ÍNDICE DE QUADROS

3.1 – EXEMPLO DE UTILIZAÇÃO DE *TASK*S 56

3.2 – CÓDIGO DE TESTE DE UM MICROCONTROLADOR E *DISPLAY*S.............61

3.3 – CÓDIGO DE TESTE APRIMORADO..65

3.4 – CÓDIGO DE SISTEMA EMBARCADO DE SENSOR DE TEMPERATURA...69

3.5 – CÓDIGO PARA DOIS SENSORES DE TEMPERATURA E *LCD*s 74

3.6 – CÓDIGO PARA DOIS SENSORES DE TEMPERATURA APRIMORADO.....77

3.7 – *LED*S PISCANTES SEM SISTEMA OPERACIONAL EMBARCADO.............84

3.8 – *LED*S PISCANTES COM SISTEMA OPERACIONAL EMBARCADO 85

INTRODUÇÃO

Antes de falar sobre sistemas embarcados, é necessário conhecer um pouco sobre sistemas computacionais como um todo. Os sistemas computacionais são compostos por qualquer conjunto de *hardware* e *software*. Comumente tais sistemas utilizam um programa, denominado de sistema operacional, para administrar e proteger todo o conjunto e para facilitar a interação dos usuários com o mesmo (TANENBAUM, 2003).

Sistemas embarcados, sistemas embutidos ou aplicações embarcadas, por sua vez, possuem nomenclatura vasta e referem-se ao uso de pequenos sistemas computacionais e programas dentro de outros produtos de forma sutil e por vezes imperceptível, a fim de melhorar funcionalidades existentes ou oferecer novas. Estão presentes em praticamente todas as áreas de atuação humana, incluindo sistemas de automação residencial, industrial e bancária, eletrodomésticos, eletroeletrônicos, equipamentos médicos, veículos e dispositivos eletrônicos móveis como celulares (CARRO, 2003).

Apesar de muitos desconhecerem sua existência, estima-se que mais de 95% dos sistemas computacionais produzidos por ano são sistemas embarcados, o que demonstra sua importância no mundo tecnológico e econômico atual e até no cotidiano das pessoas (SIFAKIS, 2012).

Aplicações embarcadas, apesar de serem fisicamente limitadas e, consecutivamente, dispor de poucos recursos computacionais e de pouca energia disponível, possuem funções específicas e necessitam ser rápidas, seguras e confiáveis (TANENBAUM, 2003). Porém, as empresas que desenvolvem tais aplicações disponibilizam poucas verbas e pouco tempo de desenvolvimento para seu projeto, já que o retorno não costuma ser alto e o produto possui tempo de vida curto devido à rápida evolução tecnológica e à dependência de outros produtos. Para resolver os problemas de custo e de tempo disponível, as empresas começaram a padronizar seus projetos de aplicações embarcadas, e para tanto, foi adotado o uso de sistemas operacionais (CARRO, 2003).

Um sistema operacional para uma aplicação embarcada é denominado um sistema operacional embarcado (TANENBAUM, 2003). Eles estão sendo integrados gradativamente nas empresas de desenvolvimento, mas são relativamente novos em comparação a outros sistemas operacionais, e seus benefícios e malefícios precisam ser mais estudados (CARRO, 2003).

É importante notar que aplicações embarcadas controlam muitas das áreas de atuação humana que são consideradas críticas, como transações bancárias, equipamentos médicos, sistemas de segurança e veículos. Qualquer alteração em uma função de um aplicativo em uma dessas áreas, mesmo que seja mínima, pode acarretar em péssimas conseqüências que muitas vezes não compensam os benefícios que a mudança trouxe. Para tanto, cada mudança deve ser planejada, e o uso de sistemas operacionais embarcados em tais áreas deve ser cuidadosamente estudado antes de aplicado (CARRO, 2003).

Como **justificativa** para este trabalho monográfico, temos que aplicações embarcadas estão cada vez mais presentes na vida das pessoas e das empresas, e por isso precisam ser rápidas, seguras e confiáveis, porém, por questões econômicas, dispõem de poucos recursos e pouco tempo de projeto, mas por outro lado, a utilização crescente de sistemas operacionais embarcados pode estar afetando essa dinâmica (CARRO, 2003).

Um **problema** é o fato de o efeito cascata que uma tecnologia incorporada em outra apresenta, o que torna interessante estudar tais sistemas operacionais, especialmente considerando a vasta gama de áreas em que as aplicações embarcadas estão inseridas, e sua proporção em relação ao desenvolvimento de sistemas computacionais como um todo (CARRO, 2003).

Assim sendo, as **perguntas** que se buscou responder incluem: Quais as vantagens e desvantagens do uso de sistemas operacionais embarcados no desenvolvimento de aplicações embarcadas? Quando e como as vantagens superam as desvantagens?

Com base nisso tudo, algumas **hipóteses** foram formuladas para serem estudadas e averiguadas neste trabalho:

a. A utilização de sistemas operacionais embarcados no desenvolvimento de aplicações embarcadas simplifica o processo de desenvolvimento;

b. A utilização de sistemas operacionais embarcados no desenvolvimento de aplicações embarcadas é viável às empresas apenas para projetos complexos;

c. A utilização de sistemas operacionais embarcados no desenvolvimento de aplicações embarcadas pode acelerar o processo de desenvolvimento;

d. Aplicações embarcadas que utilizam sistemas operacionais podem ser mais lentas do que as que não utilizam;

e. Aplicações embarcadas que utilizam sistemas operacionais podem ser mais inseguras do que as que não utilizam;

f. Aplicações embarcadas que utilizam sistemas operacionais podem ser menos confiáveis do que as que não utilizam.

O **objetivo geral** consistiu em estudar as vantagens e desvantagens do uso de um sistema operacional embarcado em aplicações embarcadas, especialmente em áreas críticas das aplicações embarcadas, como rapidez, segurança, confiabilidade, tempo de projeto e preço de desenvolvimento, para assim contribuir com o conteúdo disponível sobre o assunto e para que os leitores se conscientizem da importância das aplicações embarcadas na indústria, no comércio e nas residências, assim como os efeitos que os sistemas operacionais embarcados podem causar nessa tecnologia e em suas áreas de aplicação.

Entre os **objetivos específicos**, temos:

- Estudar as características e a história dos sistemas computacionais e dos sistemas operacionais para melhor entender as aplicações embarcadas e os sistemas operacionais embarcados

- Estudar as características e a história das aplicações embarcadas e depois dos sistemas operacionais embarcados, a fim de melhor entender como funcionam.

- Desenvolver ao menos um experimento composto pelo desenvolvimento de uma aplicação embarcada sem um sistema operacional embarcado, pelo desenvolvimento da mesma aplicação utilizando um sistema operacional embarcado e pela comparação de ambas as aplicações, a fim de obter mais dados sobre a utilização de sistemas operacionais embarcados.

- Averiguar, com base na revisão bibliográfica e nos experimentos realizados, os efeitos da utilização dos sistemas operacionais embarcados, focando nas áreas críticas de aplicações embarcadas e de seu desenvolvimento e também dispondo qualquer outro efeito positivo ou negativo encontrado.

Um trabalho monográfico é definido como um trabalho científico que reduz a abordagem a um único assunto e a um tratamento especificado (SEVERINO, 2007).

Durante o desenvolvimento do trabalho monográfico, foram utilizados dois **métodos de pesquisa**. O primeiro deles é a revisão bibliográfica, que é um método de pesquisa essencial para uma monografia, pois orienta à pesquisa de textos anteriores de outros autores sobre o mesmo assunto ou assuntos similares visando obter uma base de conhecimento. O outro método que foi utilizado é a pesquisa exploratória, o qual refere à utilização de estudos de casos ou experimentos não somente para averiguar e confirmar ou negar hipóteses, mas também para explorar situações, encontrando efeitos e recursos não previstos (LAKATOS, 1996).

O projeto foi dividido em três capítulos. O primeiro capítulo constitui o levantamento bibliográfico, e contêm conhecimento teórico necessário para entender o funcionamento de sistemas embarcados e de sistemas operacionais. O segundo capítulo constitui-se dos experimentos realizados a fim de averiguar as hipóteses. O último capítulo, desenvolvido com base nas informações adquiridas nos capítulos prévios, se reserva às considerações finais.

1 LEVANTAMENTO BIBLIOGRÁFICO

Nas subseções a seguir, serão abordados não somente os conceitos de aplicações embarcadas e sistemas operacionais embarcados, mas antes, para melhor entendimento, serão abordados também os conceitos de sistemas computacionais e sistemas operacionais.

1.1 SISTEMAS COMPUTACIONAIS

Sistemas embarcados, assim como mainframes, celulares, microcomputadores e cartões inteligentes são sistemas computacionais (TANENBAUM, 2003), e devido ao foco desse trabalho, é necessário entender um pouco sobre seu funcionamento.

Um sistema computacional moderno geralmente é composto de um ou mais processadores, memória volátil, memória estática, teclado, *mouse*, caixas de som, monitor, interfaces de rede, *scanner*, impressora, placa gráfica, câmera, microfone e diversos outros dispositivos de entrada e saída de dados (TANENBAUM, 2003). Como demonstrado na Tabela 2.1, um sistema computacional normal pode ser estudado e analisado camadas.

Tabela 2.1 – Camadas de um Sistema Computacional (TANENBAUM, 2003)

Programas de aplicação			
Compiladores	Editores	Interpretador de comandos	} Programas do sistema
Sistema operacional			
Linguagem de máquina			} Hardware
Microarquitetura			
Dispositivos físicos			

Em um sistema computacional, a camada superior é constituída dos programas e aplicações do usuário e os programas utilitários. Já segunda camada, por sua vez, é constituída de programas encapsulados e dependentes do sistema operacional, porém que não são necessários ou obrigatórios para seu funcionamento (TANENBAUM, 2003).

O sistema operacional em si constitui a terceira camada, a qual por oculta ao menos parcialmente a complexidade de camadas inferiores e proporciona aos usuários e programadores conjuntos de instruções mais convenientes e simplificadas. Desse modo, o sistema operacional, ao mesmo tempo que protege o *hardware*, é protegido por ele. O sistema operacional, em conjunto com os programas encapsulados, constituem um nível maior denominado de programas do sistema (TANENBAUM, 2003).

A quarta camada é a de linguagem de máquina, a qual movimenta dados através de instruções recebidas, além de controlar os dispositivos de entrada e saída de dados, lendo valores denominados de registradores de dispositivos. É uma camada com um conjunto amplo, complexo e detalhado de instruções específicas, as quais o sistema operacional é responsável por simplificar, agrupar e fornecer aos programadores e aos usuários (TANENBAUM, 2003).

A quinta e penúltima camada é a de micro-arquitetura, na qual os dispositivos físicos, que constituem a sexta camada, são agrupados em unidades funcionais. Os dispositivos físicos da sexta camada são, em sua maioria, conjuntos de pastilhas ou *chip*s de circuitos integrados, fontes de alimentação, fios, tubos de raios catódicos, cabos e outros dispositivos semelhantes. As três últimas camadas juntas constituem o nível de hardware de um sistema. É importante notar que os *chip*s de circuitos integrados podem ser considerados sistemas embarcados (TANENBAUM, 2003).

Um sistema computacional amplamente conhecido é o computador pessoal ou microcomputador. Conceitualmente, um microcomputador pode ser abstraído em um modelo similar ao da Figura 2.1, a qual demonstra a *CPU* (Unidade Central de Processamento), a memória e dispositivos de entrada e saída de dados ligados a um barramento, que proporciona a comunicação entre os dispositivos. Em modelos mais novos, porém, um sistema computacional ou mesmo um microcomputador pode possuir múltiplos barramentos e meios de comunicação entre os dispositivos (TANENBAUM, 2003).

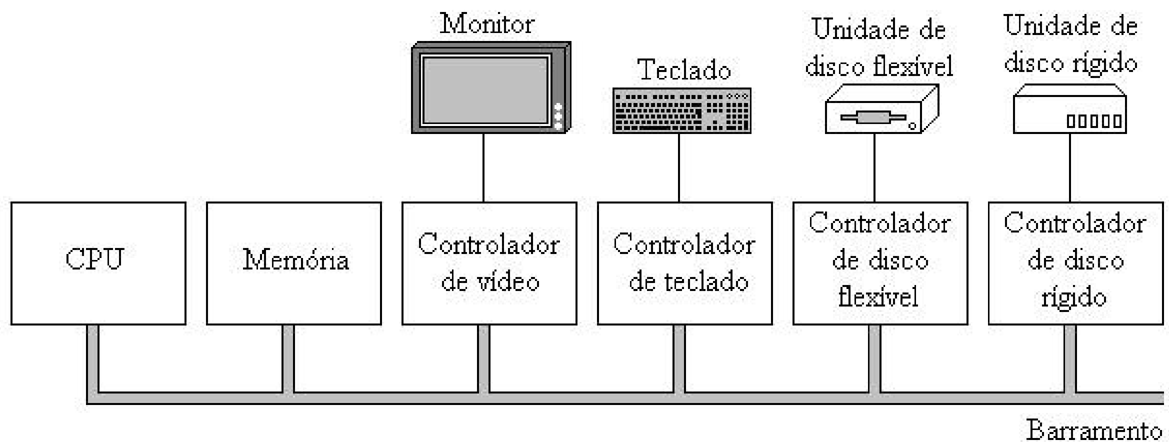

Figura 2.1 – Microcomputador Simples (TANENBAUM, 2003)

Nas subseções seguintes, serão explicados os conceitos de *CPU* (junto ao conceito de processador), memória e dispositivos de entrada e saída.

1.1.1 PROCESSADORES

Processador é o principal componente da maioria dos sistemas computacionais. Um processador possui uma ou mais *CPU*s, responsáveis por buscar, de uma em uma, as instruções armazenadas na memória, decodificar tais instruções e os operandos, executá-las e então prosseguir com as instruções na sequência (TANENBAUM, 2003).

É desse modo que os programas são executados em um sistema computacional, porém, cada *CPU* possui um conjunto limitado e específico de instruções que pode realizar, motivo pelo qual alguns computadores não possuem a capacidade de executar programas específicos (TANENBAUM, 2003).

Normalmente, cada *CPU* possui três unidades ligadas, necessárias para realizar tais ações. A *ALU* (Unidade Lógico-Aritmética) é quem faz o trabalho de fato, utilizando das quatro operações básicas (adição, subtração, multiplicação e divisão) e de métodos de comparação (maior, menor, igual etc.). A *control unit* (unidade de controle) é responsável por controlar o fluxo de dados entre a *ALU*, os diversos tipos de memória e os dispositivos de entrada e saída. Por último, há a memória para os registradores internos e os operandos, que é extremamente rápida, mas também cara e limitada, e serve para armazenar dados das instruções que estão sendo ou logo serão processadas, para armazenar variáveis importantes e para armazenar resultados temporários (LABSPACE, 2012).

O tempo para buscar instruções e operandos na memória é menor que o tempo necessário para executar tais instruções. Por esse motivo, que as CPUs possuem os registradores internos. Além dos registradores internos de propósitos gerais, há ainda registradores especiais, importantes e visíveis aos programadores (TANENBAUM, 2003). Entre eles temos:

- Contador de programa: guarda o endereço da próxima instrução a ser executada (essencialmente criando uma fila);

- Ponteiro de pilha: aponta para o cume ou topo da pilha de memória atual necessária pelo processo (pilhas contêm estruturas de cada procedimento chamado ainda não encerrado. Estruturas de pilhas contêm parâmetros de entrada e variáveis locais e temporárias que normalmente não são guardadas nos registradores, possivelmente mas não necessariamente pelo limite de tamanho dos registradores);

- PSW (*Program Status Word*): registrador que contêm os bits do código condicional, a fim de evitar que um processo ocupe espaço desnecessário na CPU quando depende de outros recursos, dados ou processos para continuar. Tais bits podem ser alterados pelo nível de prioridade de uma *CPU*, por funções ou instruções de comparações e por diversos *bits* de controle (TANENBAUM, 2003).

A maioria dos sistemas operacionais compartilham o tempo de *CPU*, especialmente os de menor porte, para que pareça ao usuário que diversos programas estão sendo executados simultaneamente (quando apenas alguns, ou no caso de haver apenas uma *CPU*, apenas um programa, está sendo executado por vez, mas em uma faixa de tempo geralmente menor que milissegundos) (TANENBAUM, 2003).

Para compartilhar o tempo de CPU, um sistema operacional geralmente interrompe a execução de um programa, salva e guarda todo o progresso e todos os registradores (para resgatar e continuar o programa posteriormente) e então busca e começa a executar outro programa (TANENBAUM, 2003).

As *CPU*s antigas utilizam um modelo simples, buscando, decodificando e executando uma instrução por vez, porém as *CPU*s mais modernas possuem recursos para trabalhar em várias instruções ao mesmo tempo, através de unidades separadas e especializadas para busca, decodificação e execução de instruções, entre outras. Desse modo, enquanto uma instrução é executada, outra instrução já está sendo decodificada e mais outra pesquisada na memória. Nota que isso cria uma fila, que é denominada *pipeline*. Um modelo de um *pipeline* com três estágios está exemplificado na Figura 2.2, mas é possível ter *pipeline*s bem maiores (TANENBAUM, 2003):

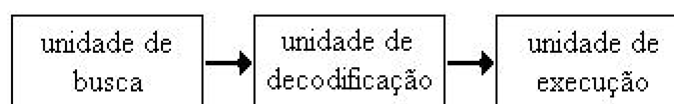

Figura 2.2 – *Pipeline* com Três Estágios (TANENBAUM, 2003)

Há ainda processadores superescalares, que possuem várias unidades de execução em uma mesma CPU, e consequentemente possuem capacidade de processamento paralelo. Neste modelo de processador, múltiplas instruções são pesquisadas e decodificadas, para então serem armazenadas em um buffer de instruções, até que possam ser executadas. Após uma unidade de execução ficar livre, caso haja instruções no buffer que tal unidade possa executar, a próxima instrução da fila é transferida do buffer e para a unidade executada (TANENBAUM, 2003).

A Figura 2.3 apresenta um modelo de um processador superescalar.

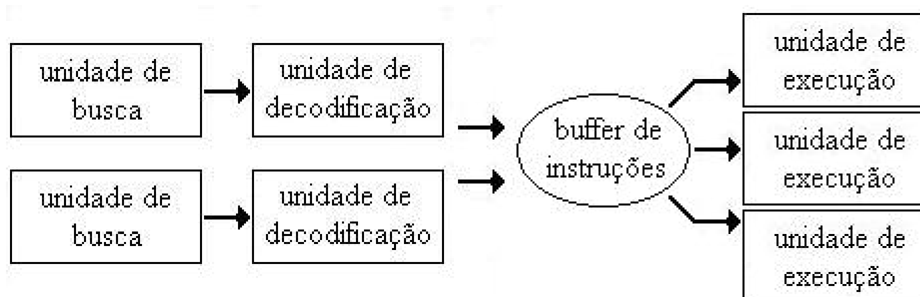

Figura 2.3 – Processador Superescalar (TANENBAUM, 2003)

Tanto em processadores superescalares, quanto em processadores multinúcleos (com várias CPUs), é possível que instruções advindas de um mesmo programa sejam executadas fora de ordem. Nesse caso, cabe ao hardware impedir que isso ocorra, o, caso aconteça, corrigir (TANENBAUM, 2003).

*CPU*s, em sua maioria, possuem dois modos principais de funcionamento: modo núcleo, no qual programas possuem acesso a todo o conjunto de instruções da *CPU* e podem usar todos os recursos e atributos do *hardware*, e modo usuário, em que o acesso e os recursos e atributos de *hardware* são restritos a apenas um subconjunto. Por questões de segurança, apenas o sistema operacional deve ser executado em modo núcleo, e caso algum programa precise de acesso à uma ou mais instruções ou atributos do *hardware* restritos, este pode fazer uma chamada ao sistema, que por sua vez alterna o modo usuário para modo núcleo e toma o controle para si. Quando as operações necessárias são completadas, o sistema operacional modifica a *CPU* novamente para o modo usuário e então retorna os resultados obtidos para o programa (TANENBAUM, 2003).

1.1.2 MEMÓRIA

A memória é o segundo principal componente de um sistema computacional.Idealmente, para que não haja gargalos, a memória deveria ser grande e possuir velocidade similar a da execução das instruções pelo processador (para que o mesmo não fosse atrasado), porém nenhuma tecnologia atual atende a

tais requisitos. Não o bastante, as memórias mais rápidas são não somente limitadas, mas extremamente caras (TANENBAUM, 2003).

Para diminuir o impacto de tal problema, a memória dos sistemas computacionais é construída baseada em um modelo hierárquico de camadas, onde as memórias mais rápidas, porém mais caras e mais limitadas estão mais próximas (referindo-se a acesso) dos processadores, enquanto as que são mais lentas e baratas, porém possuem poder de armazenamento muito maior e estático, ficam mais longe (TANENBAUM, 2003). Seguindo a ordem hierárquica padrão de proximidade ao processador, segundo Tanenbaum (2003), as camadas são:

- Registradores Internos das *CPU*s: feitas do mesmo material das *CPU*s, são tão rápidas quanto as mesmas. Normalmente possuem menos de 1 KB de espaço;

- Memória *Cache*: sendo controlada principalmente pelo *hardware*, ela guarda instruções que estão sendo frequentemente usadas pelas *CPU*s. Um mesmo computador pode possuir múltiplos níveis de memória *cache*;

- *RAM* (Memória de Acesso Aleatório): considerada como a memória principal de um computador. As requisições de busca que não são atendidas pela memória *cache* são então procuradas na memória *RAM*;

- Disco Rígido: Por ser um dispositivo mecânico é também um dispositivo de memória não volátil, o que significa que os dados continuam guardados mesmo se o computador for desligado;

- Dispositivos Externos: Também dispositivos de mecânicos (e consequentemente de memória não volátil) que são comumente usados para *backup* (cópia de segurança) e transporte físico de dados (TANENBAUM, 2003). Alguns dispositivos externos comuns incluem fita magnética, DVDs e *pendrives*.

Existem ainda memórias especiais, como a *ROM* (Memória Somente de Leitura), um tipo de memória não volátil que vem pré-programada de fábrica, é veloz e barata, mas que não pode ser alterada. Algumas variações da *ROM* podem ser alteradas, mas são consideravelmente difíceis e envolvem recursos não convencionais, como a EEPROM (Memória Somente de Leitura Eletricamente Apagável e Programável), o que permite a correção de erros da programação de fábrica. A *ROM* e suas variações são comumente usadas para carregar e iniciar um computador ou dispositivo a partir de configurações básicas (TANENBAUM, 2003).

Outro tipo de memória especial é a *CMOS* (Semi-Condutor Complementar de Metal Óxido), um tipo de memória volátil alimentada por uma bateria, geralmente utilizada para guardar configurações básicas de um computador como parâmetros de configuração do computador e dispositivos de hardware, data e hora do sistema e a localização (dispositivo) do sistema operacional (para iniciá-lo). Apesar de ser uma memória volátil, seu consumo de energia é baixo, e sua bateria pode durar por diversos anos. Ela possui configurações de base pré-programadas, o que quer dizer que se a bateria falhar ou for retirada, suas configurações são reiniciadas (TANENBAUM, 2003).

1.1.3 DISPOSITIVOS DE ENTRADA E SAÍDA DE DADOS

Dispositivos de entrada e saída de dados são fundamentais para qualquer sistema, pois sem eles, não haveria de quem obter dados nem para quem passar os resultados obtidos. Normalmente interagem intensamente com o *hardware* e com o sistema operacional e são constituídos de duas partes: o dispositivo propriamente dito e o controlador. Os dispositivos possuem interfaces simples e similares, pois não possuem ações muito diferente uns do outros (TANENBAUM, 2003).

O controlador, por outro lado, é um *chip* ou conjunto de *chip*s em uma placa (o que pode caracterizar como um sistema embarcado) que controla fisicamente o dispositivo. É ele quem recebe os comandos do sistema operacional para interagir com o dispositivo. O controle real de um dispositivo de entrada e/ou saída pode ser complexo, e é tarefa do controlador oferecer uma interface de comunicação simplificada ao sistema operacional (TANENBAUM, 2003).

Ainda assim, os controladores são variados e consideravelmente diferentes, e necessitam de programas para interagirem propriamente com um sistema operacional. Um programa de compatibilidade entre um sistema operacional e um controlador é denominado de *driver*. Cada fabricante deve fornecer um *driver* de cada dispositivo para cada sistema operacional suportado (TANENBAUM, 2003).

Para a maioria dos *driver*s funcionar, eles devem ser instalados junto ao sistema operacional e devem ser utilizados no modo núcleo. s controladores possuem um pequeno conjunto de registradores para comunicação. Para ativar um desses registradores, um *driver* primeiro recebe um comando do sistema operacional e traduz em valores apropriados para serem utilizados pelos registradores do controlador (TANENBAUM, 2003).

Um dispositivo pode ser classificado como dispositivo de entrada (o qual apenas envia dados), dispositivo de saída (o qual apenas recebe dados para armazenar ou demonstrar) ou dispositivo de entrada e saída (que pode tanto fornecer quanto receber dados) (TANENBAUM, 2003).

1.2 SISTEMAS OPERACIONAIS

Considerando a diversidade de recursos de um sistema computacional, o desenvolvimento de um programa ou aplicativo que gerencie, controle e utilize corretamente e de maneira otimizada todos esses componentes e perceba quando estão presentes e disponíveis para uso é algo extremamente difícil. Para tanto, foram foi desenvolvido um *software* denominado sistema operacional, o qual serve como um intermediador e facilitador, cujo trabalho é proteger e gerenciar os recursos, componentes e dispositivos de *hardware*, ao mesmo tempo que fornece interfaces simplificadas de acesso para que os programas dos usuários possam acessá-los corretamente. Outras funções importantes de um sistema operacional é simplificar a interação entre usuário e máquina, e estender os conjuntos de instruções do sistema computacional (TANENBAUM, 2003).

Assim sendo, um sistema operacional precisa ter grande conhecimento do *hardware* do sistema computacional ao qual está instalado e é executado, pois está intimamente ligado ao mesmo (TANENBAUM, 2003).

1.2.1 CONCEITOS DE SISTEMAS OPERACIONAIS

Para entender como funcionam e para poder diferenciar os tipos de sistemas operacionais, inclusive os sistemas operacionais embarcados, é essencial conhecer os conceitos de sistemas operacionais como um todo (TANENBAUM, 2003).

1.2.1.1 PROCESSO

Um processo poderia ser definido, basicamente, como um programa que está em execução. Um processo possui espaço de endereçamento, uma lista de localizações na memória, contendo os espaços o qual o processo pode alterar (ler e escrever), e um pequeno conjunto de registradores (inclusos nesse conjunto, temos ponteiro de pilha, contador de programas, alguns registradores de *hardware* e as demais informações necessárias para que o programa seja executado). O espaço de endereçamento é composto pelo programa executável, seus dados e sua pilha (TANENBAUM, 2003).

Nos sistemas operacionais de tempo compartilhado, processos em execução são periodicamente interrompidos e salvos (comumente numa tabela de processos) junto com todas as informações e o progresso até o momento, para liberar a *CPU* para outro processo enquanto entra em uma fila e ser retomado posteriormente, até que o processo seja encerrado (TANENBAUM, 2003).

As principais chamadas ao sistema relacionadas ao gerenciamento de processos são as que lidam com término e criação de processos, requisição de memória extra, liberação de memória e sobreposição de um processo (após o término) (TANENBAUM, 2003).

Um processo pode criar outros processos, denominados de processos filhos (enquanto o criador é denominado de processo pai), o que permite montar uma estrutura hierárquica de árvores desde o sistema operacional (TANENBAUM, 2003).

Quando processos necessitam cooperar para realizar uma tarefa, se comunicando frequentemente e sincronizando suas atividades, ocorre o que é chamado de comunicação interprocessos. A comunicação é feita através de mensagens através de um sistema operacional ou, mais comumente, através de uma rede. As mensagens são tratadas pelos programas, que usualmente interrompem temporariamente o que estavam fazendo anteriormente para tratar as mensagens. Quando um processo fica a espera de uma resposta de outro processo que não está funcionando corretamente ou que por algum motivo, não existe mais, ele é denominado de processo zumbi, pois consome tempo de *CPU* e não realiza nada de útil para nenhum usuário ou sistema (TANENBAUM, 2003).

Em sistemas de *threads* (linhas) de processos, processos possuem várias *threads* que compartilham informações sem copiá-las, o que torna processos interdependentes e diminui a formação de processos zumbis, especialmente em hierarquias de pais e filhos, pois se um pai morre, todos os filhos são excluídos automaticamente (TANENBAUM, 2003).

Todo processo possui, entre seu conjunto de registradores, uma *UID* (Identificação de Usuário), responsável por identificar o usuário sob qual o processo foi originalmente criado. É muito utilizado para controlar permissões e proteger processos de outros usuários de maior nível de serem alterados ou finalizados (porém há usuários com permissões especiais para ultrapassar essa proteção, como os administradores) (TANENBAUM, 2003).

1.2.1.2 DEADLOCK

Um *deadlock* (congestionamento) é um evento no qual um ou mais processos estão em uma situação em que não conseguem sair normalmente. Uma situação comum de *deadlock* ocorre quando um processo está utilizando e bloqueando um recurso de *hardware* e esperando para utilizar outro recurso de *hardware*, que por

sua vez está sendo bloqueado e utilizando por um segundo processo o qual está esperando para utilizar o *hardware* bloqueado pelo primeiro processo. Em uma situação assim, um desses processos precisaria da intervenção de um terceiro processo ou de um usuário. Uma solução seria destruir um dos processos, liberando um dos *hardware*s, mas isso poderia ocasionar perda de dados (TANENBAUM, 2003).

1.2.1.3 GERENCIAMENTO DE MEMÓRIA

Computadores necessitam de uma memória principal, utilizada para guardar os programas em execução e a executar. Computadores mais antigos e simples permitem que apenas um programa por vez ocupe a memória principal, de modo que para um segundo programa ocupar a memória, o primeiro deve sair (geralmente para uma CPU) (TANENBAUM, 2003).

Sistemas operacionais mais modernos e sofisticados permitem que diversos processos habitem a memória simultaneamente, através de particionamento da memória e sistemas de filas. Tais sistemas operacionais devem oferecer mecanismos de segurança para que programas que não são relacionados não possam acessar, excluir e principalmente modificar informações e dados pertinentes a outros processos (TANENBAUM, 2003).

Outro fator importante do gerenciamento de memória é gerenciamento do espaço de endereçamento de processos. Em computadores mais antigos, programas maiores que a memória disponível poderiam causar diversas complicações, mas em computadores mais novos, é usada a técnica de memória virtual, onde programas que excedem o espaço disponível na memória são armazenados no disco rígido e seus dados vão sendo resgatados conforme o necessário, porém com um alto custo de velocidade (TANENBAUM, 2003).

1.2.1.4 ENTRADA E SAÍDA

Todos os sistemas computacionais possuem dispositivos de entrada e saída de dados, também conhecidos como periféricos, uma vez que, sem eles, usuários

não poderiam fazer requisições e nem verificar os resultados obtidos (TANENBAUM, 2003).

Existem diversos tipos de dispositivos de entrada de dados (como mouse, teclado, microfone, câmera, *scanner*, etc.), de saída de dados (como monitor, caixa de som, impressora etc.) e de entrada e saída de dados (como *pendrive*, monitor *touchscreen*, placa de rede etc.). É responsabilidade do sistema operacional gerenciar e proteger esses dispositivos (TANENBAUM, 2003).

Consequentemente, sistemas operacionais possuem subsistemas de entrada e saída, para gerenciamento dos dispositivos. Alguns dos programas de entrada e saída de dados são independentes de dispositivos (aplicam-se igualmente bem a todos ou a maioria de dispositivos de um mesmo tipo ou grupo), enquanto outros programas, como os *driver*s, são específicos para cada dispositivo em cada sistema operacional (TANENBAUM, 2003).

1.2.1.5 ARQUIVOS

Uma das funções dos sistemas operacionais é a simplificação e a proteção do *hardware* para o usuário, o que inclui dispositivos de armazenamento de dados. Nesse contexto, arquivos e sistemas de arquivos são fundamentais para a maioria dos sistemas operacionais (TANENBAUM, 2003).

Para criar, renomear, ler, escrever e remover arquivos, são necessárias chamadas ao sistema, e para realizar qualquer uma dessas operações, o arquivo deve ser primeiro localizado no dispositivo de armazenamento e depois aberto, momento no qual as permissões de usuário são verificadas (TANENBAUM, 2003).

Para organizar e guardar arquivos de maneira menos confusa, a maioria dos sistemas operacionais oferece uma tecnologia de diretórios hierárquicos como um modo de agrupar arquivos. Para criar, renomear, abrir, alterar e remover diretórios e para movimentar arquivos entre diretórios também são necessárias chamadas ao sistema. Por ser um modelo hierárquico, um diretório pode ter entradas não somente

de arquivos, mas também de outros diretórios, o que permite a representação em um modelo de árvores (TANENBAUM, 2003), como exemplificado na Figura 2.4:

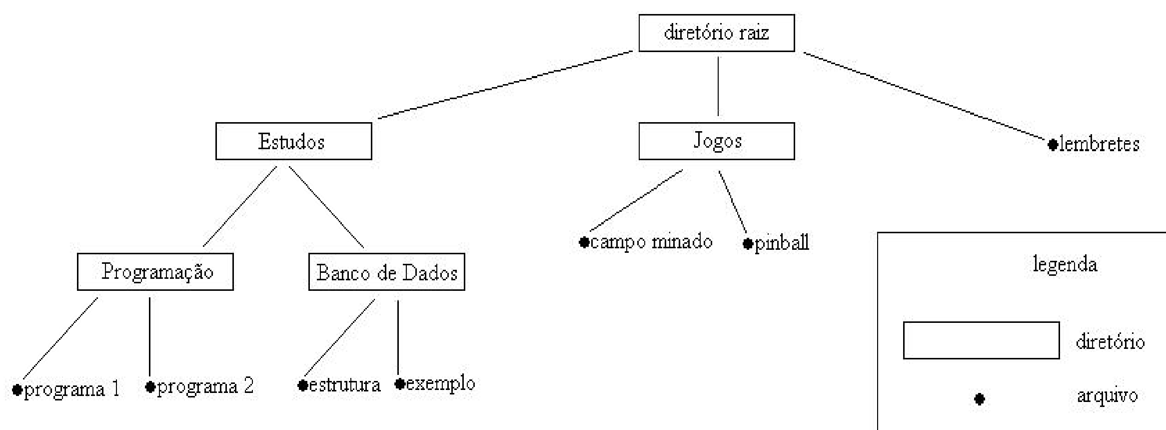

Figura 2.4 – Hierarquia de Diretórios (TANENBAUM, 2003)

1.2.1.6 SEGURANÇA

Computadores contêm muitas informações as quais os usuários desejam manter confidenciais ou restritas. É responsabilidade do sistema operacional o gerenciamento da segurança básica de tais informações, de modo que um usuário só possa acessar informações caso possua permissão, e também de modo que o sistema não seja invadido com facilidade (TANENBAUM, 2003).

1.2.1.7 INTERPRETADOR DE COMANDOS

Um sistema operacional funciona como um código que apenas executa as chamadas ao sistema, desse modo, compiladores, editores, interpretadores de comandos e outros programas que comumente vêm instalados junto ao sistema operacional, não fazem parte do sistema operacional em si, mesmo que tais programas sejam extremamente importantes e úteis (TANENBAUM, 2003).

Mesmo não sendo parte direta de um sistema operacional, interpretadores de comandos fazem uso extensivo dos aspectos do sistema operacional, e são um método de interface entre o usuário e o sistema (especialmente na falta de uma interface gráfica) (TANENBAUM, 2003).

1.2.2 TIPOS DE SISTEMAS OPERACIONAIS

Com a evolução, ramificação, diversificação e especialização dos sistemas computacionais, para atender aos novos requisitos, novos tipos de sistemas operacionais também foram sendo desenvolvidos, alguns dos quais serão tratados nas subseções a seguir

1.2.2.1 COMPUTADORES DE GRANDE PORTE

Computadores de grande porte ocupam salas inteiras e possuem ampla capacidade de armazenamento e de comunicação com dispositivos de entrada e saída. Seus sistemas operacionais são orientados à processamento simultâneo e seguro de *job*s (conjuntos de instruções), a maioria a qual necessita de muita interação com dispositivos de entrada e saída de dados (TANENBAUM, 2003). Normalmente, tais sistemas operacionais oferecem três tipos principais de serviços:

- Lote (*batch*): processa grande quantidade de *job*s em sequência, sem a necessidade de interações constantes ou da presença de um usuário;

- Processamento de transações: administra vastas quantidades de pequenas requisições;

- Tempo compartilhado: compartilha o tempo entre diferentes usuários, permitindo assim que muitos usuários remotos tenham acesso e executem suas requisições de modo simultâneo (TANENBAUM, 2003).

1.2.2.2 SERVIDORES

Servidores variam desde supercomputadores até microcomputadores bem planejados. São computadores consideravelmente potentes e geralmente destinados e especializados à uma ou mais funções específicas, normalmente interligados em uma rede com outros servidores para atingir uma gama mais completa de serviços (TANENBAUM, 2003).

Não é necessário que cada servidor seja especializado, podem haver servidores genéricos, que realizam todas as funções necessárias à rede, mas não é recomendado devido à alta quantidade de requisições que tais servidores recebem, especialmente em redes maiores (TANENBAUM, 2003).

Os sistemas operacionais destinados à servidores são especializados em servir múltiplos usuários simultaneamente através de uma rede, permitindo compartilhamento de *hardware* e *software* entre os mesmos (TANENBAUM, 2003).

1.2.2.3 PARALELOS OU MULTINÚCLEOS

Computadores que possuem processadores multinúcleos ou que possuem múltiplos processadores (computadores de multiprocessadores) precisam de sistemas operacionais diferenciados, com aspectos especiais de comunicação e de conectividade. Nada impede, porém, que sejam usados em computadores com apenas um núcleo de processamento (TANENBAUM, 2003).

1.2.2.4 COMPUTADORES PESSOAIS

Computadores pessoais são usados, em geral, por um único usuário por vez ou por um usuário principal. Sistemas operacionais para computadores pessoais são altamente configuráveis pelo usuário e possuem extenso suporte para processamento de planilhas e textos, meios de entretenimento (como músicas, jogos e vídeos) e acesso à Internet. Tais sistemas operacionais funcionam bem em computadores que possuem um único processador, mesmo que tal processador seja multinúcleos (TANENBAUM, 2003).

1.2.2.5 TEMPO REAL

Quando o tempo é um parâmetro fundamental, é comum a utilização de sistemas operacionais de tempo real. Nesse tipo de sistema operacional, é normal a existência de prazos rígidos de execução para certas tarefas. Quando é aceitável que esses prazos sejam ocasionalmente quebrados, há um sistema operacional é considerado como de tempo real não-crítico, mas caso contrário, é considerado como um sistema de tempo real crítico (TANENBAUM, 2003).

1.2.2.6 EMBARCADOS

Sistemas embarcados são fisicamente limitados e geralmente bem pequenos. Possuem restrições severas de tamanho físico, memória disponível, poder de processamento e energia elétrica, sendo comum a dependência de baterias. Como velocidade e segurança são comumente parâmetros fundamentais de suas aplicações, suas funções geralmente são otimizadas. Felizmente, o escopo de funções de um sistema embarcado costuma ser pequeno, já que um sistema embarcado é normalmente associado ou embutido em um outro produto a fim de melhorar ou estender suas funcionalidades, porém isso também quer dizer que o sistema embarcado necessita ser construído a um custo baixo, para manter o preço competitivo do produto. Como velocidade é um parâmetro importante, um sistema embarcado é, por vezes, também um sistema de tempo real (TANENBAUM, 2003).

Em sistemas embarcados, é comum o uso de microcontroladores, que desempenham não só funções de processadores, mas possuem memória própria e suporte à periféricos como dispositivos de entrada e saída de dados. Basicamente, são um computador (BRAIN, 2012).

Todo software, especialmente um sistema operacional, ocupa espaço na memória e gasta tempo de processamento, então a implementação de um sistema operacional embarcado, apesar de ajudar e padronizar o desenvolvimento de aplicações embarcadas, pode requerer um hardware ligeiramente mais potente e mais caro, o que significa que não é sempre vantajoso usar sistemas operacionais embarcados, especialmente para projetos menores (CARRO, 2003).

É importante notar que sistemas operacionais embarcados são bem mais utilizados para satisfazer necessidades de desenvolvimento do que necessidades de usuários (CARRO, 2003).

1.2.2.7 DISPOSITIVOS MÓVEIS

Comumente utilizada em *smartphones*, celulares e outros dispositivos móveis, seus sistemas operacionais se assemelham aos de computadores pessoais ao serem destinados a um usuário por vez, porém, assim como os sistemas

embarcados, são fisicamente limitados (WEBOPEDIA, 2012). Anteriormente a tais sistemas operacionais móveis, os celulares utilizavam sistemas operacionais embarcados robustos para seu funcionamento (ANTI ESSAYS, 2012).

1.2.2.8 VIDEOGAMES

Assim como os celulares, os videogames também começaram como sistemas embarcados robustos, utilizando dos sistemas operacionais embarcados (CARRO, 2012), mas hoje possuem seus próprios sistemas operacionais especializados, se assemelhando com sistemas operacionais de computadores pessoais (TYSON, 2012).

1.2.2.9 CARTÕES INTELIGENTES

Cartões inteligentes são, basicamente, cartões com sistemas embarcados, mas devido ao seu tamanho, as restrições que se aplicam a sistemas embarcados são muito mais severas. Geralmente, possuem apenas um *chip*, CPU ou microcontrolador e realizam apenas algumas funções (normalmente apenas uma). Ainda assim, é possível e até comum o uso de sistemas embarcados para cartões inteligentes, já que sua taxa de utilização costuma ser bem menor do que de um sistema embarcado. Alguns exemplos de cartões inteligentes são cartões de crédito e de débito bancário (TANENBAUM, 2003).

1.3 SISTEMAS EMBARCADOS

Sistemas embarcados são considerados uma tecnologia ubíqua e pervasiva (SIFAKIS, 2012), isso quer dizer que seus aparatos tendem a estar no cotidiano das pessoas, em todos os lugares, se comunicando constantemente (*NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY*, 2002).

1.3.1 HISTÓRIA DE SISTEMAS EMBARCADOS

Nas últimas décadas, houve um crescente interesse por tecnologia da informação, mas também houve pouca pesquisa voltada para a reengenharia de

produtos físicos, área responsável pela incorporação de sensores em microcontroladores complexos e em sistemas de software de produtos (tecnologia embarcada da informação). Porém, está dinâmica está mudando, e o foco em reengenharia de produtos físicos está crescendo (KONANA, 2007).

Apesar disso, a utilização de tecnologia embarcada em produtos físicos não é nova (SIFAKIS, 2012). O primeiro sistema embarcado surgiu em 1961, fabricado para um míssil militar norte-americano chamado de *Minuteman ICMB*. Tal sistema utilizava transistores discretos e portas lógicas, mas não possuía nenhum microprocessador, ao contrário do sistema de ar do avião a jato *F-14 Tomcat*, de 1970, que possuía oito processadores e dezenove *chip*s de memória (BURNSIDE, 2012).

Porém, os sistemas embarcados começaram a ser usados amplamente comercialmente em 1980, com a integração *SoC* (*System-on-a-Chip*), que combina microprocessador, memória, acesso a dispositivos de entrada e saída e funções de controle em um único *chip* (BURNSIDE, 2012).

Por terem restrições severas de tamanho e consecutivamente pouca disponibilidade de energia elétrica, seu desenvolvimento foi mais lento do que os outros sistemas computacionais desde então (BURNSIDE, 2012).

Porém, com os avanços tecnológicos, a necessidade de energia elétrica diminuiu drasticamente nos últimos anos, o que acoplado ao rápido desenvolvimento de videogames e dispositivos móveis impulsionou o mercado embarcado nos últimos anos (BURNSIDE, 2012).

Hoje, os sistemas embarcados podem ser encontrados em todas as áreas de atuação humana, como militar, corporativo, governamental, acadêmico, industrial, religioso, privado, entretenimento, segurança etc. (CARRO, 2003). Alguns exemplos de produtos que utilizam tecnologia embarcada incluem automóveis, aviões, eletrodomésticos, brinquedos, celulares, videogames, equipamentos hospitalares, máquinas industriais, câmeras fotográficas, câmeras de segurança, ventiladores e até artigos de roupas (como um sapato com ajuste automático de altura).

1.3.2 CARACTERÍSTICAS DE SISTEMAS EMBARCADOS

Sistemas Embarcados são sistemas computacionais fisicamente limitados, comumente com restrições de memória, tamanho, energia elétrica e potencia de processamento devido a esse limite. Geralmente possuem um número limitado, porém, bem específico de funções. São comumente associados ou mesmo embutidos em outros produtos, origem de sua denominação. Normalmente possuem também características de sistemas de tempo real, como rapidez e confiabilidade (TANENBAUM, 2003).

Sistemas embarcados podem, através da captura e processamento de eventos (geralmente através de sensores), identificar e até corrigir problemas e defeitos antes que aconteça uma falha em um produto. Tornam também fácil personalizar, operar, e diagnosticar produtos e serviços, além de ajudarem na disponibilização de informações de uso aos fabricantes, o que lhes permitem desenvolver e aprimorar novos produtos e serviços, utilizar processos de inovação e construir relações diretas com clientes, o que significa que tecnologia embarcada possui alto impacto e implicações importantes ao escopo das empresas, ao sistema de competição industrial e à aplicação dos negócios, mas também há preocupações com privacidade dos clientes (KONANA, 2007).

Sistemas embarcados permitem ainda a produtores e a prestadores de serviços a mapear o desempenho de um produto durante seu ciclo de vida, descobrindo e aprimorando seu valor de serviço e integrando operações com processos de negócio (KONANA, 2007).

Esse mapeamento disponibiliza também informações importantes, como estado de manutenção e problemas de qualidade de um produto, o que permite aprimorar futuros modelos do produto ou mesmo de outros produtos embarcados similares e também permite aos produtores a prover serviços novos ou existentes de forma proativa, fortalecendo as relações com os clientes (KONANA, 2007).

Tal mapeamento pode ser útil ainda em decisões de estocagem de produtos ou peças, especialmente quando há uma grande concentração de diferentes modelos de um mesmo produto em certas regiões (KONANA, 2007).

1.3.3 DESENVOLVIMENTO DE SISTEMAS EMBARCADOS

Se tratando de seu desenvolvimento, temos ainda restrições de custo e tempo e uma preocupação com portabilidade, possível acesso a rede e reaproveitamento de projetos anteriores, o que não pode comprometer seu desempenho final. Essas restrições de desenvolvimento se devem não somente ao fato de geralmente serem embutidos em outros produtos, mas também de que o preço do produto final não pode ser muito alterado e principalmente da rápida evolução da tecnologia computacional, que garante que os computadores se tornem desatualizados rapidamente e tenham vida curta (algo ainda mais acentuado aos sistemas embarcados devido aos seus limites físicos), como explica a lei de Moore (a cada 18 meses, dobra-se o poder de processamento, pois um processador têm a sua disposição o dobro de transistores) (CARRO, 2003). É previsto o término da Lei de Moore apenas em 2020 (SIFAKIS, 2012).

Empresas têm razão estratégica para utilizar tecnologia embarcada, já que tal tecnologia permite oportunidades de inovações de serviços, produtos e processos. Há três tipos principais de iniciativas para incorporar tecnologia embarcada em outros produtos, segundo Konana (2007). São elas:

- Enriquecimento: Os produtos são redesenhados com adição de tecnologia embarcada, de modo a prover maior qualidade, conveniência e desempenho ou novas funcionalidades sem alteração da funcionalidade básica do produto original. No enriquecimento, pela funcionalidade básica não ser alterada, ela continua independente do sistema embarcado e continua a funcionar mesmo que o sistema embarcado falhe por completo. Exemplo: Sapato com ajuste automático;

- Digitalização: Partes já existentes de um produto são substituídas ou modificadas e ajustadas, usualmente para prover resultados digitais. O produto ainda possuirá partes que são operadas manualmente, sem intervenção ou dependência de um sistema embarcado. Esse tipo de incorporação embarcada também pode ser usada para criar um novo

produto, utilizando tecnologia mista, ou pode ser usado para criar novos serviços. Exemplo: aparelho de monitoramente cardíaco;

- Substituição: Um produto é refeito e substituído por tecnologia embarcada ou é projetado a partir do zero. Exemplo: Máquinas fotográficas.

Por estarem comumente embutidos a outros produtos, é mais complicada a atualização do *software* de sistemas embarcados, especialmente quando o produto não possui uma natureza computacional (como, por exemplo, em um urso de pelúcia falante), seja para corrigir erros ou estender funcionalidades, o que quer dizer que o *software* precisa ser bem planejado antes de implantado (CARRO, 2003).

Atualmente, o custo de aparatos tecnológicos está diminuindo, o que por sua vez permite que o custo de fabricação de sistemas embarcados diminua e consequentemente adentrem em uma gama maior de produtos, se tornando cada vez mais presentes no cotidiano das pessoas (CARRO, 2003).

Apesar da necessidade de ser um projeto rápido e de baixo custo, um projeto de sistema embarcado (tanto em *hardware* quanto em *software*) é considerado complexo, pois aborda conceitos e temas pouco analisados pela computação de propósito geral (CARRO, 2003).

Devido a essas necessidades, um atraso em um projeto de sistema embarcado, mesmo que seja um atraso mínimo, pode significar grande perda de lucros ou até prejuízos à uma organização (CARRO, 2003).

Um desses atrasos é exemplificado na Figura 2.5.

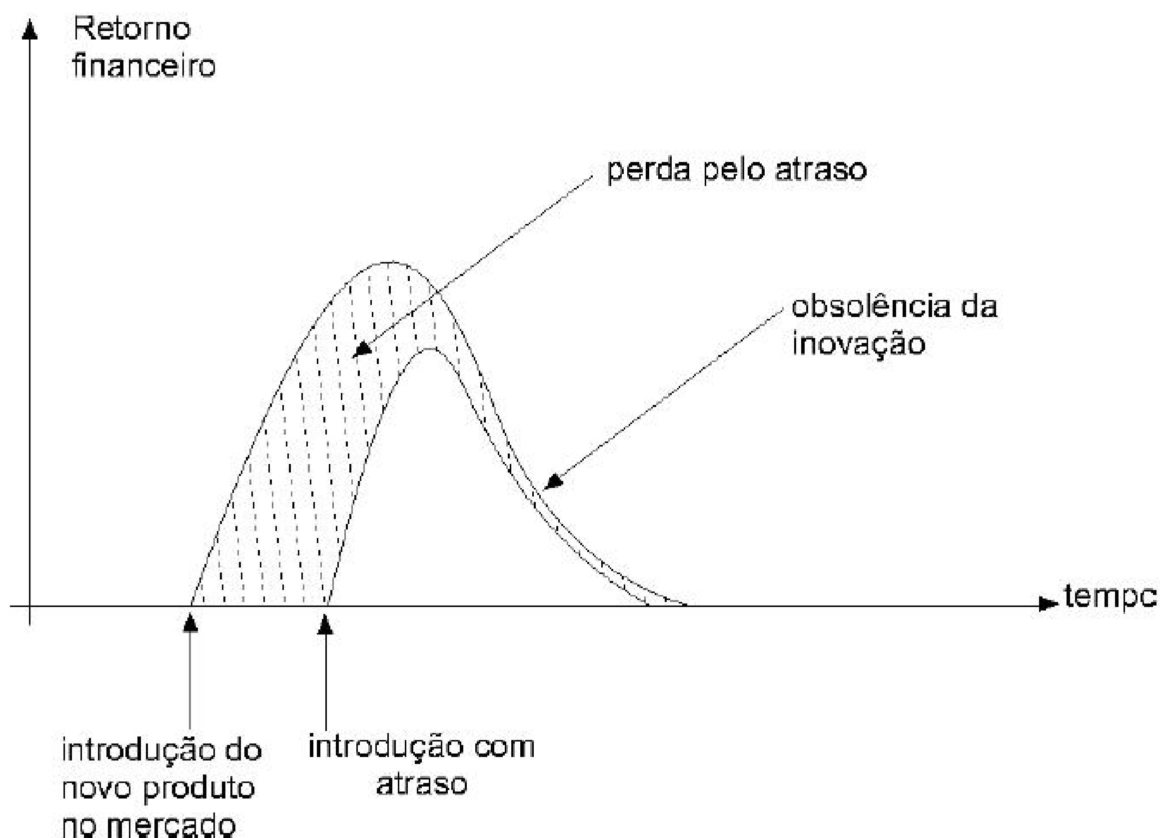

Figura 2.5 – Janelas de Tempo e Retorno Financeiro (CARRO, 2003)

Essa complexidade e o preço de desenvolvimento tende a escalar para projetos maiores, pois podem necessitar de modelagem ampliada e melhor definida, com vários níveis de abstração, ferramentas computacionais de custo muito elevado e pode precisar do envolvimento de equipes multidisciplinares (*hardware* analógico, *hardware* digital, sistema operacional, desenvolvimento de *software*, teste etc.). A complexidade e o preço de desenvolvimento são ainda maiores caso o projeto envolva uma rede de sistemas embarcados interligados, chamado de *Controller Area Network* (*CAN*), e/ou utilize *SoC*s, o que leva às organizações a aceitarem apenas projetos que possuam garantidamente volume muito alto de produção ou lucro garantido (CARRO, 2003).

A Figura 2.6 apresenta um exemplo de metodologia para um grande projeto de sistema embarcado:

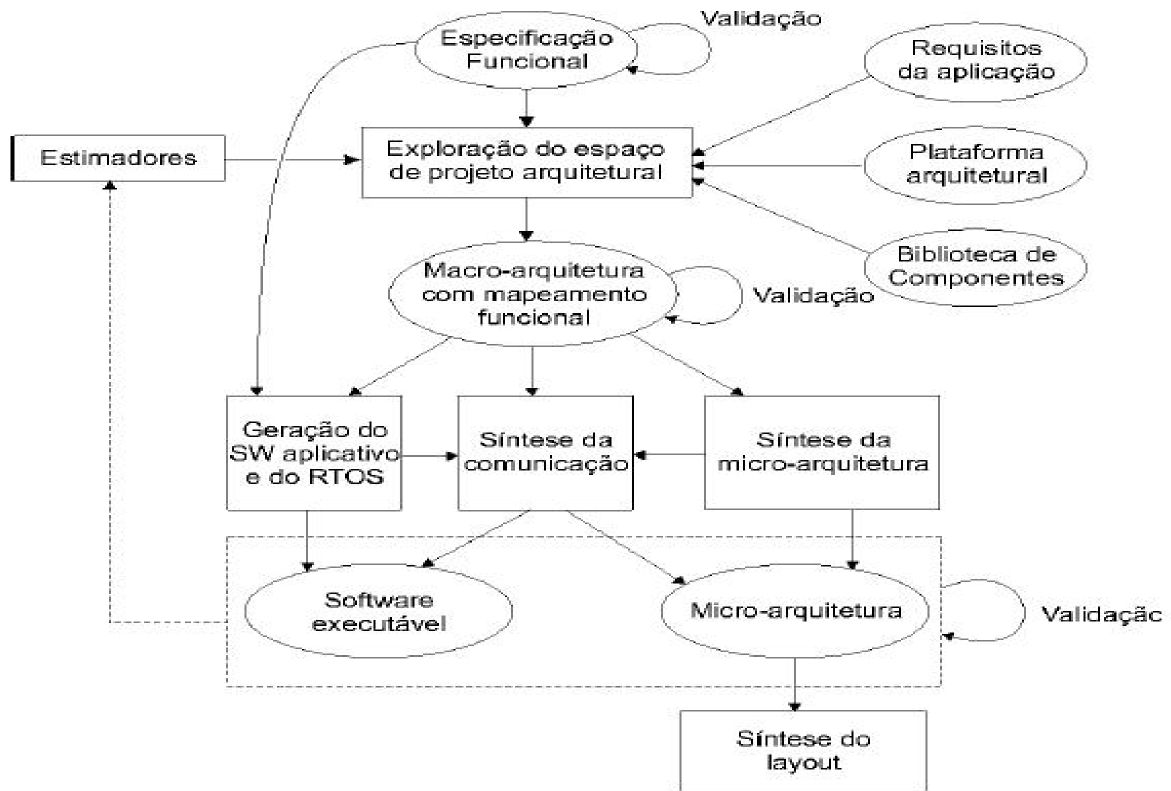

Figura 2.6 – Metodologia de Projeto Embarcado (CARRO, 2003)

Os projetos de sistemas embarcados enfrentam ainda diversos desafios devido à sua natureza não tradicional, especialmente quando uma empresa não especializada em TI (Tecnologia da Informação) decide incorporar tecnologia embarcada em seus produtos. Segundo Konana (2007), alguns desses desafios são:

- Projeto modular: existem dois modelos de projeto atualmente, o modular (dividido em partes que são realizadas individualmente) e o tradicional (feito seqüencialmente). A maioria dos produtos é feito de modo tradicional, levando os sistemas embarcados juntos para esse modelo de desenvolvimento, mas os sistemas embarcados, por serem sistemas computacionais, se beneficiam bem mais da forma modular.

Isso quer dizer que os projetos de produtos envolvendo sistemas embarcados precisam ser atualizados;

- Especificação: As interfaces de usuários entre um sistema embarcado e um produto físico precisam ser claramente especificadas;

- Capacidades organizacionais: Devido à evolução da tecnologia embarcada e o sistema de projeto modular, a frequência e a velocidade das inovações irão aumentar, tornando produtos obsoletos ainda mais rapidamente, o que por sua vez diminui a vida útil dos mesmos e a possibilidade de lucros (algo já crítico em sistemas embarcados);

- Conflitos potenciais com serviços já existentes: Podem haver conflitos entre serviços criados antes da implantação da tecnologia embarcada em um produto com o sistema embarcada ou suas funções;

- Treinamento e desenvolvimento profissional da força de trabalho: Os desenvolvedores necessitam ter conhecimento sobre TI para reconhecer oportunidades, enquanto os seus funcionários, especialmente os que prestam serviços, necessitam ter ao menos um conhecimento razoável de sistemas embarcados;

- Privacidade: Dependendo de como for desenvolvido um sistema embarcado, os produtores e os provedores de serviço podem ter acesso à localização do consumidor, suas preferências de uso, lugares que frequenta, senhas de aplicativos e diversas suas informações pessoais e privadas.

Para combater esses desafios, as organizações que desenvolvem sistemas embarcados vêm trabalhando com a padronização das fases de desenvolvimento, como a adoção do paradigma de projeto baseado em plataformas, uma arquitetura de *hardware* e de *software* que atende a um mesmo domínio de aplicações similares. Esse paradigma é altamente parametrizável, então não é muito

complicado e nem custoso fazer pequenas alterações para cada projeto, mesmo que o ideal seria reutilizar o máximo possível (CARRO, 2003).

Esta estratégia apóia e viabiliza o reuso de componentes e sistemas previamente projetados, desenvolvidos e testados, reduzindo o tempo e os recursos despendidos com pesquisa, modelagem, projeto e desenvolvimento e permitindo que o produto seja integrado antecipadamente no mercado e permaneça por mais tempo sem se tornar obsoleto. Este reuso pode ser ainda mais reforçado adotando-se padrões de arquitetura, modelagem, projeto, desenvolvimento e teste, e também com a adoção de ferramentas e ambientes de modelagem, projeto, desenvolvimento e teste (CARRO, 2003).

A Figura 2.7 demonstra a diferença de custo de um mesmo sistema *SoC* em um modelo tradicional e em um modelo com reuso e com padrões, mas essa figura pode ser usada para qualquer tipo de sistema embarcado (CARRO, 2003):

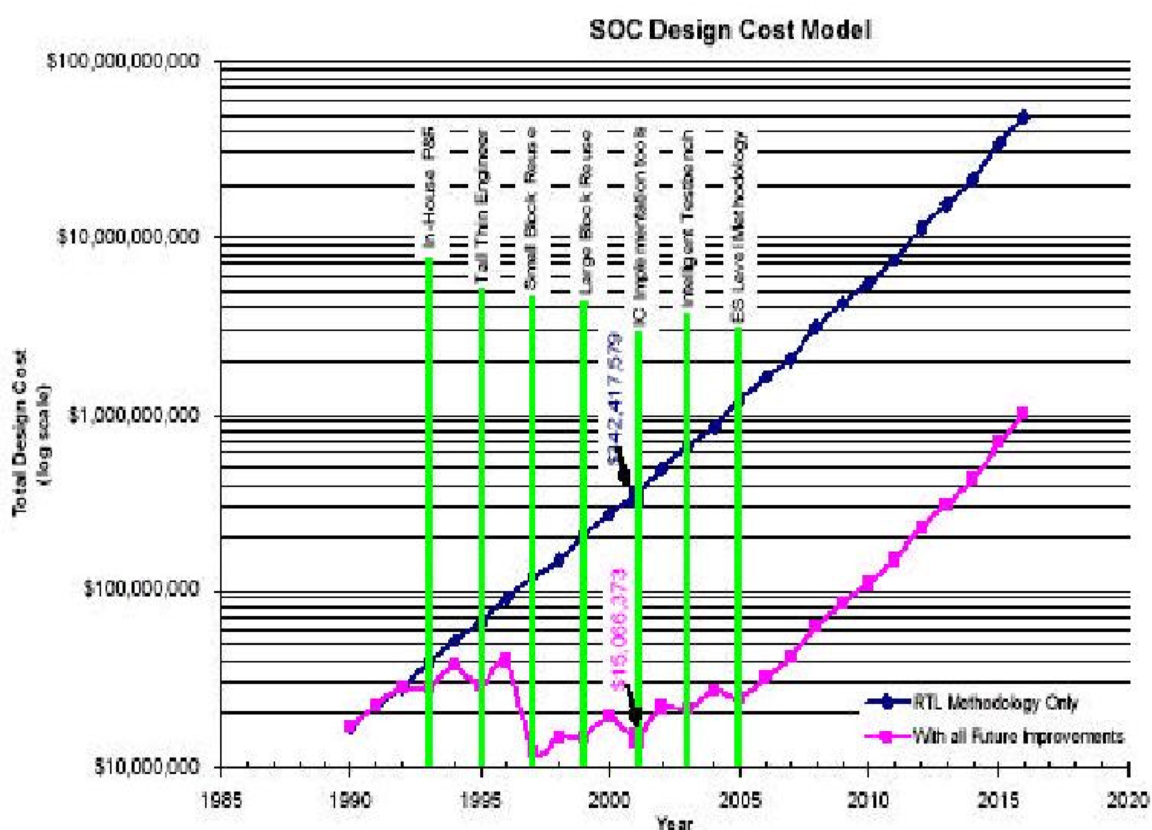

Figura 2.7 – Comparação entre Projeto Tradicional e de Reuso (CARRO, 2003)

Neste modelo de paradigmas e reuso, um projeto de sistema embarcado consiste em encontrar um derivativo de uma plataforma que atenda os requisitos da aplicação. Iniciando-se de uma especificação de alto nível da aplicação, explora-se as soluções arquiteturais possíveis, estimando o impacto das modificações necessários e dos particionamento das funções. É então feita a síntese da estrutura de comunicação mais adequada, responsável por integrar os componentes. O resultado pode não ser o melhor possível ao cliente em termos de energia, desempenho, tempo e recursos quando comparado ao sistema tradicional, mas o impacto não é tão grande e os desenvolvedores poupam bastante tempo de projeto (CARRO, 2003).

Ainda neste modelo, as inovações dependem e focam mais em *software* do que em *hardware*, já que a arquitetura de *hardware* está automatizada. Com a automação e padronização dos projetos de *hardware* dos sistemas embarcados, o mercado agora visa automatizar o projeto de *software*, com objetivo de diminuir ainda mais o tempo de projeto, aumentar a vida útil do produto e maximizar lucros (CARRO, 2003).

Já existem poderosas armas de padronização do projeto de software, sendo uma das mais notáveis e importantes a utilização de sistemas operacionais especializados denominados sistemas operacionais embarcados (CARRO, 2003).

Independente dos modelos utilizados, é vital a utilização de uma metodologia de testes para o sistema, pois é complicada a substituição ou a atualização de um sistema embarcado em produtos já lançados no mercado (CARRO, 2003).

1.3.4 ARQUITETURA DE SISTEMAS EMBARCADOS

O espaço arquitetural de projeto de sistemas embarcados é demasiado vasto, o que oferece tanto oportunidades quanto desafios aos produtores de tais sistemas. A arquitetura de *hardware* pode conter um ou mais processadores, microprocessadores e/ou microcontroladores (que, recapitulando, desempenham não só funções de processadores, mas também possuem memória própria e suporte à periféricos como dispositivos de entrada e saída de dados), além de memória

extra, interfaces de periféricos, blocos dedicados e diversos periféricos. Além disso, a estrutura de comunicação pode varias de um simples cabeamento direto, passando por barramentos, até a utilização de redes *wireless* complexas (CARRO, 2003).

Não o bastante, segundo Carro (2003), os microprocessadores e microcontroladores especializados para aplicações embarcadas podem ser ainda mais específicos e pertencer a um tipo conforme seu comportamento e possibilidades, alguns dos quais são mencionados a seguir:

- *RISC* (*Reduced Instruction Set Computing*), que possui um conjunto de instruções reduzidas afim de simplificar os sistemas, melhorar o desempenho e economizar energia;

- *VLIW* (*Very Long Instruction Word*), o qual trabalha com palavras de instrução maiores para diminuir os gargalos de memória e para aumentar a velocidade de execução através de instruções múltiplas ou de maior complexidade, diminuindo também o desperdício de energia;

- *DSP* (*Digital Signal Processor*), que é especializado em trabalhar com sinais digitais (sinais de valores discretos, ou seja, de valores definidos e finitos);

- *ASIP* (*Application-Specific Instruction Set*), o qual possui conjunto de instruções definidos e delimitados para realizar um conjunto especifico de ações ou funções, melhorando desempenho, diminuindo gastos de energia e possivelmente barateando custos. Geralmente é desenhado para atender sistemas embarcados específicos e amplamente utilizados.

Além do tipo do, é essencial saber ainda a potencia, o consumo de energia, além de a quantos dispositivos pode-se conectar e a quais possui suporte (CARRO, 2003).

Uma alternativa mais econômica é um *FPGA* (*Field Programming Gate Array*), que permite a programação de portas em tempo de execução (CARRO, 2003).

Se tratando da memória, em um sistema embarcado, sua utilização deve ser cuidadosamente calculada, pois maiores tamanhos e velocidades estão relacionadas à um maior consumo de energia, porém memórias não permitem expansões, apenas substituições, algo complicado e custoso de se fazer em um sistema embarcado já lançado no mercado, então capacidade de reuso e tamanho extra para futuras atualizações também devem ser levados em conta (CARRO, 2003).

Há duas possíveis soluções de reuso de memória: descarte de dados usados, como usado em computadores comuns, ou utilização de um *buffer* circular, que ao encher, retorna para o começo da memória e começa a excluir instruções antigas e substituir por novas (porém um cuidado especial deve ser tomado para que instruções não resolvidas não sejam excluídas). Se um *buffer* enche e passa de seus limites, perdendo dados, independente de ser um *buffer* normal ou circular, é um evento denominado *buffer overflow* (CARRO, 2003).

Como os dispositivos de memória são mais lentos do que os dispositivos de processamento, podem acontecer problemas de desempenho quando o sistema não é muito complexo e não é devidamente planejado. Felizmente, o sistema de hierarquia de memórias também funciona para sistemas embarcados (CARRO, 2003).

Um *software* embarcado pode ainda depender de diversos microprocessadores, microcontroladores, pastilhas ou *chip*s para ser executado em um mesmo sistema embarcado, sendo constituído de programas, módulos ou processos diferentes implantados em cada um, mas mantendo uma dependência e se comunicando frequentemente. Projetos desse tipo podem ocorrer por diversos motivos, como limite de processamento individual de cada unidade ou criticidade e velocidade do sistema como um todo (CARRO, 2003).

Podemos ter dispositivos de memória locais (conectado a um ou a alguns dispositivos de processamento) e globais (conectados a todos os dispositivos de

processamento) em tais projetos, o que adiciona à complexidade do sistema (CARRO, 2003).

Para melhorar o desempenho, o paralelismo e/ou o pseudoparalelimo (utilização de *pipeline*s) pode ser necessário. Não é raro encontrar, em grandes projetos de sistemas embarcados, microprocessadores e microcontroladores multinúcleos, ou mesmo averiguar a existencia de múltiplos microprocessadores e microcontroladores, mas assim como em outros sistemas computacionais, deve-se tomar cuidado com dependências de dados (as instruções que dependem de resultados de outras instruções) e a ordem correta de instruções, especialmente quando é usado um mesmo conjunto de dispositivos de memória entre os microprocessadores e microcontroladores (CARRO, 2003).

O último desafio de uma boa arquitetura é a comunicação entre os componentes. As ligações diretas, também conhecidas como ponto-a-ponto, são as mais rápidas, mas necessitam ser planejadas e refeitas para cada projeto em que haja qualquer mudança de *hardware*, o que consecutivamente afeta o tempo de projeto. Já os barramentos são mais reusáveis, porém não permitem transações simultâneas, permitem que apenas uma transação ocorra por vez dentro de sua área ou domínio, o que ocasiona perda de desempenho conforme o tamanho do projeto e a quantidade de componentes conectados (CARRO, 2003).

Uma opção para esse problema é a utilização de uma hierarquia de barramentos, onde há vários sub-barramentos espalhados conectados por um barramento maior, o que diminui o impacto de usar um único barramento, mas pode ocasionar uma paralisação dos recursos de comunicação. É possível utilizar também uma arquitetura de comunicação mista entre ponto-a-ponto e barramentos, com uma das tecnologias a nível local e outra a nível global no sistema. Há ainda outras tecnologias de comunicação, mas nenhuma é ideal para todos os projetos, pois tudo depende da quantidade de dispositivos e da velocidade necessária de comunicação para cada dispositivo em um projeto (CARRO, 2003).

Podemos ter essa estrutura inteira de um sistema embarcado dentro de uma única pastilha ou *chip*, o que é denominado como *SoC* (*System-on-a-Chip*). Um *SoC*

é comumente menor que um microcontrolador, porém é mais potente, possui mais memória e é mais especifico em suas funções, além de ser customizado e poder possuir periféricos já embutidos, com capacidade de substituir um sistema embarcado inteiro. Desenvolver um *SoC* ao invés de um sistema embarcado padrão é consideravelmente mais custoso financeiramente e mais complexo em seu desenvolvimento, sendo comumente usado em aplicações críticas ou quando o tamanho do sistema é crucial (CARRO, 2003).

Por vezes, um ou mais sistemas operacionais embarcados ou de tempo real podem ser necessários devido ao suporte à comunicação entre processos ou dispositivos e também devido ao suporte à escalonamento de processos, mas também pode ser usado simplesmente para diminuir o tempo de projeto através da padronização, o que aumentaria a vida útil do produto e a possibilidade de lucros (CARRO, 2003).

Alguns trabalhos propõem o uso de uma rede dentro de uma pastilha ou *chip*, o que é chamado de *NoC* (*Network-on-a-Chip*), que consiste em um conjunto de roteadores e canais dedicados à comunicação dos componentes internos e externos. Pode ser, portanto, utilizado dentro de um *SoC*. As conexões entre os roteadores são conexões ponto-a-ponto, o que significa que a rede pode transmitir varias mensagens ao mesmo tempo, possui paralelismo e é escalável (basta apenas aumentar o número de roteadores). Porém, a eficiência de um *NoC* depende de diversos fatores, como tamanho (em quantidade de fios) do canal de roteamento, política de prioridade das mensagens, topologia (disposição e conexão dos elementos) utilizada no *NoC*, e estratégia de chaveamento. Além disso, a distancia entre certos elementos no *NoC*, como memória e processador, pode ser crucial. É importante enfatizar que roteadores possuem custo elevado, e uma *NoC* é constituída de diversos roteadores (CARRO, 2003).

A Figura 2.8 exemplifica três modelos de topologia *NoC*, sendo a mais adequada dependente do projeto e do custo.

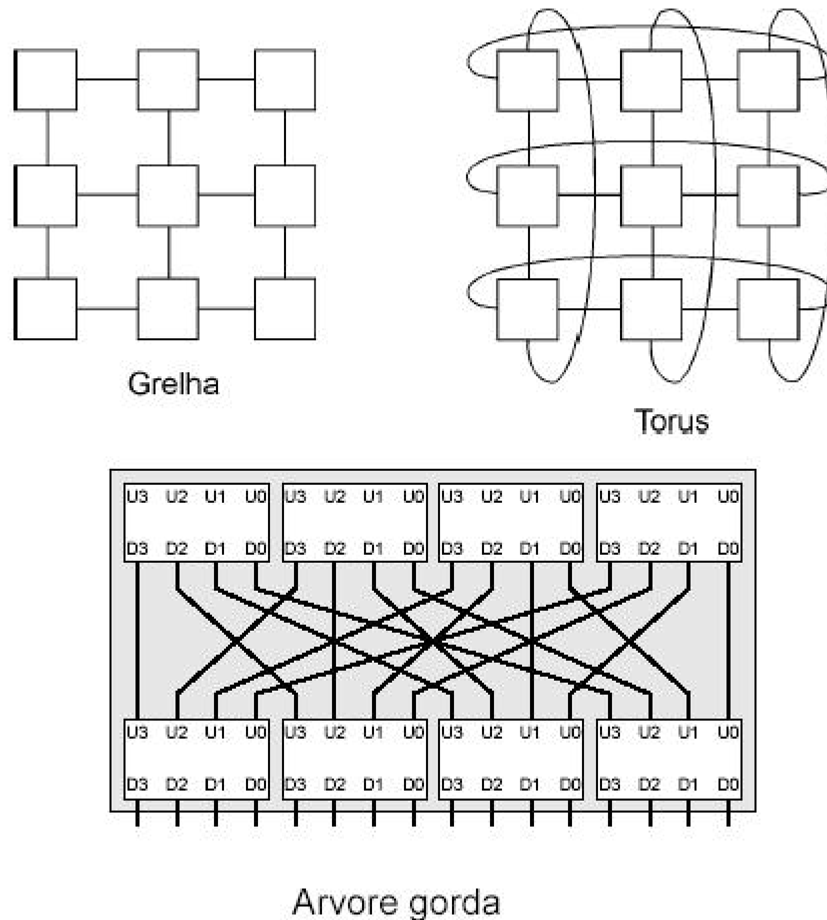

Figura 2.8 – Modelos de *NoC*s (CARRO, 2003)

1.3.5 REQUISITOS DE SISTEMAS EMBARCADOS

Os projetos de sistemas embarcados, em sua maioria, possuem seus requisitos bem definidos (FRIEDRICH, 2009). Os requisitos mais comuns e mais importantes, segundo Friedrich (2009) são:

1. Computação de Recursos Restritos: Devido à escassez dos recursos e componentes disponíveis, é necessário utilizá-los eficientemente;

2. Requisitos de Tempo Real: A maioria das aplicações embarcadas interagem constantemente com o mundo real ou então tratam de assuntos sensitivos como médicos e bancários, então precisam de limites severos de tempo;

3. Portabilidade: Para garantir o barateamento dos custos, a maioria dos componentes deve ter suporte a diferentes plataformas e devem ser reusáveis sob diferentes circunstâncias;

4. Confiabilidade Alta: Por serem utilizados em diversas aplicações críticas e pela dificuldade de se atualizar ou substituir sistemas embarcados em produção, falhas de *software* e de *hardware* são muito custosas e problemáticas;

5. Estabilidade, robustez e confiança: Disponibilidade, ou seja, capacidade de continuar trabalhando, preferencialmente sem que haja falhas ou com isolamento ou tolerância de falhas;

6. Controle de falhas: Falhas podem ocorrer por diversos motivos, e a tecnologia embarcada, especialmente em redes, por mais estável e confiável que necessite ser, não está isenta disso. De modo geral, falhas não devem impactar as funções gerais do sistema, devendo ser toleradas, controladas, isoladas ou registradas para correção posterior;

7. Proteção: Danos acidentais severos ao ambiente, às propriedades e às pessoas (incluindo óbito) não devem acontecer. Todo sistema embarcado em que isso pode acontecer, como equipamentos médicos, químicos, industriais e militares, além de meios de transporte, deve atender a esse requisito;

8. Segurança: Capacidade do sistema de prevenir acesso não autorizado ao sistema e a dados importantes e confidenciais aos clientes (como senhas);

9. Privacidade: Há informações e dados que podem não ser importantes aos clientes, mas que devem ser mantidos privados, como fotos. Esse requisito se preocupa em manter confidenciais os dados do cliente;

10. Escalabilidade: É a capacidade do sistema ser prontamente expandido ou poder gerenciar quantidades crescentes de trabalho, informações e/ou registros sem impactos profundos;

11. Melhorias: Relacionado com a capacidade do sistema ser melhorado ou mesmo se atualizar espontaneamente, seja em novas funcionalidades ou melhoras nas existentes.

1.4 SISTEMAS OPERACIONAIS EMBARCADOS

Se um sistema embarcado precisa ser produzido rapidamente, ou possua software de aplicação composto de múltiplos processos, um sistema operacional pode ser necessário, oferecendo serviços como escalonamento de processos e suporte a comunicação entre componentes (CARRO, 2003).

1.4.1 SUAS CARACTERÍSTICAS

Um sistema operacional embarcado possui características muito similares a de um sistema embarcado em si, pois um sistema operacional funciona como um extensor do sistema computacional a qual está conectado, e não deve interferir muito em suas funcionalidades básicas (CARRO, 2003).

Por isso, um sistema operacional embarcado deve possuir baixo custo, alto desempenho, segurança, confiabilidade, privacidade, escalabilidade e baixos consumos de energia, memória, processamento e espaço de armazenamento (CARRO, 2003).

Outra característica importante de qualquer sistema operacional, inclusive embarcado, é ajudar e servir de plataforma para que outros programas desempenhem suas funções, ao mesmo tempo em que protege e gerencia os recursos de *hardware*. Deve também tratar tanto de falhas de *software* quanto de *hardware* (TANENBAUM, 2003).

Sistemas operacionais modernos, inclusive embarcados, devem também ser capazes de escalonar (organizar) processos aos núcleos de processamento, gerenciar a memória e auxiliar nas comunicações dos componentes de *hardware* e processos (TANENBAUM, 2003).

Diferente de outros sistemas operacionais, a maioria dos sistemas operacionais embarcados trabalham apenas no modo núcleo, pois chamadas ao sistema utilizam muito poder de processamento, memória, energia elétrica e tempo em sistemas embarcados. Isso quer dizer que os programas possuem acesso total ao hardware, o que poderia causar problemas de segurança, mas gerenciar isso também impactaria muito o desempenho do sistema, e como sistemas embarcados costumam ser bem específicos e ter poucos programas, é algo que ou pode ser desconsiderado (dependendo do projeto) ou previsto e endereçado no projeto (CARRO, 2003), além de que diversos sistemas operacionais embarcados também possuem propriedades e atingem exigências de sistemas de tempo real, não somente em termos de velocidade, mas em garantia de resposta às instruções (FRIEDRICH, 2009).

Um sistema operacional embarcado deve ainda ser flexível e configurável, especialmente para ajudar na padronização das etapas de desenvolvimento de aplicações embarcadas (CARRO, 2003).

Por outro lado, um sistema computacional que utiliza um sistema operacional necessita de mais memória principal e mais poder de processamento, o que implica na necessidade de *hardware* mais potente, além de maiores gastos com energia em sistemas embarcados (CARRO, 2003).

Outro fator importante é a confiabilidade, a segurança e a privacidade, que podem ser impactadas tanto positivamente quanto negativamente dependendo da qualidade e do projeto do sistema operacional devido ao fato de proteger o *hardware* do *software* e facilitar as interações do usuário com o sistema ao abstrair e interpretar as informações em uma interface (TANENBAUM, 2003). Todas essas áreas são críticas para um projeto de sistema embarcado, especialmente para os projetos mais simples ou menos funções (CARRO, 2003), o que torna a utilidade de

sistemas operacionais em aplicações embarcadas de menor porte questionável, mas justificável às empresas dependendo de quanto tempo será poupado no projeto. Por isso, tais sistemas operacionais são normalmente usados em projetos mais complexos ou menos exigentes.

Um sistema operacional embarcado pode ser do tipo *GPOS* (*General Purpose Operating System*), que é um sistema operacional destinado a propósitos gerais e atende uma vasta gama de diferentes instruções, ou pode ser especializado ou configurável para algum propósito ou conjunto de instruções, o que melhora o desempenho e reduz o consumo de energia, mas geralmente é mais caro (FRIEDRICH, 2009).

1.4.2 SISTEMAS ATUALMENTE DISPONÍVEIS

Os sistemas operacionais embarcados mais tradicionais costumam ser *GPOS*s e são tipicamente grandes para sistemas embarcados (ocupam centenas de *KB*s), mas possuem ricos conjuntos de interfaces de programação. São projetados para os sistemas com maiores recursos de memória, processamento e energia e comumente possuem suporte à escalabilidade, multitarefas, conexões de rede, proteção de memória, interfaces de programação que também são conhecidas como *API*s (*Aplication Programing Interface*) e proteção contra falhas. Alguns sistemas operacionais embarcados tradicionais incluem *WinCE* (*Windows Embedded Compact*), *VxWorks*, *PalmOS*, *QNX*, *LynxOS*, *OS-9* e *Symbian* (FRIEDRICH, 2009).

Entre os citados, o *VxWorks*, desenvolvido pela *Wind River*, é o mais adotado, e inclusive é utilizado pela Estação Espacial Internacional. O *VxWorks* foi criado no começo da década de 80 e atualmente fornece uma *API* com pelo menos 1800 métodos, além de gerenciamento de dispositivos de entrada e saída de dados, suporte à rede e sistemas de arquivos. Ele provê um ambiente de desenvolvimento virtual e traz ainda a bancada de trabalho da *Wind River*, que é um conjunto de ferramentas para acelerar e auxiliar no desenvolvimento com o *VxWorks*. Possui suporte para agendamento, o qual inclui até 256 níveis de prioridades. O servidor ou ambiente de desenvolvimento pode ser *Windows XP*, *Windows* 2000 *Professional*, *SuSE Linux*, *Solaris* ou *Red Hat Linux* (FRIEDRICH, 2009).

O *QNX* foi desenvolvido sobre a idéia de executar um sistema operacional dividido em várias partes, denominadas de *servers*. Isto difere dos sistemas operacionais mais tradicionais, nos quais o sistema operacional é um único e grande programa constituído de diversas partes ou habilidades especiais. Essa idéia habilita que o *QNX* seja portado ou executado apenas parcialmente, poupando poder de processamento, memória e energia. Como o VxWorks, o *QNX* também possui suporte para agendamento com 256 níveis de prioridades. A versão *QNX Neutrino*, produzida em 2001, foi portada para diversas plataformas e funciona em tecnicamente todos os processadores modernos disponíveis no mercado de sistemas embarcados (FRIEDRICH, 2009).

O *Windows CE* é um sistema operacional embarcado modular (dividido e utilizado em módulos), portável e de tempo real desenvolvido no fim da década de 90 pela *Microsoft* para ser utilizado em aparelhos com memória pequena. Suporta até 32 processos ativos com *threads* (múltiplas linhas de execução) em cada processo, além de agendamento com 256 níveis de prioridades. Possui também primitivas de eventos, semáforos e sincronização. Possui três principais plataformas de desenvolvimento ricas em recursos, que são *Windows Mobile*, *SmarthPhone* e *Portable Media Center* (FRIEDRICH, 2009).

Outros sistemas operacionais embarcados seguem uma orientação por componente para configurar aplicações específicas, como o *icWORKSHOP* e o *eCos* (*embedded Configurable operating system*), os quais têm como objetivo composições leves e estáticas. Tais sistemas são constituídos por conjuntos de componentes que são ligados para formar uma aplicação. Tais componentes podem ser classes ou pacotes, como no *eCos*, ou podem ser especializados, assim como no *icWORKSHOP* (FRIEDRICH, 2009).

O *VEST* é um conjunto de ferramentas para a construção de sistemas embarcados baseados em componentes e que realizam análises estatísticas extensivas, como dependência de recursos, agendamento e interfaces de correção gramatical (FRIEDRICH, 2009).

O eCos é o sistema operacional *freeware* (livre para uso) e *opensource* (de código aberto à comunidade) mais adotado no mercado. Foi lançado em 1986 e ele provê uma ferramenta de configuração por linhas e outra ferramenta de configuração gráfica, o que permite adaptar o sistema operacional para realizar funções e requerimentos de aplicações específicas, além de permitir ao desenvolvedor configurar o sistema para melhor atender os requisitos de memória e desempenho. Possui primitivas de eventos, semáforos, sincronização e de caixas de correio eletrônico, além de suportar agendamento com até 32 níveis de prioridades. O *eCos* suporta ainda diversos tipos de processadores e o ambiente ou servidor de desenvolvimento pode ser *Linux* ou *Windows* (FRIEDRICH, 2009).

Há sistemas operacionais embarcados especializado para sistemas operacionais de menor porte, como *pSOSystem*, *CREEM*, *Ariel* e *OSEKWorks*. Tais sistemas operacionais possuem restrições severas de execução e de modelos de armazenamento (FRIEDRICH, 2009).

Alguns sistemas operacionais contemporâneos, originalmente destinados à sistemas computacionais de maior porte, como o Linux, possuem extensões e ferramentas que os permitem suportar aplicações de tempo real, (extensões como *uClinux*, *RT-Linux* e *RTAI*,) porém são adequados apenas à grandes projetos de sistemas de tempo real devido à sua arquitetura e seu tamanho. Além disso, as preocupações com segurança, confiabilidade e privacidade devem ser atendidas. Através de algumas soluções propostas, foram desenvolvidos o *L4 OS* e o *Minix OS* (FRIEDRICH, 2009).

As Tabelas 2.2a e 2.2b apresentam o quão bem alguns dos sistemas operacionais embarcados mencionados atendem, se adéquam ou fornecem os requisitos estudados na subseção 2.3.5.

Tabela 2.2a – Requerimentos de Sistema Operacionais Embarcados (FRIEDRICH, 2009)

SO	**1**	**2**	**3**	**4**	**5**	**6**
VxWorks	Parcial	Parcial	Adequado	Adequado	Parcial	Parcial
QNX	Parcial	Parcial	Adequado	Adequado	Parcial	Parcial
WinCE	Parcial	Parcial	Adequado	Parcial	Inadequado	Inadequado
eCos	Adequado	Parcial	Adequado	Parcial	Inadequado	Inadequado
pSOSystem	Adequado	Parcial	Parcial	Parcial	Inadequado	Inadequado
RTAI	Inadequado	Adequado	Parcial	Parcial	Inadequado	Inadequado
uClinux	Adequado	Inadequado	Adequado	Parcial	Inadequado	Inadequado
L4	Parcial	Parcial	Parcial	Adequado	Adequado	Adequado
Minix	Parcial	Inadequado	Parcial	Adequado	Adequado	Adequado

Tabela 2.2b – Requerimentos de Sistema Operacionais Embarcados (FRIEDRICH, 2009)

SO	**7**	**8**	**9**	**10**	**11**
VxWorks	Parcial	Adequado	Parcial	Adequado	Parcial
QNX	Parcial	Adequado	Parcial	Adequado	Parcial
WinCE	Inadequado	Parcial	Parcial	Parcial	Inadequado
eCos	Parcial	Parcial	Parcial	Adequado	Inadequado
pSOSystem	Parcial	Parcial	Inadequado	Parcial	Inadequado
RTAI	Inadequado	Parcial	Inadequado	Parcial	Inadequado
uClinux	Inadequado	Parcial	Inadequado	Parcial	Inadequado
L4	Parcial	Adequado	Adequado	Adequado	Parcial
Minix	Parcial	Adequado	Adequado	Adequado	Parcial

2 DESENVOLVIMENTO

O desenvolvimento desse projeto consiste em uma série de experimentos com ferramentas de desenvolvimento e programação de sistemas embarcados, com tentativa de conversão de ao menos um experimento de um sistema embarcado sem sistema operacional para uma aplicação embarcada com sistema operacional embarcado ou vice-versa.

2.1 FERRAMENTAS UTILIZADAS E ESTUDADAS

Há uma grande variedade de ferramentas de desenvolvimento para sistemas embarcados, o que inclui compiladores, projetadores de *hardware*, simuladores de *hardware* e sistemas operacionais. Ao decorrer do desenvolvimento deste projeto, algumas dessas ferramentas foram selecionadas e estudadas, as quais estão descritas nas subseções a seguir.

2.1.1 PROTEUS

O *Proteus* é um conjunto de ferramentas desenvolvidas pela *Labcenter Eletronics* para *design*, esquematização e simulação de sistemas embarcados e de componentes e recursos para sistemas embarcados. O módulo mais utilizado desse conjunto é o *Isis*, pois permite esquematização, *design* e simulação de sistemas embarcados, além de permitir alterações e customizações leves nos componentes disponíveis para caso estes não sejam adequados para o sistema em mente (LABCENTER ELETRÔNICS, 2011), motivos pelo qual foi escolhido para os experimentos.

2.1.1.1 ISIS

Isis é uma ferramenta do *Proteus* que permite, de forma simples e rápida, montar, configurar e simular componentes de *hardware* embarcado, além de definir seus atributos e condições (LABCENTER ELETRÔNICS, 2011).

Um exemplo de tela de um projeto no *Isis* pode ser observado na Figura 3.1

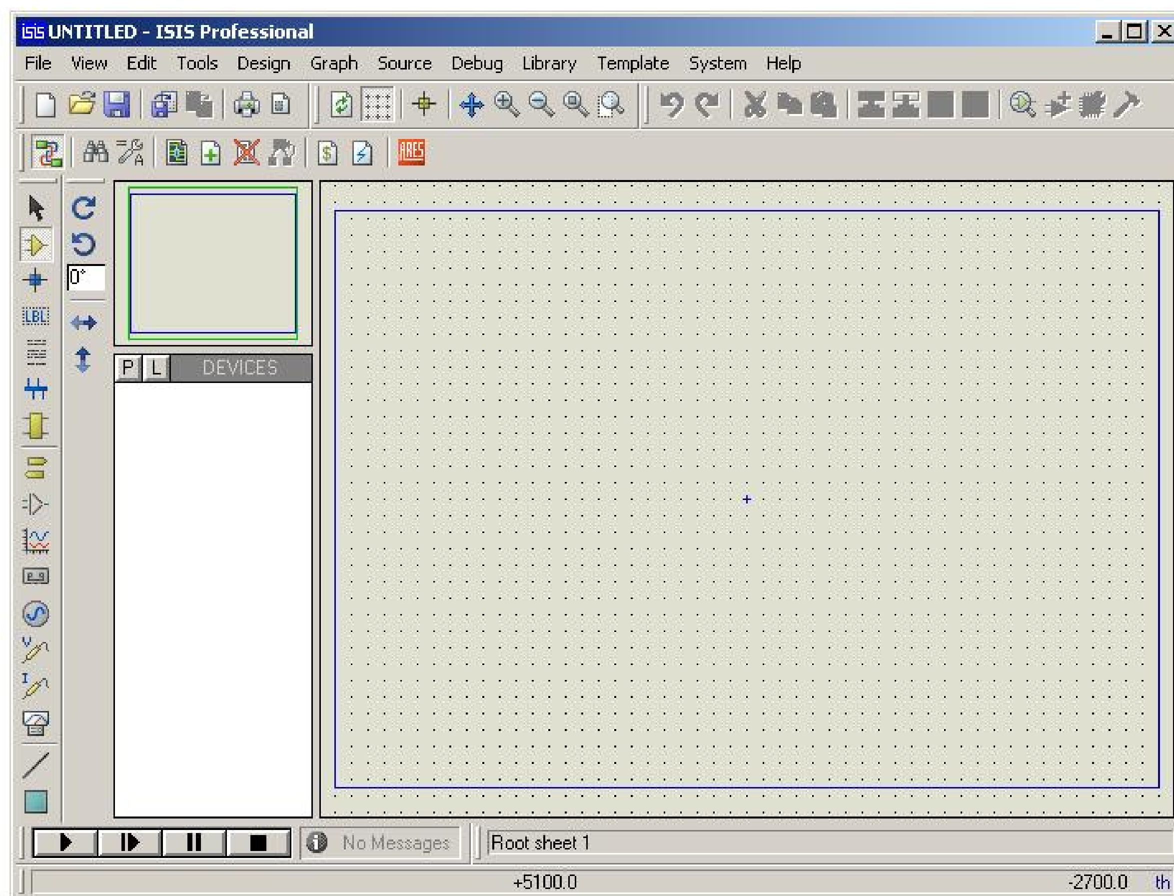

Figura 3.1 – Tela de Projeto no *Isis* (GALLASSI, 2011)

Os componentes precisam ser adicionados ao projeto antes de utilizados, e podem ser conectados através de circuitos, porém os componentes não podem ser generalizados, já que são diversos e possuem atributos específicos (LABCENTER ELETRÔNICS, 2011).

2.1.1.1.1 ENERGIA

O *Isis* fornece vários dispositivos de energia, como baterias, pilhas e motores, além de força e aterramento elétrico gerais padrões, denominados de *power* e *ground*, que por serem frequentemente usados e não serem dispositivos propriamente ditos, se localizam separados dos demais componentes de projetos e sempre estão disponíveis. Há ainda os inputs e outputs, que são ligados quando atribuídos com o mesmo nome, e transferem sinal elétrico, analógico ou digital.

2.1.1.1.2 RESISTORES

Os resistores e fusíveis são de extrema importância para qualquer projeto, pois impedem que os componentes físicos sejam danificados ou queimados devido a correntes elétricas com alta voltagem ou variações anormais de tensão elétrica. Isto é evidenciado também no *Isis*, pois se uma tensão muito alta ou muito baixa passa por um componente, em especial se o componente for um microcontrolador ou microprocessador, o componente não funcionará corretamente. O *Isis* permite a configuração da resistência de cada resistor individualmente.

2.1.1.1.3 BOTÕES

Os botões são componentes que transferem sinal apenas enquanto estão pressionados, deixando de transferir o sinal quando não estão. Podem ser utilizados junto a componentes para causar efeitos apenas quando o usuário apertar o botão, como em buzinas, ou podem ser usados junto à microcontroladores configurados para detectar alterações em um sinal para criar efeitos mais duradouros, como modificar os cálculos de um sistema para mostrar em outra medida ou desligar uma porta de abastecimento, desligando certos componentes sem desligar o componente inteiro.

2.1.1.1.4 INTERRUPTORES

Um interruptor, também chamado de *switch*, é muito similar a um botão, porém a alteração na transmissão de sinal é constante após pressionado, requerendo que o switch seja pressionado novamente para voltar ao estado original.

Por fornecer sinal continuamente, é comumente usado em circuitos elétricos, como para ligar lâmpadas ou para fornecer energia elétrica a um sistema embarcado, mas, como botões, também podem ser usados junto à microcontroladores configurados para detectar mudanças em um sinal e realizar outras funções com base nas informações desse sinal.

2.1.1.1.5 PORTAS LÓGICAS

As portas lógicas são componentes emissores de sinal digital, e podem alterar o estado de transmissão, sendo ligados e desligados, de maneira similar a um interruptor (um pressionamento muda o sinal, e outra retorna ao sinal original). O sinal já vem no formato digital, o que facilita a manipulação de pinos e *port*s em componentes que os usam para ler ou enviar sinal, como os microcontroladores.

2.1.1.1.6 LÂMPADAS E *LED*S

As lâmpadas e os *LED*s são componentes que emitem luz ao receberem tensão elétrica suficiente. Normalmente são utilizados para iluminação ou sinalização de eventos, como se um botão for pressionado ou então ocorrer uma falha em um microcontrolador.

*2.1.1.1.7 DISPLAY*S

*Display*s são monitores que são ativados quando recebem sinal em seus diferentes pinos e conectores. Um *display* de segmentos precisa apenas discernir tensão forte de tensão fraca em cada um de seus pinos para acender ou desligar o segmento correspondente, enquanto um display de LCD possui leitores lógicos que recebem a mensagem, que é então interpretada e depois mostrada.

2.1.1.1.8 SENSORES DE TEMPERATURA

Sensores de temperatura são componentes que lêem a temperatura de determinado objeto ou do ambiente, temperatura a qual pode ser determinada manualmente pelo usuário no *Isis* para fins de teste, e enviam através de um circuito, as informações no formato analógico. Porém, esse sinal analógico precisa ser interpretado antes de demonstrado à um usuário, função em que um microcontrolador é bem útil.

2.1.1.1.9 MICROCONTROLADORES

Os microcontroladores são componentes complexos, e possuem dezenas ou até centenas de pinos, cada pino com uma identificação própria e com diferentes atributos, configurações e habilidades, dependente de como o microcontrolador foi projetado pelo fabricante. Os pinos são agrupados em conjuntos maiores, geralmente de 8 pinos, denominados *port*s. Os pinos e *port*s podem ser analógicos ou digitais, e ambos os tipos podem tanto enviar quanto receber informações (dependendo das configurações).

A programação de um programa para um microcontrolador pode ser feita pino a pino ou pode referenciar a *port* inteira, sendo que se todos os pinos forem digitais, a *port* se caracterizaria como um inteiro não-negativo de 8 *bit*s (que vai de 0 a 255 em números decimais).

Um parâmetro dos microcontroladores no *Isis*, para efeito de simulação, é o lugar do arquivo hexadecimal do programa o qual o microcontrolador deve executar. Ao contrário da programação convencional, se o programa não estiver sob um sistema operacional, é recomendado que o programa possua *loop* infinito (execução infinita), para que assim continue executando suas funções enquanto o microcontrolador receber energia elétrica, mas exceções podem ser necessárias dependendo da finalidade do sistema ou do consumo de energia.

A Figura 3.2 demonstra um projeto no *Isis* com alguns dos componentes descritos.

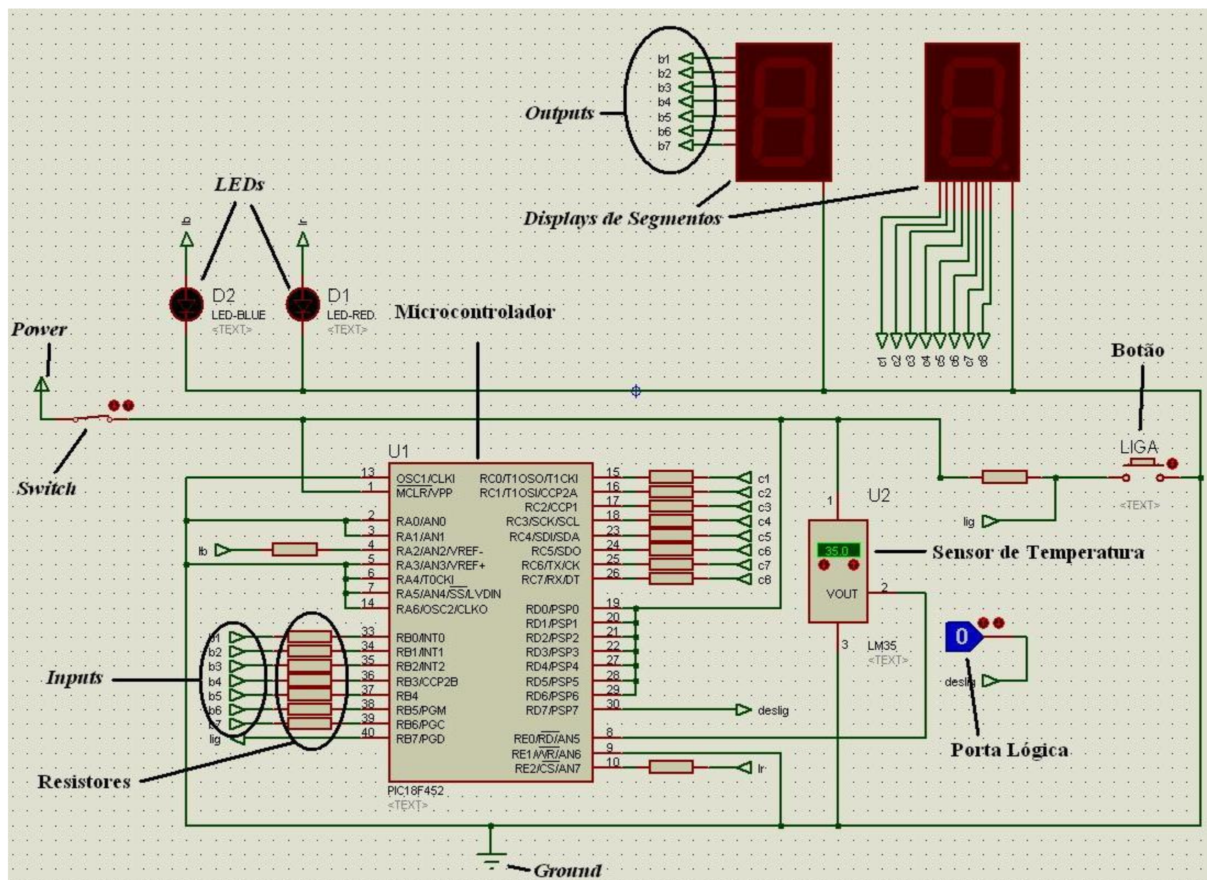

Figura 3.2 – Exemplos de Componentes no *Isis* (GALLASSI, 2011)

2.1.2 MIKROC

O *mikroC* é um compilador de linguagem C, desenvolvido pela *MikroEletronika*, para programação em microcontroladores da família *PIC* (uma série de microcontroladores que são produzidos pela *Microship Technology*). O *mikroC* possui ainda algumas ferramentas para auxiliar no desenvolvimento, como o decodificar de código para displays de 8 e 7 segmentos anodo (que recebe energia elétrica em um pino de alimentação e acende os pinos referentes aos segmentos quando ficam com tensão baixa) e cátodo (que é conectado ao aterramento de energia elétrica e acende os pinos referentes aos segmentos quando estes recebem tensão alta) (MIKROELETRONIKA, 2006).

O *mikroC* foi desenvolvido visando oferecer uma interface simples, que não compromete o controle e o desempenho das aplicações. Possui suporte para as bibliotecas em C, além de permitir a compilação de arquivos para o formato hexadecimal ou *HEX* (MIKROELETRONIKA, 2006), que podem ser usados no *Isis* pelos microcontroladores e motivo da utilização do *mikroC* nesse projeto.

Para compilar um programa no *mikroC*, é necessário primeiro criar um projeto, definindo qual microcontrolador será utilizado, qual a frequência de atualização do *clock* ou relógio microcontrolador, e as *flag*s ou variáveis de inicialização (MIKROELETRONIKA, 2006).

Para uma aplicação funcionar adequadamente, é recomendado sempre iniciar os pinos e *port*s do microcontrolador que serão usados para enviar e receber dados (MIKROELETRONIKA, 2006). A Figura 3.3 demonstra uma tela de projeto no *mikroC*.

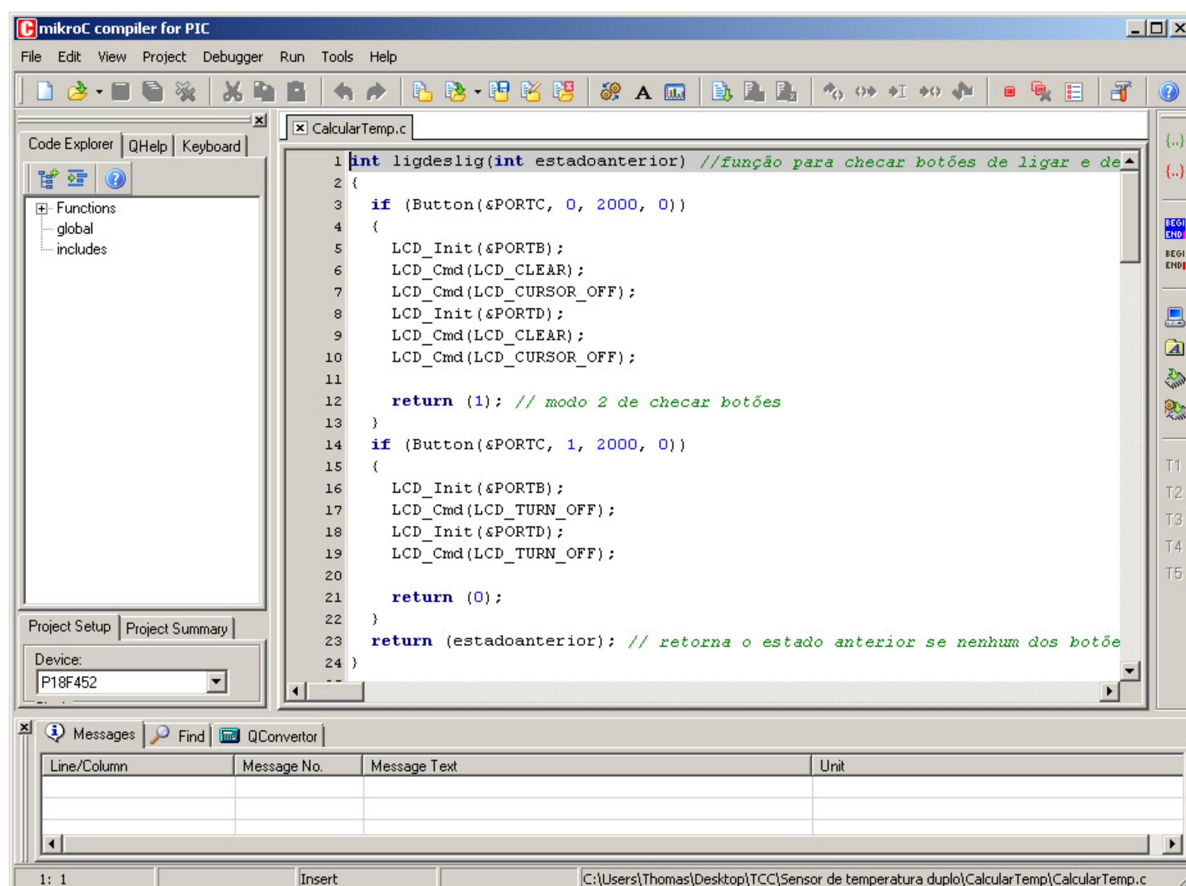

Figura 3.3 – Tela de um Projeto no *mikroC* (GALLASSI, 2012)

2.1.3 FREERTOS

Freeware (livre para uso) e *opensource* (código aberto à comunidade), o *FreeRTOS* é um sistema operacional embarcado excelente para tarefas de tempo real exigentes em sistemas de microcontroladores de pequeno e médio porte (que possuem de 16KB a 256KB de memória *RAM*). Ele é desenvolvido majoritariamente sobre a linguagem C e permite que as aplicações sejam organizadas em conjuntos de *thread*s independentes, permitindo a categorização das *thread*s, o que por sua vez facilita a programação concorrente (onde os programas compartilham o tempo de *CPU*) (BARRY, 2010). O fato de ser *freeware* e *opensource* influenciou em sua escolha para os experimentos desse projeto.

O *FreeRTOS* possui suporte à diversas famílias de microcontroladores e é suportado por muitos compiladores. Isto é possível devido à possibilidade de usá-lo através do seu código *source* (não compilado) e por ser desenvolvido como um conjunto de bibliotecas, pacotes e métodos, podendo ser facilmente integrado na maioria dos compiladores através dos comandos de inclusão de bibliotecas. Isto também permite a utilização de parte específicas do *FreeRTOS*, economizando poder de processamento, e também permite a modificação de seus arquivos para atender as especificações de projetos de sistemas embarcados específicos e para poder ser utilizado por compiladores que originalmente não dão suporte ao *FreeRTOS* (BARRY, 2010), como é o caso do *mikroC*.

Segundo Barry (2010), o *FreeRTOS* possui, entre suas funções e características:

- Abstração das informações de tempo, o que ajuda a facilitar a estruturação de aplicações, simplificando-a e deixando-a com menos dependências;

- Maior facilidade para fazer manutenção e expandir aplicações;

- Modularização (habilidade de separar as aplicações em módulos);

- Facilidade de desenvolvimento em times;
- Facilidade para testes de aplicações;
- Reuso de código aprimorado;
- Maior eficiência, já que a utilização de um sistema operacional embarcado permite que aplicações sejam totalmente dirigidas por eventos;
- Informações sobre o tempo de inatividade;
- Tratamento de interrupções de modo flexível;
- Suporte a processamento misto, ou seja, pode-se utilizar processamentos contínuos, periódicos e de eventos em uma mesma aplicação;
- Simplicidade no controle de periféricos;
- Operações cooperativas e operações preventivas;
- Suporte à filas
- Semáforos;
- Possibilidade de utilizar exclusões simultâneas;
- Checagem automática de pilhas (para evitar estouros);
- Suporte para licenciamento comercial das aplicações;
- Outros.

Além dessas características, segundo Barry (2010), o *FreeRTOS* também possui diversas funções de *API*, as quais podem ser divididas em quatro grupos diferentes conforme suas ações e reações à outras funções:

- Relacionadas à agendamento e à tarefas (possibilitam e facilitam processamento paralelo e pseudoparalelo);

- Relacionadas à filas (auxiliam na intercomunicação de processos, *threads* e/ou *task*s no registro de dados);

- Relacionadas à semáforos (possuem diversas utilidades, como exclusão mútua de processos, *thread*s e *task*s, sincronização de processos, *thread*s e *task*s, gerenciamento de recursos e contagemd e eventos);

- Relacionadas à contadores de tempo e à relógios.

As funções do *FreeRTOS* podem ainda ser classificadas de acordo com seu tipo de retorno, pois as funções possuem um prefixo de uma ou mais letras que especifica o tipo de retorno da função. Por exemplo, o prefixo v significa *void* ou nulo, que quer dizer que a função não retornará nada para a outra função que a invocou (BARRY, 2010).

Para aproveitar a maioria dessas funções e características do *FreeRTOS*, é necessário utilizar tarefas, também denominadas como *task*s, que é uma das funções de maior importância para o entendimento do funcionamento e também para a utilização do *FreeRTOS* (BARRY, 2010).

2.1.3.1 ***TASK*S (TAREFAS)**

No *FreeRTOS*, as *task*s ou tarefas são implementadas na forma de funções normais, apenas com protótipo diferenciado, que deve receber e retornar *void* como parâmetro (BARRY, 2010).

Cada *task* é, de forma básica, um pequeno programa de *loop* infinito (execução infinita) e sem saída. Isto porque uma *task*, para não causa problemas, não devem executar além do limite ou fim da função e também não devem ter um retorno (comando *return*), sendo que se uma *task* não for mais necessária, a mesma deve ser excluída. Uma *task* pode, por outro lado, ser utilizada para criar diversas outras *task*s, desde que cada *task* possua um nome diferente (BARRY, 2010).

Cada *task*, seja ela criada diretamente ou indiretamente por outra *task*, possui pilha própria e uma própria cópia de variáveis definidas. As *task*s possuem dois estados possíveis, que são executando (quando um dos núcleos de processamento está executando seu código e suas instruções) e não executando (quando está pronta para ser executada, porém está esperando em uma fila ou esperando o resultado de outras *task*s) (BARRY, 2010).

A troca de *task*s em um núcleo de processamento é feito de modo que os dados, variáveis e informações, assim como o estágio do processamento, são salvos logo que uma *task* é interrompida, e são recarregados quando uma *task* retorna a ser executada por um núcleo. No *FreeRTOS*, para que haja compartilhamento e rotatividade das *task*s nos núcleos, é necessário utilizar a entidade denominada de *scheduler*, um agendador que gerencia esse processo e as *task*s (BARRY, 2010).

As *task*s são criadas através do uso da função de *API* "*FreeRTOSxTaskCreate()*", que é provavelmente a função mais complexa entre todas as funções de *API* do *FreeRTOS*, mas que também é o componente fundamental para sistemas embarcados multitarefas que utilizam o *FreeRTOS*, e por esse motivo é o primeiro a ser estudado e explicado no manual de referências do *FreeRTOS* (BARRY, 2010).

Segundo Barry (2010), os parâmetros desta função são:

1. Nome da função (que deve ser de *loop* infinito) que se tornará a *task*;

2. Nome descritivo para a função;

3. Tamanho em palavras da pilha. O valor verdadeiro da pilha é calculado com base neste valor e na largura da pilha (que é outro valor, configurado separadamente, que define o tamanho das palavras);

4. Parâmetros que devem ser repassados à *task* ao iniciar (utilizar NULL se nenhum parâmetro for necessário);

5. Prioridade da *task* (valores menores significam maiores prioridades);

6. Nome do manuseador de *task* (isso permite que a *task* possa ser alterada pelo manuseador e vice-versa).

Quando uma *task* é criada com sucesso, ela retorna uma mensagem de aviso, ou uma mensagem de erro caso contrário (BARRY, 2010).

É importante notar que *task*s funcionam como *thread*s, pois mesmo que trabalhem em conjunto para um aplicativo ou mesmo que compartilhem informações entre si, elas concorrem para utilizar os núcleos de processamento, utilizando as prioridades para obter vantagem nessa competição. Não o bastante, as *task*s possuem estados de execução, algo comum para processos e para *thread*s.

O Quadro 3.1 mostra um exemplo de programa contido no manual de referências do *FreeRTOS*.

```
/* FreeRTOS.org includes. */
#include "FreeRTOS.h"
#include "task.h"

/* Demo includes. */
#include "basic_io.h"

/* Used as a loop counter to create a very crude delay. */
#define mainDELAY_LOOP_COUNT            ( 0xffffff )

/* The task functions. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/*-----------------------------------------------------------*/

int main( void )
{
	/* Create one of the two tasks. */
	xTaskCreate(	vTask1,		/* Pointer to the function that implements the task. */
					"Task 1",	/* Text name for the task.  This is to
facilitate debugging only. */
					1000,		/* Stack depth – most small
microcontrollers will use much less stack than this. */
					NULL,		/* We are not using the task parameter. */
					1,		/* This task will run at priority 1. */
					NULL );		/* We are not using the task handle. */

	/* Create the other task in exactly the same way. */
	xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );


	/* Start the scheduler so our tasks start executing. */
	vTaskStartScheduler();

	/* If all is well we will never reach here as the scheduler will now be
	running.  If we do reach here then it is likely that there was insufficient
	heap available for the idle task to be created. */
	for( ;; );
	return 0;
}
/*-----------------------------------------------------------*/

void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

	/* As per most tasks, this task is implemented in an infinite loop. */
	for( ;; )
	{
		/* Print out the name of this task. */
		vPrintString( pcTaskName );

		/* Delay for a period. */
```

```
            for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
            {
                /* This loop is just a very crude delay implementation.  There is
                nothing to do in here.  Later exercises will replace this crude
                loop with a proper delay/sleep function. */
            }
        }
}
/*-----------------------------------------------------------*/

void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile unsigned long ul;

        /* As per most tasks, this task is implemented in an infinite loop. */
        for( ;; )
        {
            /* Print out the name of this task. */
            vPrintString( pcTaskName );

            /* Delay for a period. */
            for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
            {
                /* This loop is just a very crude delay implementation.  There is
                nothing to do in here.  Later exercises will replace this crude
                loop with a proper delay/sleep function. */
            }
        }
}
```

Quadro 3.1 – Exemplo de utilização de *Task*s (BARRY, 2010)

2.1.4 OPEN WATCOM

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

É uma útil ferramenta para executar os exemplos anexados ao manual de referência do *FreeRTOS*, o que auxilia à entender o funcionamento de *task*s no *FreeRTOS* (BARRY, 2010). Infelizmente, o *Open Watcom* não possui compatibilidade com o *Proteus*, motivo pelo qual foi usado apenas para estudos.

A Figura 3.4 apresenta uma tela de projeto do *Open Watcom*.

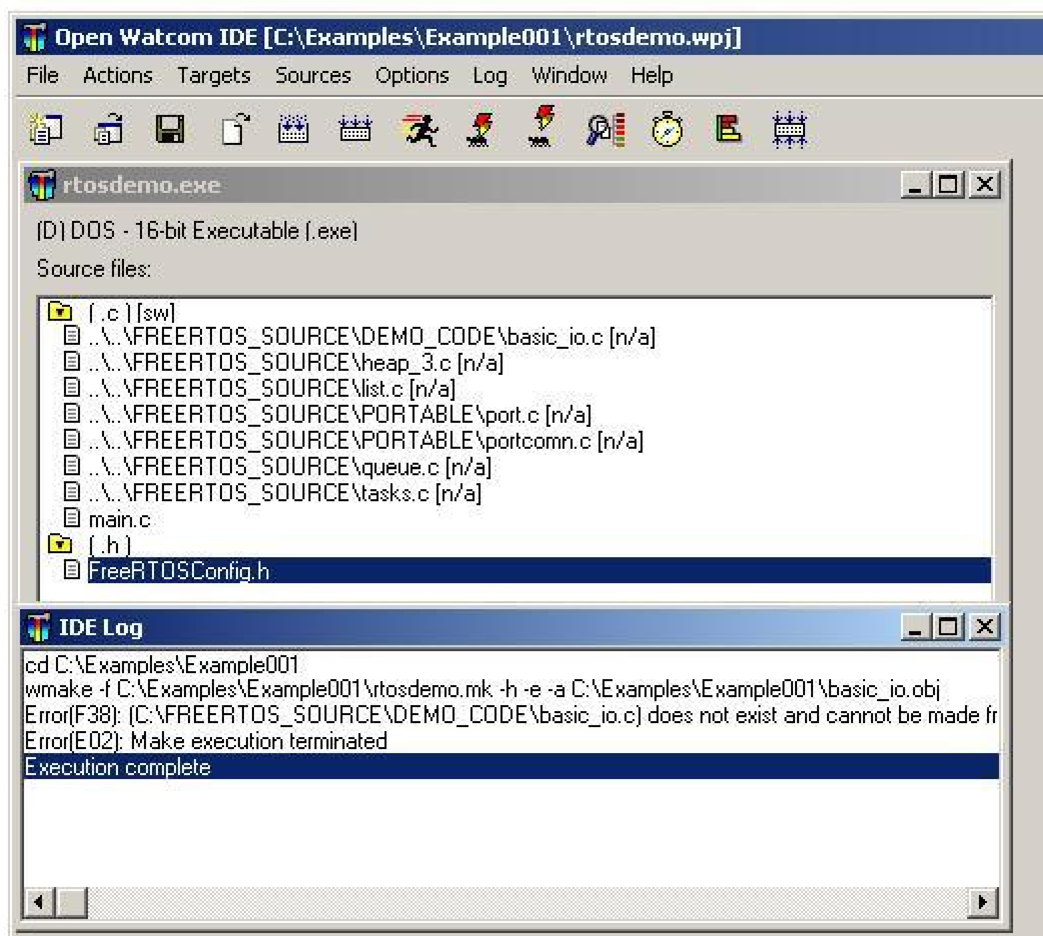

Figura 3.4 – Tela de um Projeto no *Open Watcom* (GALLASSI, 2011)

2.1.5 MPLAB

O *MPLAB* é um ambiente integrado de desenvolvimento, também conhecido como *IDE* (*Integrated Development Environment*) para microcontroladores desenvolvidos pela *Microship Technology* e que facilita o desenvolvimento de projetos com diversos arquivos. O *MPLAB* possui compatibilidade com o *FreeRTOS* e pode ser usado para compilar arquivos *hex* (MICROCHIP TECHNOLOGY, 2005), que por sua vez podem ser utilizados no *Proteus*.

O *MPLAB* é apenas uma *IDE*, isso quer dizer que precisa também de compiladores associados para poder funcionar (MICROCHIP TECHNOLOGY, 2005), como por exemplo, o *MPLAB C18*, que possui as características e configurações ideais para trabalhar com microcontroladores da família *PIC18*.

Apesar de oferecer compatibilidade com o *Proteus* e com o *FreeRTOS*, houve dificuldades em desenvolver experimentos funcionais com sua utilização em diferentes ambientes computacionais.

A Figura 3.5 apresenta a *IDE* do *MPLAB*.

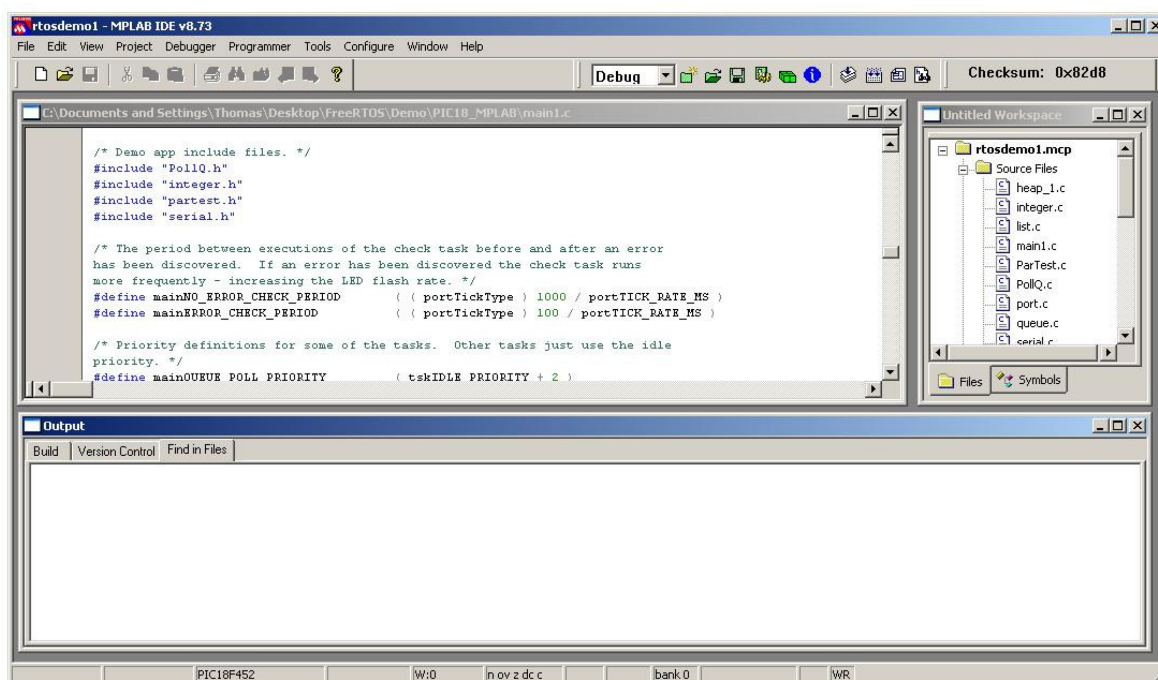

Figura 3.5 – *IDE* do *MPLAB* (GALLASSI, 2011)

2.2 EXPERIMENTOS REALIZADOS

Os experimentos deste projeto consistem em duas partes. A primeira, relacionada apenas ao desenvolvimento de sistemas embarcados sem sistemas operacionais, consiste em resgatar os experimentos da iniciação científica denominada “Um Estudo Exploratório Sobre Sistemas Operacionais Embarcados”, do mesmo autor deste projeto, e promover melhorias.

A segunda parte, relacionada ao desenvolvimento de sistemas embarcados com a utilização de sistemas operacionais, consiste em tentar desenvolver ou resgatar um programa sobre o *FreeRTOS*., e então desenvolver o mesmo programa sem o *FreeRTOS* e compará-los.

Em nota, as partes que estão em cinza nos quadros não fazem parte do código do programa propriamente dito, porém explicam com mais detalhes os códigos acima delas.

2.2.1 EXPERIMENTOS SEM SISTEMA OPERACIONAL EMBARCADO

Para esses, foram utilizados o *Isis* do *Proteus* na versão 7.4 SP3 (*Build* 6792) e o *mikroC* na versão 8.1.0.0, sendo que nenhum desses programas utiliza um sistema operacional.

2.2.1.1 TESTE DE MICROCONTROLADOR

A proposta original deste experimento foi explorar o *Proteus* e o *mikroC*, utilizando o microcontrolador PIC18F452. Para isso, foi montado um projeto o qual o microcontrolador está ligado a dois displays de sete segmentos, um do tipo anodo e outro do tipo cátodo, e faz contagem progressiva no superior e regressiva no inferior utilizando o limite de um digito.

Uma demonstração gráfica do projeto no Isis se encontra na Figura 3.6.

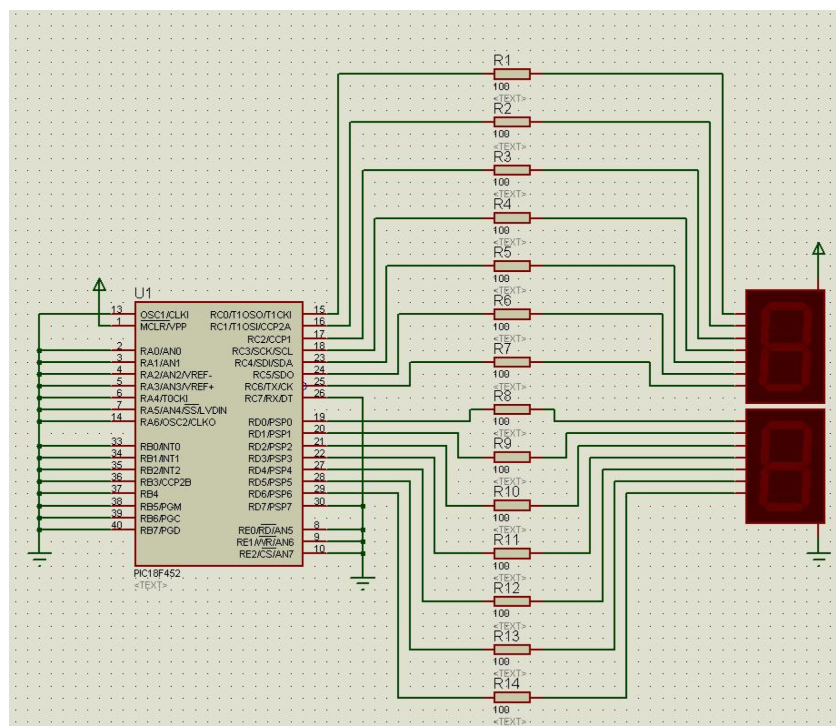

Figura 3.6 – Projeto de Teste de *Displays* e Microcontrolador (GALLASSI, 2011)

2.2.1.1.1 CODIGO DO PROGRAMA ORIGINAL

O Quadro 3.2 demonstra o código resgatado do programa de teste desenvolvido no *mikroC*. Não foram utilizadas funções extras, pois a intenção era apenas explorar o *mikroC* e o *Proteus*, ou seja, a codificação toda está inclusa na função principal ou *main* do programa, o que inclui o loop infinito, a inicialização e as configurações dos *port*s do microcontrolador e a sequência de incrementos e decrementos de números para os dois displays de sete segmentos, assim como um comando de espera para que um usuário humano possa checar as mudanças.

```
// Programa teste para display de 7 segmentos
void main() { // função principal, ou seja, o programa propriamente dito
```

A função *main* é a função principal e essencial em qualquer programa em linguagem C. Pode-se dizer que é o programa propriamente dito.

```
  TRISA = 0, PORTA = 0; // configuração dos ports
  TRISB = 0; PORTB = 0;
  TRISC = 0; PORTC = 0;
  TRISD = 0; PORTD = 0;
  TRISE = 0; PORTE = 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*. Nesse caso, os *ports* “A”, “B”, “C”, “D” e “E” são configurados para envio e recebimento de dados.

```
  while(1) { // loop infinito para que o programa não pare
```

O comando *while* é um comando de retorno condicional, ou seja, o programa ou função retorna ao ponto de início 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” em programação e deixa o programa com uma condicional sempre verdadeira, criando o *loop* infinito recomendado para programas embarcados.

```
    PORTC.RC0 = 0; // configura os pinos do port C para mostrar 0 no display
    PORTC.RC1 = 0;
    PORTC.RC2 = 0;
    PORTC.RC3 = 0;
    PORTC.RC4 = 0;
    PORTC.RC5 = 0;
    PORTC.RC6 = 1;
```

Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Juntos, esses comandos devem desenhar um “0” em um *display* de sete segmentos do tipo anodo.

```
    PORTD.RD0 = 1; // configura os pinos do port D para mostrar 9 no display
    PORTD.RD1 = 1;
    PORTD.RD2 = 1;
    PORTD.RD3 = 1;
    PORTD.RD4 = 0;
    PORTD.RD5 = 1;
    PORTD.RD6 = 1;
```

Cada um desses comandos modifica um pino em específico do *port* “D” de um microcontrolador. Juntos, esses comandos devem desenhar um “9” em um *display* de sete segmentos do tipo cátodo.

Código	Explicação
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar	Este comando faz com que o núcleo de processamento espere 1000 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.

PORTC.RC0 = 1; // configura os pinos do port C para mostrar 1 no display
PORTC.RC3 = 1;
PORTC.RC4 = 1;
PORTC.RC5 = 1;
	Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “1” em um *display* de sete segmentos do tipo anodo.

PORTD.RD4 = 1; // configura os pinos do port D para mostrar 8 no display	Esse comando modifica um único pino do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esse comando deve desenhar um “8” em um *display* de sete segmentos do tipo cátodo.
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar	Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.

PORTC.RC0 = 0; // configura os pinos do port C para mostrar 2 no display
PORTC.RC2 = 1;
PORTC.RC3 = 0;
PORTC.RC4 = 0;
PORTC.RC6 = 0;
	Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente n *port*, esses comandos devem desenhar um “2” em um *display* de sete segmentos do tipo anodo.

PORTD.RD3 = 0; // configura os pinos do port D para mostrar 7 no display
PORTD.RD4 = 0;
PORTD.RD5 = 0;
PORTD.RD6 = 0;
	Cada um desses comandos modifica um pino em específico do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “7” em um *display* de sete segmentos do tipo cátodo.

Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar	Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.

PORTC.RC2 = 0; // configura os pinos do port C para mostrar 3 no display
PORTC.RC4 = 1;
	Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “3” em um *display* de sete segmentos do tipo anodo.

PORTD.RD1 = 0; // configura os pinos do port D para mostrar 6 no display
PORTD.RD3 = 1;
PORTD.RD4 = 1;
PORTD.RD5 = 1;
PORTD.RD6 = 1;

Cada um desses comandos modifica um pino em específico do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “6” em um *display* de sete segmentos do tipo cátodo.
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar
Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.

PORTC.RC0 = 1; // configura os pinos do port C para mostrar 4 no display
PORTC.RC3 = 1;
PORTC.RC5 = 0;

Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “4” em um *display* de sete segmentos do tipo anodo.
PORTD.RD4 = 0; // configura os pinos do port D para mostrar 5 no display
Esse comando modifica um único pino do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esse comando deve desenhar um “5” em um *display* de sete segmentos do tipo cátodo.
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar
Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.

PORTC.RC0 = 0; // configura os pinos do port C para mostrar 5 no display
PORTC.RC1 = 1;
PORTC.RC3 = 0;

Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “5” em um *display* de sete segmentos do tipo anodo.

PORTD.RD0 = 0; // configura os pinos do port D para mostrar 4 no display
PORTD.RD1 = 1;
PORTD.RD3 = 0;

Cada um desses comandos modifica um pino em específico do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “4” em um *display* de sete segmentos do tipo cátodo.
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar
Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.
PORTC.RC4 = 0; // configura os pinos do port C para mostrar 6 no display
Esse comando modifica um único pino do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esse comando deve desenhar um “6” em um *display* de sete segmentos do tipo anodo.

```
PORTD.RD0 = 1; // configura os pinos do port D para mostrar 3 no display
PORTD.RD3 = 1;
PORTD.RD5 = 0;
```

Cada um desses comandos modifica um pino em específico do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “3” em um *display* de sete segmentos do tipo cátodo.

```
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar
```

Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.

```
PORTC.RC1 = 0; // configura os pinos do port C para mostrar 7 no display
PORTC.RC3 = 1;
PORTC.RC4 = 1;
PORTC.RC5 = 1;
PORTC.RC6 = 1;
```

Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “7” em um *display* de sete segmentos do tipo anodo.

```
PORTD.RD2 = 0; // configura os pinos do port D para mostrar 2 no display
PORTD.RD4 = 1;
```

Cada um desses comandos modifica um pino em específico do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “2” em um *display* de sete segmentos do tipo cátodo.

```
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar
```

Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.

```
PORTC.RC3 = 0; // configura os pinos do port C para mostrar 8 no display
PORTC.RC4 = 0;
PORTC.RC5 = 0;
PORTC.RC6 = 0;
```

Cada um desses comandos modifica um pino em específico do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “8” em um *display* de sete segmentos do tipo anodo.

```
PORTD.RD0 = 0; // configura os pinos do port D para mostrar 1 no display
PORTD.RD2 = 1;
PORTD.RD3 = 0;
PORTD.RD4 = 0;
PORTD.RD6 = 0;
```

Cada um desses comandos modifica um pino em específico do *port* “D” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um “1” em um *display* de sete segmentos do tipo cátodo.

```
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar
```

Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.

```
PORTC.RC4 = 1; // configura os pinos do port C para mostrar 9 no display
```

Esse comando modifica um único pino do *port* “C” de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esse comando deve desenhar um “9” em um *display* de sete segmentos do tipo anodo.

PORTD.RD0 = 1; // configura os pinos do port D para mostrar 0 no display
PORTD.RD3 = 1;
PORTD.RD4 = 1;
PORTD.RD5 = 1;

Cada um desses comandos modifica um pino em específico do *port* "D" de um microcontrolador. Considerando as mudanças feitas anteriormente no *port*, esses comandos devem desenhar um "0" em um *display* de sete segmentos do tipo cátodo.
Delay_ms(1000); // espera de 1000 ciclos de processamento antes de continuar
Novamente, Este comando faz com que o núcleo de processamento espere 1000 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.
} // fim do *loop*, o que o reinicia, reiniciando também o processo
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.
} // fim da função principal
Essa chave marca o fim do programa. O programa não deve chegar até essa marca em condições normais.

Quadro 3.2 – Código de Teste de um Microcontrolador e *Display*s (GALLASSI, 2011)

2.2.1.1.2 PROPOSTAS DE MELHORIAS

Por ser apenas um projeto de teste, o programa não precisa estar tão otimizado a ponto de termos que controlar pino por pino. Para melhor visualização, o ideal seria controlar os *port*s como um todo, já que os seus pinos estão conectados sequencialmente aos *display*s, o que permite a manipulação como uma variável não negativa de 8 *bit*s (que vai de 0 a 255). Podemos ainda criar variáveis e funções separadas para os displays anodo e cátodo, melhorando ainda mais a visualização. O código para essa proposta se encontra no Quadro 3.3.

// Programa teste para display de 7 segmentos aprimorado int disp7seg(int unidade) { // função de ativação de pinos para um display de 7 segmentos do tipo cátodo
Uma função que deve receber um valor numérico do tipo inteiro e retornar outro valor numérico do tipo inteiro. Ela foi desenvolvida para verificar um digito e retornar a conversão dele para um valor que o represente display de sete segmentos do tipo cátodo conectado de forma apropriada à um port de microcontrolador.

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

Esse comando é do tipo condicional, ou seja, ele toma decisões baseando-se em cálculos e comparações de valores. O valor que é verificado nesse caso é o valor recebido como digito.

```
    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 um valor comparado aos que o digito pode assumir. Para cada caso possível, é realizada uma conversão para um formato legível a um display de sete ou oito segmentos do tipo cátodo, e depois enviada como resposta à chamada de função.

```
    default: return(0);
  }
}
```

Caso o digito não seja válido ou legível, é retornado o valor 0, o qual não mostra nada em um display de sete oi oito segmentos do tipo cátodo.

```
void main()
{
```

A função *main* é a função principal e essencial em qualquer programa em linguagem C.

```
  int anodo = 0;
  int catodo = 9;
```

Duas variáveis do tipo inteiras, iniciadas com 0 e 9 respectivamente, e que serão usadas como os contadores progressivo e recessivo respectivamente.

```
  TRISA = 0, PORTA = 0;
  TRISB = 0; PORTB = 0;
  TRISC = 0; PORTC = 0;
  TRISD = 0; PORTD = 0;
  TRISE = 0; PORTE = 0;
```

Este conjunto de variáveis é utilizado para configurar os *port*s do microcontrolador, definindo o estado inicial, além das 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.

```
  while(1)
  {
```

O comando *while* é um comando de retorno condicional, ou seja, o programa ou função retorna ao ponto de início 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” em programação e deixa o programa com uma condicional sempre verdadeira, criando o *loop* infinito recomendado para programas embarcados.

```
    PORTC = ~disp7seg(anodo); // o ~ representa o comando not (não) em binário, invertendo os valores d todos os bits. Como um anodo trabalha de forma oposta a um cátodo, a inversão binária de um resulta no mesmo valor obtido em outro
```

Essa chamada de função no *port* “C” deve converter o digito e apresentar no display de sete segmentos apropriado. É utilizado o ~ para causar uma inversão binária de 8 *bit*s, o que permite a apresentação do resultado dessa função em um *display* de sete ou oito segmentos anodo.

```
    PORTD = disp7seg(catodo);
```

Essa chamada de função no *port* “D” deve converter o digito e apresentar no display de sete segmentos apropriado.

Delay_ms(1000);
Este comando faz com que o núcleo de processamento espere 1000 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.
anodo++;
Incrementa 1 à variável contador referente ao *display* anodo.
catodo--;
Decrementa 1 à variável contador referente ao *display* cátodo.
if (anodo >= 10) { anodo = 0; }
Verifica se a variável referente ao contador anodo ultrapassou o limite de um digito e a reinicia se for o caso.
if (catodo <= -1) { catodo = 9; }
Verifica se a variável referente ao contador cátodo ultrapassou o limite de não se tornar negativa e a reinicia se for o caso.
}
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, pois a condicional definida será sempre verdadeira.
}
Esta chave marca o fim do programa. O programa não deve chegar até esta marca em condições normais.

Quadro 3.3 – Código de Teste Aprimorado (GALLASSI, 2012)

2.2.1.2 SENSOR DE TEMPERATURA

Este experimento inclui botões de ligar e desligar, um interruptor para o caso de emergência, dois *LED*s para indicar se a temperatura está excessivamente alta ou baixa, um sensor de temperatura e quatro displays de segmentos que demonstram a temperatura. Para simplificar os circuitos também foram utilizados *inputs* e *outputs*, os quais basicamente transferem o sinal obtido entre si.

A Figura 3.7 apresenta o projeto que fora desenvolvido no *Isis*.

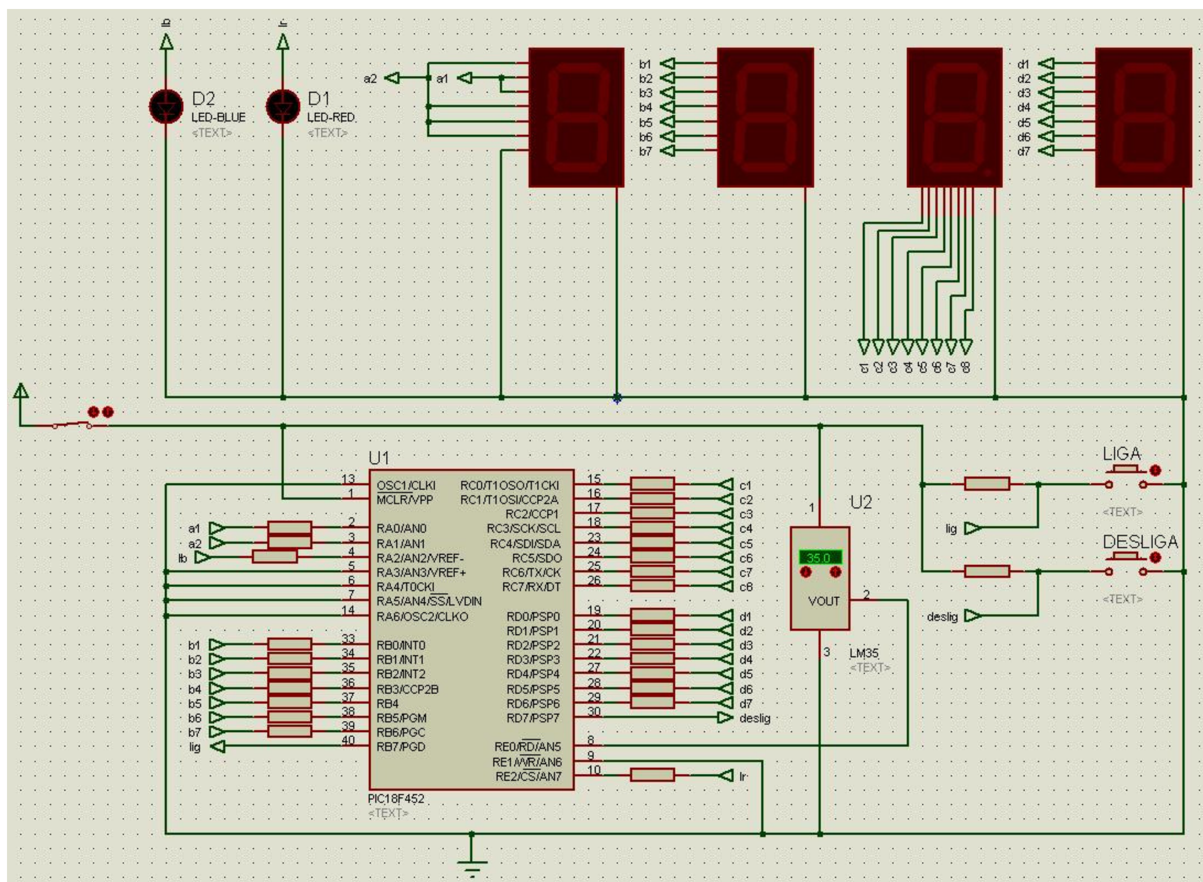

Figura 3.7 – Sistema Embarcado com Sensor de Temperatura (GALLASSI, 2011)

2.2.1.2.1 CÓDIGO DO PROGRAMA ORIGINAL

Como a proposta para o segundo experimento foi desenvolver gradualmente, além de explorar sistemas embarcados, procurou-se utilizar formas mais legíveis e menos repetitivas de desenvolvimento, como o emprego de funções extras, ou seja, a função *main* possui majoritariamente chamadas de funções, mas ainda possui as declarações de variáveis, o *loop* infinito, a inicialização e as configurações dos *port*s do microcontrolador. Em especial, foi utilizada uma única função para converter dígitos para cada *display* de sete ou oito segmentos, e uma função para converter os dados capturados e enviados pelos sensores de temperatura.

O Quadro 3.4 apresenta o código resgatado do programa de sensor de temperatura desenvolvido no *mikroC*.

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 desenvolvida com o intuito de checar se um entre dois botões definidos foi ou está pressionado, retornando um *flag* ou valor de sinalização que será utilizado 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
Este 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 *port*s desligados do microcontrolador.
F (Button(&PORTD, 7, 2000, 0)) return (0);
Este 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 retornado é verificado e usado para desligar os *port*s ligados do microcontrolador.

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

Caso nenhum botão tenha sido pressionado, esta 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. Neste 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. Esta função foi construída com o objetivo de verificar os dados emitidos por um sensor de temperatura e convertê-los para o formato de ºC (graus Celsius).

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

Este comando usa uma fórmula para converter um valor analógico (neste caso recebido do sensor de temperatura) para o formato digital e retorna o valor para a função que pediu a verificação (neste 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 digito e retornar a conversão deste digito 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
{

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

```
   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 destes comandos é uma condição a qual o digito 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 digito não seja legível ou válido, é retornado um valor que não apresenta 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
}
```

Este comando condicional está verificando se a temperatura atual informada ultrapassou ou não o limite informado, retornando um *flag* ou 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, esta 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
}
```

Este comando condicional está verificando se a temperatura atual informada é inferior ou não ao limite informado, retornando um *flag* ou valor de sinalização que deve ser interpretado pelo programa.

```
void desligar() { // função para desligar/hibernar os ports
```

Esta 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;
}

Este 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, estes não são hibernados e continuam funcionando normalmente.
void main() { // função principal
A função principal e estencial do programa. Devido às mudanças no estilo de programação desde o último estudo de caso, neste programa ela majoritariamente invoca as outras funções.
int inttemp10, centena, ligado = 0; // declaração de variáveis
Declaração de variáveis para manipular as funções. A primeira variável é usada para armazenar o valor da temperatura (multiplicado por 10 para fins de cálculos e comparações), a segunda variável é utilizada para calcular e manipular a centena dos valores de temperatura recebido, já que este necessitou um tratamento especial devido ao limite de pinos do microcontrolador, e a terceira variável é utilizada como *flag* ou sinalizadora 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;

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.
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;

Este 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 houveste conflito. Este 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 temperatura está alta
Esta chamada de função 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
Esta chamada de função no *port* “B” deve converter o digito de dezena da temperatura e demonstrar no *display* de sete segmentos apropriado.
PORTC = disp7seg(inttemp10 % 100 / 10) + 128; // calcula unidades
Esta chamada de função no *port* “C” deve converter o digito 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 neste 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
Esta chamada de função no *port* “D” deve converter o digito do decimal da temperatura e demonstrar no *display* de sete segmentos apropriado.
}
Esta 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 neste 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.
}
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, pois a condicional definida será sempre verdadeira.
}
Esta chave marca o fim do programa. O programa não deve chegar até esta marca em condições normais.

Quadro 3.4 – Código de Sistema Embarcado de Sensor de Temperatura (GALLASSI, 2011)

2.2.1.2.2 PROPOSTAS DE MELHORIAS

Adicionar um sistema de refrigeração, como ventiladores, ventoinhas, *cooler*s e dispositivos de ar condicionado, para o caso da temperatura aumentar muito e outro sistema de aquecimento para o caso de a temperatura abaixar muito seriam melhorias interessantes para o sistema, porém tais dispositivos não foram encontrados no *Isis* para melhorar o projeto, o que por sua vez impossibilita a codificação do mesmo.

2.2.1.3 SENSOR DE TEMPERATURA DUPLO

A proposta deste experimento foi de desenvolver um sistema embarcado que gerencie dois sensores de temperatura e demonstre seus resultados em dois monitores de cristal líquido ou *LCD* (*Liquid Crystal Display*), também conhecidos popularmente como *display*s de *LCD*. A Figura 3.8 apresenta o projeto desenvolvido no *Isis*.

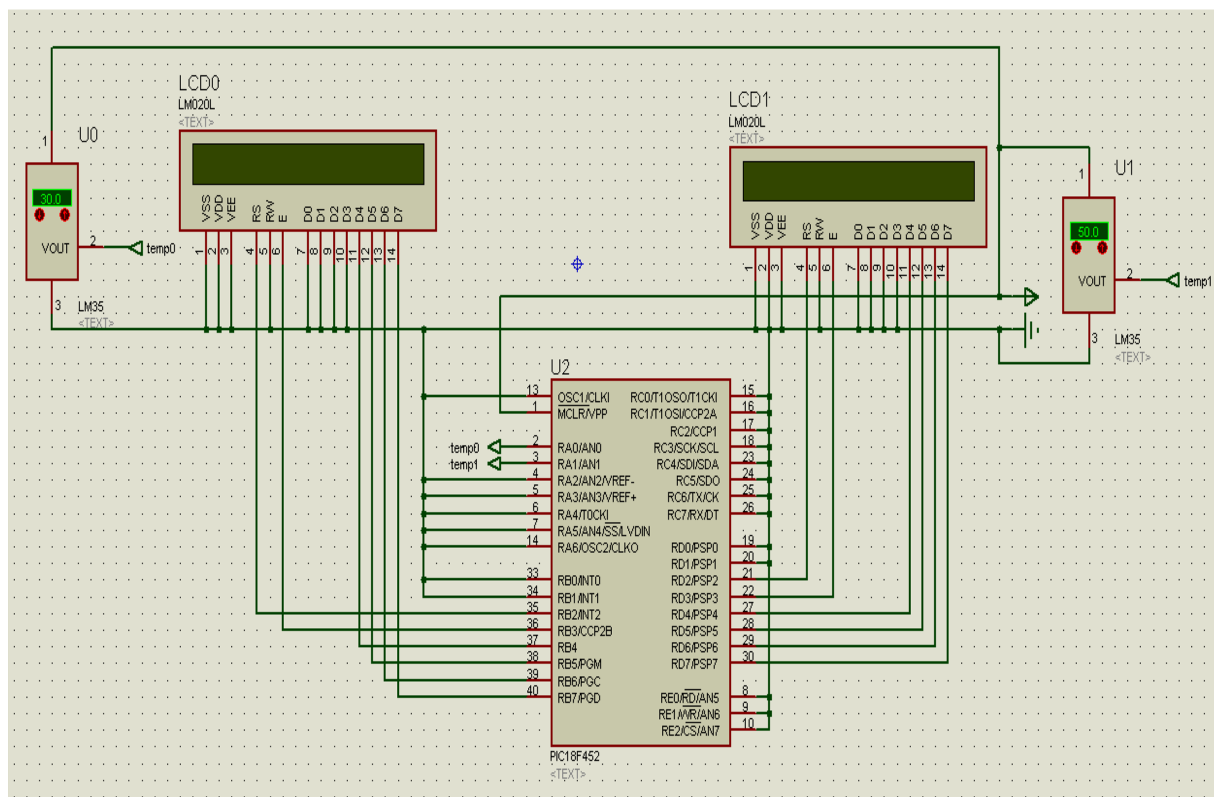

Figura 3.8 – Sistema Embarcado com Dois Sensores de Temperatura (GALLASSI, 2012)

2.2.1.3.1 CÓDIGO DO PROGRAMA ORIGINAL

Para este experimento, foram desenvolvidas duas funções além da função principal, uma para facilitar a coleta e conversão dos dados dos sensores de temperatura e outra função para imprimir os textos nos *display*s, cabendo à função principal fazer a conversão de valores números para textos e a ordenar a sequência de passos e instruções. O Quadro 3.5 apresenta o código resgatado no *mikroC* do programa de dois sensores de temperatura descrito.

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. Esta função foi construída com o objetivo de verificar os dados emitidos por um sensor de temperatura e convertê-los para o formato de ºC (graus Celsius).
return ((int)((5. * Adc_read(port) * 100.) / 1024 + 0.5)); }
Este comando usa uma fórmula para converter um valor analógico (neste caso recebido do sensor de temperatura) para o formato digital e retorna o valor para a função que pediu a verificação (neste caso, a função principal).
void printlcd(char text) { // função para imprimir em um display de LCD
Esta 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);
Este comando desativa o *cursor* do *display* de *LCD* ativo. Reusar este comando garante que o *cursor* continuará desligado e escondido.
LCD_Out(1,1,text);
Este 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); }
Este 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;
```

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.

```
LCD_Init(&PORTB); // inicia o LCD do port B
```

Este comando ativa e obtém controle do *display* de *LCD* ligado ao *port* “B”.

```
LCD_Cmd(LCD_CLEAR); // limpa o conteúdo do LCD
```

Este comando limpa qualquer texto que possa estar contido no *display* de *LCD* ativo, que deve ser o *display* ligado ao *port* “B”. O uso deste comando é o motivo pelo qual os *displays* de *LCD* já estão sendo ativados.

```
LCD_Cmd(LCD_CURSOR_OFF); // esconde o cursos do LCD
```

Este comando desativa o *cursor* do *display* de *LCD* ativo.

```
LCD_Init(&PORTD); // inicia o LCD do port D
```

Este comando ativa e obtém controle do *display* de *LCD* ligado ao *port* “D”, desligando qualquer outro *display* de *LCD* ativo.

```
LCD_Cmd(LCD_CLEAR); // limpa o conteúdo do LCD
```

Este comando limpa qualquer texto que possa estar contido no *display* de *LCD* ativo, que deve ser o *display* ligado ao *port* “D”. O uso deste comando é o motivo pelo qual os *displays* de *LCD* já estão sendo ativados.

```
LCD_Cmd(LCD_CURSOR_OFF); // esconde o cursos do LCD
```

Este comando desativa 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 destes 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
```

Este comando reinicia, ativa e obtém controle do display de *LCD* conectado ao *port* “B”, desligando-se do conectado ao *port* “D”.

```
printlcd(temp0); // imprime o conteúdo da variável no LCD
```

Este 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
Este comando reinicia, ativa e obtém controle do display de *LCD* conectado ao *port* “D”, desligando-se do conectado ao *port* “B”.
printlcd(temp1); // imprime o conteúdo da variável no LCD
Este comando invoca a função para escrever um texto no *display* de *LCD* ativo, enviando o valor guardado na primeira variável de texto.
}
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.

Quadro 3.5 – Código para dois Sensores de Temperatura e *LCD*s (GALLASSI, 2011)

2.2.1.3.2 PROPOSTAS DE MELHORIAS

A proposta de melhoria seria incluir dispositivos como botões para escolher entre os modelos de temperatura Kelvin, Celsius e Fahrenheit, um switch emergencial para desligar o sistema e dois botões para hibernar e religar o sistema. O projeto com as melhorias propostas está representado na Figura 3.9 e a codificação no Quadro 3.6.

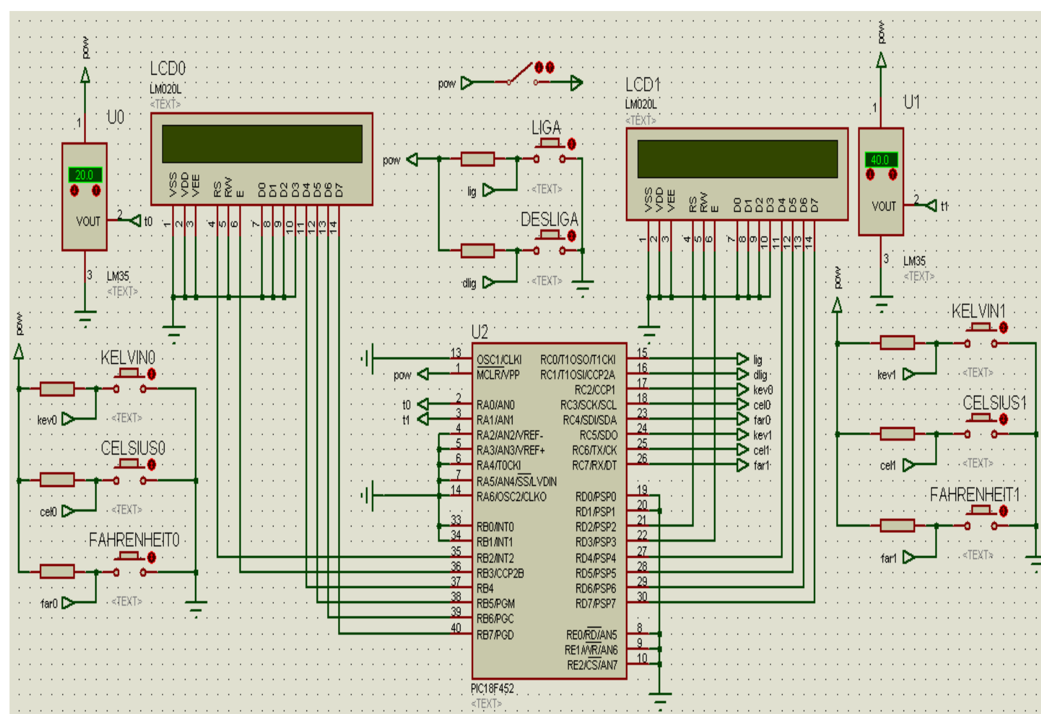

Figura 3.9 – Sistema com Dois Sensores de Temperatura Aprimorado (GALLASSI, 2012)

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 desenvolvida com o intuito de checar se um entre dois botões definidos foi ou está pressionado, retornando um *flag* ou valor de sinalização que será utilizado pela função principal do programa para ligar ou desligar os *port*s do microcontrolador e os *display*s de *LCD*. O valor recebido por essa função é o valor de sinalização que representa o estado atual dos *port*s, e é retornado caso nenhum botão tenha sido pressionado. Isto é feito para evitar e facilitar a manipulação dos *port*s. O programa inicia com o microcontrolador hibernando como padrão.
if (Button(&PORTC, 0, 2000, 0)) {
Este comando verifica se o botão conectado ao primeiro pino do *port* "C" está ou foi pressionado. Este botão referencia o religar do microcontrolador e dos *display*s de *LCD*.
LCD_Init(&PORTB); // inicia o LCD do port B
Este comando ativa e obtém controle do *display* de *LCD* ligado ao *port* "B".
LCD_Cmd(LCD_CLEAR); // limpa o conteúdo do LCD
Este comando limpa qualquer texto que possa estar contido no *display* de *LCD* ativo, que deve ser o *display* ligado ao *port* "B".
LCD_Cmd(LCD_CURSOR_OFF); // esconde o cursos do LCD
Este comando desativa o *cursor* do *display* de *LCD* ativo, que é o que está ligado ao *port* "B".
LCD_Init(&PORTD); // inicia o LCD do port D
Este comando ativa e obtém controle do *display* de *LCD* ligado ao *port* "D", desligando-se de qualquer outro *display* de *LCD* ativo.
LCD_Cmd(LCD_CLEAR); // limpa o conteúdo do LCD
Este comando limpa qualquer texto que possa estar contido no *display* de *LCD* ativo, que deve ser o *display* ligado ao *port* "D".
LCD_Cmd(LCD_CURSOR_OFF); // esconde o cursos do LCD
Este comando desativa o *cursor* do *display* de *LCD* ativo, que é o que está ligado ao *port* "D".
return (1); }
Retorna o calor de sinalização que referencia que o microcontrolador deve continuar realizando calculos.
if (Button(&PORTC, 1, 2000, 0)) {
Este comando verifica se o botão conectado ao segundo pino do *port* "C" está ou foi pressionado. Este botão referencia o hibernar do microcontrolador e dos *display*s de *LCD*.
LCD_Init(&PORTB); // inicia o LCD do port B
Este comando ativa e obtém controle do *display* de *LCD* ligado ao *port* "B". É necessário para que o microcontrolador possa então desligá-lo.
LCD_Cmd(LCD_TURN_OFF);
Desliga o display de LCD ao qual o microcontrolador está focado, que no caso deveria ser o microcontrolador do *port* "B". Isto é feito nesta função para que o microcontrolador não gaste ciclos tentando desligar algo já inativo.
LCD_Init(&PORTD);
Este comando ativa e obtém controle do *display* de *LCD* ligado ao *port* "D", descartando seu controle sobre qualquer outro display de *LCD*. É necessário para que o microcontrolador possa então desligá-lo.

LCD_Cmd(LCD_TURN_OFF);
Desliga o display de LCD ao qual o microcontrolador está focado, que no caso deveria ser o microcontrolador do *port* “D”. Isto é feito nesta função para que o microcontrolador não gaste ciclos tentando desligar algo já inativo.
return (0); }
Retorna o calor de sinalização que referencia que o microcontrolador deve hibernar, desligando os *port*s do microcontrolador e interrompendo os cálculos de temperatura.

return (estadoanterior); // retorna o estado anterior se nenhum dos botões tiverem sido pressionados
}

Caso nenhum botão tenha sido pressionado, esta função deve retornar o valor de sinalização recebido, simulando o pressionamento de um dos botões, para que o programa possa continuar executando.
void desligar() { // função para desligar/hibernar os ports
Esta 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;
}

Este 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, estes não são hibernados e continuam funcionando normalmente.
int tipolcd0(int tipo0) // função para checar botões de conversão de temperatura para primeiro LCD {
Similarmente à função referente aos botões de ligar e desligar, esta função deve receber um valor numérico do tipo inteiro e retornar outro valor numérico do tipo inteiro. Ela foi desenvolvida com o intuito de checar se um entre três botões definidos foi ou está pressionado, retornando um *flag* ou valor de sinalização que será utilizado pela função principal do programa para determinar o modelo de temperatura preferencial do usuário, seja Kelvin, Celsius ou Fahrenheit. O valor recebido por essa função é o valor de sinalização que representa o modelo atualmente utilizado, e é retornado caso nenhum botão tenha sido pressionado. Isto é feito para facilitar a manipulação dos *port*s. O programa utiliza o Kelvin como modelo inicial padrão.
if (Button(&PORTC, 2, 1000, 0)) return (0); // Kelvin
Este comando verifica se o botão, conectado ao terceiro pino do *port* “C”, está ou foi pressionado, retornando um valor utilizado para sinalização. Na função principal, o valor retornado é verificado e utilizado para fazer a conversão da temperatura obtida de um dos sensores para o modelo Kelvin (K).
if (Button(&PORTC, 3, 1000, 0)) return (1); // Celsius
Este comando verifica se o botão, conectado ao quarto pino do *port* “C”, está ou foi pressionado, retornando um valor utilizado para sinalização. Na função principal, o valor retornado é verificado e utilizado para manter o cálculo de temperatura obtida de um dos sensores para o modelo Celsius (ºC).

if (Button(&PORTC, 4, 1000, 0)) return (2); // Fahrenheit
Este comando verifica se o botão, conectado ao quinto pino do *port* “C”, está ou foi pressionado, retornando um valor utilizado para sinalização. Na função principal, o valor retornado é verificado e utilizado para fazer a conversão da temperatura obtida de um dos sensores para o modelo Fahrenheit (ºF).

return (tipo0); // retorna o estado anterior se nenhum dos botões tiverem sido pressionados
}

Caso nenhum botão tenha sido pressionado, esta função deve retornar o valor de sinalização recebido, simulando o pressionamento de um dos botões, para que o programa possa continuar executando uma parte referente à um dos sensores de temperatura com o último modelo utilizado ou o modelo Kelvin se nenhum botão tiver sido pressionado ainda.
int tipolcd1(int tipo1) // função para checar botões de conversão de temperatura para segundo LCD
Similarmente à função anterior, esta função deve receber um valor numérico do tipo inteiro e retornar outro valor numérico do tipo inteiro. Ela foi desenvolvida com o intuito de checar se um entre três botões definidos foi ou está pressionado, retornando um *flag* ou valor de sinalização que será utilizado pela função principal do programa para determinar o modelo de temperatura preferencial do usuário, seja Kelvin, Celsius ou Fahrenheit. O valor recebido por essa função é o valor de sinalização que representa o modelo atualmente utilizado, e é retornado caso nenhum botão tenha sido pressionado. Isto é feito para facilitar a manipulação dos *port*s. O programa utiliza o Kelvin como modelo inicial padrão. Uma função diferente deve que ser utilizada devido aos diferentes pinos conectados aos botões.
if (Button(&PORTC, 5, 1000, 0)) return (0); // Kelvin
Este comando verifica se o botão, conectado ao sexto pino do *port* “C”, está ou foi pressionado, retornando um valor utilizado para sinalização. Na função principal, o valor retornado é verificado e utilizado para fazer a conversão da temperatura obtida de um dos sensores para o modelo Kelvin (K).
if (Button(&PORTC, 6, 1000, 0)) return (1); // Celsius
Este comando verifica se o botão, conectado ao sétimo pino do *port* “C”, está ou foi pressionado, retornando um valor utilizado para sinalização. Na função principal, o valor retornado é verificado e utilizado para manter o cálculo de temperatura obtida de um dos sensores no modelo Celsius (ºC).
if (Button(&PORTC, 7, 1000, 0)) return (2); // Fahrenheit
Este comando verifica se o botão, conectado ao oitavo pino do *port* “C”, está ou foi pressionado, retornando um valor utilizado para sinalização. Na função principal, o valor retornado é verificado e utilizado para fazer a conversão da temperatura obtida de um dos sensores para o modelo Fahrenheit (ºF).

return (tipo1); // retorna o estado anterior se nenhum dos botões tiverem sido pressionados
}

Caso nenhum botão tenha sido pressionado, esta função deve retornar o valor de sinalização recebido, simulando o pressionamento de um dos botões, para que o programa possa continuar executando uma parte referente à um dos sensores de temperatura com o último modelo utilizado ou o modelo Kelvin se nenhum botão tiver sido pressionado ainda.
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. Esta função foi construída com o objetivo de verificar os dados emitidos por um sensor de temperatura e convertê-los para o modelo Celsius.

return ((int)((5. * Adc_read(port) * 100.) / 1024 + 0.5)); }
Este comando usa uma fórmula para converter um valor analógico (neste caso recebido do sensor de temperatura) para o formato digital e retorna o valor para a função que pediu a verificação (neste caso, a função principal). O valor convertido é no formato de ºC.
void printlcd(char text) { // função para imprimir em um display de LCD
Esta 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);
Este comando desativa o *cursor* do *display* de *LCD* ativo. Reusar este comando garante que o *cursor* continuará desligado e escondido.
LCD_Out(1,1,text);
Este 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); }
Este comando faz com que o processador espere 200 ciclos de processamento antes de continuar a execução do programa.
int convert(int valor, int tipo) // função para checar botões de conversão de temperatura para segundo LCD {
Está função recebe dois valores inteiros e retorna outro valor inteiro. Ela foi feita com o intuito de receber o valor de temperatura em graus Celsius (ºC) e, com base no valor de sinalização, tenha ele mudado recentemente ou não, aplicar a conversão para Kelvin ou Fahrenheit ou ainda deixar no modeo Celsius.
switch (tipo) {
Esse comando é do tipo condicional, ou seja, ele toma decisões baseando-se em cálculos e comparações de valores. O valor que é verificado nesse caso é o valor de sinalização recebido.

case 0:
{
valor = valor + 273.15;
break;
}

Se o valor de sinalização for zero, é aplicada a conversão do valor de temperatura para o formato Kelvin.

case 2:
{
valor = (valor * 1.8) + 32;
break;
}

Se o valor de sinalização for um, o valor de temperatura deve continuar no formato Celsius, o qual já fora capturado e não é necessário haver alterações (por isso não é referenciado nesse conjunto de condições). Se o valor de sinalização for dois, é aplicada a conversão para o formato Fahrenheit.

```
  }
```

Fim do conjunto de condicionais.

```
  return (valor); // retorna o estado anterior se nenhum dos botões tiverem sido pressionados
}
```

É retornado o valor de temperatura, tenha sido ele convertido ou não, e encerrada a função.

```
void main() { // função principal
```

A função principal e essencial do programa.

```
  int tempint0, tempint1, tipo0 = 0, tipo1 = 0, ligado = 0;
```

Declarações de variáveis para manipulação das funções. As duas primeiras variáveis são utilizadas para armazenar o valor de temperatura capturado e depois convertido. As duas variáveis seguintes são usadas para armazenar um valor de sinalização que refere o formato de temperatura desejado, seja Kelvin, Celsius ou Fahrenheit, ambas iniciando com Kelvin como padrão. O último valor é outra variável de sinalização, responsável por indicar se os *port*s do microcontrolador e os *display*s de LCD devem estar hibernando ou não, começando com um valor que indicam que os mesmos devem iniciar hibernando.

```
  char temp0[17], temp1[17]; // variáveis para guardar e transferir texto para os displays
```

Mais 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;
```

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.

```
  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.

```
    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.

```
      tempint0 = calctemp(0);
      tempint1 = calctemp(1);
```

Estas invocações da mesma função são utilizadas para obter e armazenar os valores de temperatura de diferentes sensores em duas variáveis inteiras, utilizando o formato Celsius como padrão de obtenção.

```
tipo0 = tipolcd0(tipo0);
tipo1 = tipolcd1(tipo1);
```

Cada uma dessas funções verifica se um dos botões referentes à preferência do modelo de exibição de temperatura foi pressionado, mudando o valor de sinalização referente para o cálculo.

```
tempint0 = convert(tempint0, tipo0);
tempint1 = convert(tempint1, tipo1);
```

Cada um dos valores de temperatura é convertido para a preferência do usuário, através de uma função que utiliza o valor de temperatura obtido e um valor de sinalização.

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

Cada um destes 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). Nestes casos, o valor numérico e já convertido de temperatura são copiados à variáveis de texto..

```
LCD_Init(&PORTB); // reinicia o LCD do port B
```

Este comando reinicia, ativa e obtém controle do display de *LCD* conectado ao *port* “B”, desligando-se do conectado ao *port* “D”.

```
printlcd(temp0); // imprime o conteúdo da variável no LCD
```

Este 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
```

Este comando reinicia, ativa e obtém controle do display de *LCD* conectado ao *port* “D”, desligando-se do conectado ao *port* “B”.

```
printlcd(temp1); // imprime o conteúdo da variável no LCD
```

Este comando invoca a função para escrever um texto no *display* de *LCD* ativo, enviando o valor guardado na primeira variável de texto.

```
}
```

Esta chave indica o fim da condicional para caso o microcontrolador esteja ativo. É importante notar que toda a coleta de dados, cálculos e comparações e exibições relacionados aos sensores de temperatura são feitos apenas neste caso.

```
else desligar();
```

Caso o microcontrolador e os *display*s de *LCD* devam estar hibernando, toda a coleta de dados, cálculos, comparações e exibições 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.

```
}
```

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.

Quadro 3.6 – Código para dois Sensores de Temperatura Aprimorado (GALLASSI, 2012)

2.2.2 EXPERIMENTOS COM SISTEMA OPERACIONAL EMBARCADO

Como houve problemas na utilização do *MPLAB* para realizar experimentos com o *FreeRTOS* no *Isis*, novas alternativas foram pesquisadas. Uma delas foi as modificações de Milinkovic (2012) no *FreeRTOS* para funcionar sob o *mikroC PRO for ARM*. Suas modificações incluem mudanças em algumas bibliotecas do *FreeRTOS*, como a *queue.c* e a *tasks.c*, seguido por um programa de testes que está na versão 0.0.0.4.

Os seus experimentos foram resgatados, e o primeiro deles foi utilizado no experimento descrito na subseção 2.2.2.1.

2.2.2.1 *LED*s Piscantes

Para este experimento, foi utilizada a versão 0.0.0.1 do programa de testes de Milinkovic (2012) e replicado seus resultados para um projeto sem utilização de sistema operacional embarcado, utilizando o *mikroC 8.1.0.0* para programação. O *mikroC PRO for ARM 2.50* foi utilizado para compilar o projeto original, já que o mesmo foi desenvolvido para o microcontrolador *ARM STM32 M3*. Em nota, o *Isis* do *Proteus* na versão 7.4 SP3 (*Build* 6792) não possui suporte nativo para este microcontrolador (porém algumas bibliotecas podem ser usadas para adicionar suporte a este e à outros microcontroladores, ou então pode-se usar a ferramenta de simulação diferente). A Figura 3.10 apresenta tal projeto implementado, independente de compilador e microcontrolador (apesar de que há restrições dependendo da configuração individual de cada pino em cada microcontrolador).

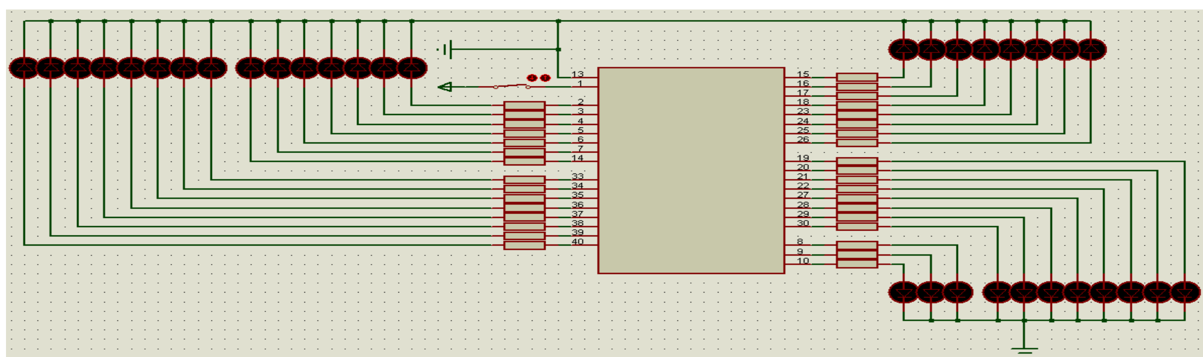

Figura 3.10 – Projeto de *LED*s Piscantes (GALLASSI, 2012)

O Quadro 3.7 apresenta o código para este projeto sem a utilização de um sistema operacional embarcado, em um microcontrolador da família *PIC*.

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.

Quadro 3.7 – *LED*s Piscantes sem Sistema Operacional Embarcado (GALLASSI, 2012)

Já o Quadro 3.8 apresenta o programa de Milinkovic (2012) para um microcontrolador *ARM STM32 M3*.

```
#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.

Quadro 3.8 – *LED*s Piscantes com Sistema Operacional Embarcado (GALLASSI, 2012)

Comparando os Quadros 3.7 e 3.8, podemos perceber que o sistema embarcado com sistema operacional é mais complexo e requer mais memória, mas também é mais maleável à mudanças, o que facilita sua extensão, como Milinkovic (2012) fez com as versões conseguintes, e a utilização do programa como base à outros programas, facilitando a padronização de projetos futuros. Ainda assim, o sistema embarcado sem sistema operacional não é tão exigente quanto à memória e ao poder de processamento do microcontrolador, evidenciado não só pelo código, mas também por não necessitar de bibliotecas adicionais, e isto é um diferencial muito importante em projetos de baixo custo.

É notável também que desenvolver utilizando o *FreeRTOS* é consideravelmente diferente, especialmente com utilização de *task*s. Em uma equipe de desenvolvimento ou empresa em migração para um sistema operacional embarcado, talvez um treinamento seja necessário.

Além disso, Milinkovic (2012) necessitou fazer alterações no *FreeRTOS* para que funcionasse adequadamente, o que implica que, mesmo que alterações não sejam necessárias aos programas, um conhecimento adicional sobre sistemas operacionais embarcados é necessário para sua utilização, e também demonstra que sistemas operacionais embarcados possuem incompatibilidades com certos ambientes de desenvolvimento.

3 CONSIDERAÇÕES FINAIS

Presentes em praticamente todas as áreas de atuação humana (CARRO, 2003), mais de 95% dos sistemas computacionais produzidos são sistemas embarcados, o que demonstra sua importância para o mercado e para a tecnologia atual (SIFAKIS, 2012). Ainda assim, seus projetos são restritos fisicamente, o que acarreta em restrições de consumo de energia, potencia de processamento e disponibilidade de memória, além das exigências de velocidade, segurança e confiabilidade serem altas devido às exigências do mercado (CARRO, 2003).

Os sistemas operacionais embarcados ajudam na padronização dos projetos e aumentam a vida útil dos produtos, mas também requerem mais recursos computacionais (evidenciados pela utilização de bibliotecas adicionais nos experimentos), o que acarreta em maiores custos com dispositivos físicos e pode não compensar lucrativamente às empresas em certos casos (CARRO, 2003).

O experimento com sistema operacional embarcado demonstra ainda que o estilo de programação necessita mudar ao implementar um sistema operacional embarcado e que talvez seja necessário treinar as equipes de desenvolvimento em uma migração, mas também demonstra que o código torna-se mais maleável e que, assim como nas descrições de Friedrich (2009) sobre diversos sistemas operacionais embarcados, que eles trazem características normalmente ausentes mas úteis em aplicações embarcadas, como a concorrência entre processos e a política de prioridades. Isto facilita bastante a programação conforme a quantidade de funções que o programa deve realizar escala seguindo o aumento de complexidade do projeto, o que leva a crer que talvez os únicos casos em que um projeto de sistema embarcado não se beneficiará da utilização de um sistema operacional são nos casos de o programa planejado ser demasiado simples e sem expectativa de expansões futuras.

Complementarmente, um sistema operacional embarcado é complexo (TANENBAUM, 2003), e suas implicações devem sempre ser estudadas antes da implantação em um projeto, caso contrário, o sistema operacional pode ser mal implantado, com consequências como desperdício de recursos físicos, financeiros e

computacionais como velocidade de processamento e disponibilidade de memória (CARRO, 2003).

3.1 CONCLUSÃO

O objetivo geral de estudar as vantagens e desvantagens da utilização de sistemas operacionais embarcados foi concluído, e foi averiguado que as vantagens superam as desvantagens na maioria dos casos, mas não no caso de o projeto de sistema embarcado ser demasiado simples, pois a utilização de sistemas operacionais embarcados ajuda a padronizar os projetos, diminuindo o tempo de projeto e aumentando a vida útil do produto, mas também requer mais energia, poder de processamento e memória, mesmo quando feito para impactar minimamente essas áreas.

Segurança e confiabilidade não são muito impactadas pela utilização de um sistema operacional embarcado, e velocidade é minimamente impactada, até porque os sistemas operacionais embarcados atualmente disponíveis conseguem cumprir exigências de sistemas de tempo real.

Os objetivos específicos também foram atendidos, pois foram estudadas as características e a história de sistemas computacionais e sistemas operacionais, incluindo aplicações embarcadas e sistemas operacionais embarcados, foram desenvolvidos diversos experimentos e tudo isso foi utilizado para averiguar as hipóteses.

Entre as hipóteses confirmadas ou não, temos que:

- A hipótese a. (A utilização de sistemas operacionais embarcados no desenvolvimento de aplicações embarcadas simplifica o processo de desenvolvimento) é falsa, pois um sistema operacional pode tanto simplificar como aumentar a complexidade de um projeto;

- A hipótese b. (A utilização de sistemas operacionais embarcados no desenvolvimento de aplicações embarcadas é viável às empresas

apenas para projetos complexos) foi confirmada, já que o sistema de prioridades de processos disponível na maioria dos sistemas operacionais embarcados e a maneabilidade advinda disso facilitam a programação de projetos complexos;

- A hipótese c. (A utilização de sistemas operacionais embarcados no desenvolvimento de aplicações embarcadas pode acelerar o processo de desenvolvimento) foi confirmada, pois, quando corretamente implantado e estudado, faz parte do conjunto de táticas e ferramentas de padronização, o que diminui o tempo de projeto, aumenta a vida útil e maximiza os lucros de um produto embarcado;

- A hipótese d. (Aplicações embarcadas que utilizam sistemas operacionais podem ser mais lentas do que as que não utilizam) é parcialmente verdadeira, pois isto acontece apenas quando o projeto é mal planejado, já que os sistemas operacionais embarcados possuem, em sua maioria condições de atender restrições de sistemas de tempo real;

- A hipótese e. (Aplicações embarcadas que utilizam sistemas operacionais podem ser mais inseguras do que as que não utilizam) é parcialmente verdadeira, pois o projeto se torna mais complexo e pode adquirir algumas brechas ao implantar um sistema operacional embarcado, mas isto não quer dizer que os desenvolvedores de sistemas operacionais embarcados não possuem tais preocupações. De fato, é possível que uma aplicação embarcada sem sistema operacional embarcado venha a ser problemática neste aspecto;

- A hipótese f. (Aplicações embarcadas que utilizam sistemas operacionais podem ser menos confiáveis do que as que não utilizam) é parcialmente verdadeira, pois o projeto se torna mais complexo e pode adquirir algumas brechas ao implantar um sistema operacional embarcado, mas isto não quer dizer que os desenvolvedores de sistemas operacionais embarcados não possuem tais preocupações.

> De fato, é possível que uma aplicação embarcada sem sistema operacional embarcado venha a ser problemática neste aspecto.

Além disso, algumas constatações foram feitas que não foram contemplados pelos objetivos e pelas hipóteses, como a possibilidade de equipes de desenvolvimento precisarem de treinamentos em uma migração, que as ferramentas de desenvolvimento de sistemas embarcados podem ser incompatíveis com a utilização de sistemas operacionais embarcados e que produtos embarcados possuem vida útil no mercado muito curta devido às constantes evoluções tecnológicas. Temos ainda que sistemas tradicionais diferem consideravelmente de sistemas embarcados, e o mesmo pode ser dito entre sistemas operacionais tradicionais e sistemas operacionais embarcados, mas que muito há a se ganhar com utilização de tecnologia embarcada.

3.2 PROPOSTAS DE TRABALHOS FUTUROS

Para trabalhos futuros, algumas sugestões foram elaboradas, cada qual pode ser trabalhada independente das demais:

- Focar o funcionamento do *FreeRTOS* sob o *mikroC*, estudando as modificações de Milinkovic (2012) e propondo melhorias;
- Trabalhar e estudar com o *FreeRTOS* oficial, sem modificações, em outros compiladores e ferramentas;
- Procurar experimentar e estudar outros sistemas operacionais embarcados;
- Aprofundar ainda mais a bibliografia e realizar experimentos mais diversificados, a fim de tornar mais material sobre sistemas embarcados e sobre sistemas operacionais embarcados disponível;
- Procurar estudos de caso e aplicar os conhecimentos adquiridos através deste trabalho;

- Adicionar detalhes sobre *System-in-Package* (*SiP*);

- Aprofundar sobre a utilização de sistemas embarcados para uma área específica, como jogos, robótica, equipamentos hospitalares, automação residencial etc..

REFERÊNCIAS BIBLIOGRÁFICAS

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

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **Citação**: NBR-10520/ago - 2002. Rio de Janeiro: ABNT, 2002.

_______. **Referências**: NBR-6023/ago. 2002. Rio de Janeiro: ABNT, 2002.

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-and-book.html>. Acesso em: 10 jul. 2010.

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.

CARRO, L.; WAGNER, F. R. **Sistemas Computacionais Embarcados**. Anais das Jornadas de Atualização em Informática da SBC. Cap. 2, 2003.

FRIEDRICH, L. F. **A Survey of Operating Systems Infrastructure for Embedded Systems**. Relatório Técnico. Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal, 2009.

GALLASSI, T. T. **Um Estudo Exploratório Sobre Sistemas Operacionais Embarcados**. Relatório de Iniciação Científica. Americana: Faculdade de Tecnologia de Americana, ago. de 2011.

KONANA, P.; RAY G. **Physical Product Reengineering With Embedded Information Technology**. Artigo Técnico. Revista Communications of the ACM. Vol. 50, nº 10. Outubro, 2007.

LABCENTER ELETRÔNICS; **Proteus Design Suíte Product Guide**. Guia Online. Disponível em: <http://downloads.labcenter.co.uk/proteus7brochure.pdf>. Acesso em: 10 jul. 2011.

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**. 3ª ed. São Paulo: Atlas, 1996.

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.

MICROCHIP TECHNOLOGY **MPLAB® IDE User's Guide**. Manual Técnico Digital. 2005. Disponível em: <http://ww1.microchip.com/downloads/en/devicedoc/51519a.pdf>. Acesso em: 1 jul. 2011.

MIKROELETRONIKA **mikroC User's manual**. Manual Técnico Digital. 2006. Disponível em: <http://www.mikroe.com/pdf/mikroC/mikroC_manual.pdf>. Acesso em: 10 jul. 2011.

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.

OPEN WATCOM COMMUNITY; SYBASE **Open Watcom 1.9 C/C++ Getting Started Help**. Manual de *Software*, 2010. Seção "Introduction to Open Watcom C/C++".

SEVERINO, A. J. **Metodologia do Trabalho Científico**. 23ª ed. São Paulo: Cortez. 2007, p. 199-209.

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.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª ed. Tradução de Ronaldo A. L. Gonçalves et al. São Paulo: Prentice Hall. 2003, p. 1-48.

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

BIBLIOGRAFIA

BARRY, R., **The FreeRTOS™ Reference Manual**. Manual Técnico Digital, Versão 1.2.0, Real Time Engineers Ltd., 2011. Disponível em: <http://www.freertos.org/Documentation/FreeRTOS-documentation-and-book.html>. Acesso em: 12 jul. 2011.

CARRO, L.; WAGNER, F. R.; **Metodologias e Técnicas de Engenharia de Software para Sistemas Embarcados**, Capítulo 2. Jornadas de Atualização em Informática da SBC, 2003.

TAURION, C. **Software Embarcado**: A Nova Onda da Informática. Rio de Janeiro: Brasport Livros e Multimídia Ltda. 2005.