



Dinesh Atul Rodrigues Trivedi

**DESENVOLVIMENTO E SIMULAÇÃO DE SOFTWARE DE
CONTROLE PARA UM PROTÓTIPO DE BOMBA DE
INFUSÃO DE INSULINA BASEADOS EM
MICROCONTROLADOR DA FAMÍLIA PIC**

São José dos Campos, SP

Dinesh Atul Rodrigues Trivedi

DESENVOLVIMENTO E SIMULAÇÃO DE SOFTWARE DE CONTROLE PARA UM PROTÓTIPO DE BOMBA DE INFUSÃO DE INSULINA BASEADOS EM MICROCONTROLADOR DA FAMÍLIA PIC

Trabalho de conclusão de curso apresentado ao
Instituto de Ciência e Tecnologia – UNIFESP,
como parte das atividades para obtenção do tí-
tulo de Bacharel em Ciência da Computação.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência e Tecnologia

Bacharelado em Ciência da Computação

Orientador: Prof. Dr. Luiz Eduardo Galvão Martins

São José dos Campos, SP

Julho de 2014

Dinesh Atul Rodrigues Trivedi

DESENVOLVIMENTO E SIMULAÇÃO DE SOFTWARE DE CONTROLE PARA UM PROTÓTIPO DE BOMBA DE INFUSÃO DE INSULINA BASEADOS EM MICROCONTROLADOR DA FAMÍLIA PIC

Trabalho de conclusão de curso apresentado ao
Instituto de Ciência e Tecnologia – UNIFESP,
como parte das atividades para obtenção do tí-
tulo de Bacharel em Ciência da Computação.

Trabalho aprovado em 01 de Julho de 2013:

Prof. Dr. Luiz Eduardo Galvão Martins
Orientador

Prof. Dr. Tiago de Oliveira
Convidado 1

Prof. Dr. Fabiano C. Paixão
Convidado 2

São José dos Campos, SP
Julho de 2014

Este trabalho é dedicado ...

Agradecimentos

Escreva aqui os agradecimentos ...

*“Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2)*

Resumo

Estima-se que o Diabetes Melito (DM) já afeta 246 milhões de pessoas em todo mundo. A estimativa é que aumente para 380 milhões até 2025, sendo que no Brasil esse número chegue à aproximadamente 7 milhões. Seu tratamento adequado, na maioria das vezes, é o uso contínuo da bomba de infusão de insulina, entretanto é inacessível a grande parte da população, devido ao seu alto custo, aproximadamente R\$ 14.000,00. Sendo assim, este projeto tem como objetivo o desenvolvimento de um sistema embarcado crítico de controle de protótipo de bomba de infusão de insulina, baseado no microcontrolador da família PIC, PIC18F452. Neste trabalho utilizou-se o compilador MikroC que dá suporte aos periféricos necessários através de uma biblioteca completa em C. O desenvolvimento utilizou conceito de OOC, *Object Orientated Programming in ANSI-C*, para desacoplamento de todos os módulos existentes. Esse conceito possibilitou o uso de *Design Patterns*, ambos contribuíram para aumentar a capacidade de expandir e dar manutenção ao *software* de forma mais simples. Os testes do software foram realizados através do simulador Proteus, o que possibilita uma grande facilidade para o depuramento. O circuito montado nesse simulador contém todos os componentes reais necessários para o desenvolvimento do protótipo. E, além disso, optou-se pelo uso do Proteus devido as grandes vantagens e minimização de problemas de integração que um simulador proporciona. Por fim esse projeto serve como uma prova de conceito para o problema abordado, demonstrando a evolução e problemas encontrados durante o desenvolvimento. Servindo como base para projetos futuros.

Palavras-chaves: PIC, Microcontrolador, Sistema embarcado, Sistema crítico, Motor de passo, OOC.

Abstract

It is estimated that diabetes mellitus (DM) already affects 246 million people around world. It is estimated to increase to 380 million by 2025, and in Brazil this the number will reach about 7 million. Adequate treatment in most times, is the continuous use of insulin pump therapy, however it is inaccessible to a large part of the population due to its high cost approximately R\$ 14,000.00. Thus, this project aims to develop an embedded system critical control prototype insulin infusion pump based microcontroller family PIC, PIC18F452. In this study we used the MikroC compiler that supports the required peripherals through a complete library in C. The development concept used OOC *emph* Object Orientated Programming in ANSI-C for decoupling of all the modules. This concept allowed the use of *emph* Design Patterns, both contributed to increase the ability to expand and maintain the *emph* software in a simpler way. The tests were performed using the software simulator Proteus, which allows a great facility for the depuration. The circuit used in simulator contains all the components necessary for real prototype development. Also, Proteus was chosen because of the great benefits and minimize integration problems that a simulator provides. Finally this project serves as a proof of concept for the addressed problem, demonstrating the progress and problems encountered during development. Serving as a base for future projects.

Key-words: PIC, Microcontroler, Embedded System, Critical System, Step Motor, OOC.

Lista de ilustrações

Figura 1 – Imagens de uma bomba de infusão de insulina	25
Figura 2 – Exemplo de Sistemas embarcados	32
Figura 3 – Sistemas embarcados críticos	33
Figura 4 – Forças normais e tangenciais	37
Figura 5 – Visão paralela e perpendicular das <i>stacks</i> e rotor	38
Figura 6 – Interconexão das bobinas	39
Figura 7 – Relação das interconexões das bobinas	39
Figura 8 – <i>Stator single stack</i>	40
Figura 9 – Motor híbrido	40
Figura 10 – Tabelas de pinos LCD comum	42
Figura 11 – PIC18F452	45
Figura 12 – Estrutura Interna PIC18F452	46
Figura 13 – Pinos PIC18F452	47
Figura 14 – Modelo de contexto da bomba de infusão de insulina	53
Figura 15 – Caso de uso do módulo de interface com o usuário	54
Figura 16 – Caso de uso do módulo de controle de infusão	55
Figura 17 – Caso de uso do módulo de monitoramento dos sensores	55
Figura 18 – Estrutura do padrão <i>Factory Method</i>	60
Figura 19 – <i>Software</i> Proteus	60
Figura 20 – Estrutura do <i>Software</i>	61
Figura 21 – Diagrama de classe GlobalConfig	62
Figura 22 – Diagrama de classe InsulinPump	63
Figura 23 – Diagrama de classe Lcd	63
Figura 24 – Diagrama de classe Menu	64
Figura 25 – Diagrama de classe Motor	65
Figura 26 – Diagrama de classe TimerMotoror	65
Figura 27 – Placa Microgenios PIC18F4452	68

Lista de tabelas

Lista de abreviaturas e siglas

μA	Microampère
ADC	<i>Analogic Digital Conversor</i>
A/D	<i>Analogic Digital</i>
CA	Corrente alternada
CC	Corrente contínua
CCP	<i>Capture / Compare / PWM</i>
DM	Diabetes Melito
EEPROM	<i>Electrically-Erasable Programmable Read-Only Memory</i>
I/O	Entrada/Saída
IDE	<i>Integrated Development Environment</i>
kHzs	<i>Kilohertz</i>
LCD	<i>Liquid Crystal Display</i>
LVD	<i>Low Voltage Detect</i>
mA	Miliampère
MCLR	<i>Master Clear or Reset</i>
MHz	<i>Megahertz</i>
MIPS	Milhões de Instruções por Segundo
MSSP	<i>Master Synchronous Serial Port</i>
OO	Orientado à Objetos
OOO	<i>Object Orientated Programming in ANSI-C</i>
PC	<i>Personal Computer</i>
PSP	<i>Parallel Slave Port</i>
PWM	<i>Pulse-Width Modulation</i>

RAM	<i>Random Access Memory</i>
ROM	<i>Read-Only Memory</i>
TTL	<i>Transistor-transistor Logic</i>
V	Volts

Lista de símbolos

μ	micro
-------	-------

Sumário

1	Introdução	25
1.1	Motivação	27
1.2	Objetivos	27
1.2.1	Objetivos Secundários	27
1.3	Procedimentos Metodológicos	27
1.4	Organização do Trabalho	28
2	SISTEMAS EMBARCADOS	31
2.1	SISTEMAS EMBARCADOS DE TEMPO	32
2.2	SISTEMAS EMBARCADOS CRÍTICO	33
2.3	MICROCONTROLADOR	34
2.3.1	PIC	34
2.3.2	SENSORES	34
2.3.3	SENSORES BIOLÓGICOS	35
2.3.4	ATUADORES	35
2.3.5	MOTOR DE PASSO	36
2.3.6	MOTOR DE PASSO RELUTÂNCIA VARIÁVEL – <i>MULTI STACK</i>	36
2.3.6.1	<i>DESIGN</i> DO MOTOR	38
2.3.7	MOTOR DE PASSO RELUTÂNCIA VARIÁVEL – <i>SINGLE STACK</i>	39
2.3.8	MOTOR DE PASSO HÍBRIDO	40
2.3.9	COMPARAÇÃO ENTRE TIPOS DE MOTORES	41
2.3.10	LCD - <i>Liquid Crystal Display</i>	42
2.4	PROTEUS <i>DESIGN SUITE</i>	42
2.4.1	ISIS <i>SCHEMATIC CAPTURE</i>	43
2.5	MIKROC	44
2.6	PIC18F452	44
2.6.1	Estrutura Interna	46
2.6.2	DESCRIÇÃO DOS PINOS	47
3	Bomba de infusão de insulina	49
3.1	Contextualização	49
3.2	Características gerais da bomba de infusão	49
3.3	Aspectos de segurança da bomba de infusão	51
3.4	Requisitos da bomba de infusão	53
4	Módulo de controle de infusão de insulina	57

4.1	<i>OOO - Object Orientated Programming in ANSI-C</i>	57
4.2	<i>Design Patterns</i>	58
4.2.1	Usos de <i>Design Patterns</i>	58
4.2.2	<i>Factory Method</i>	59
4.3	Proteus	60
4.4	Arquitetura do Software	61
4.4.1	Módulo Config	61
4.4.2	Módulo InsulinPump	62
4.4.3	Módulo Lcd	62
4.4.4	Módulo Menu	64
4.4.5	Módulo Motor	64
4.4.6	Módulo TimerMotor	65
4.4.7	Módulo Principal	65
5	Análise e discussão dos resultados	67
5.1	Simulador	67
5.2	Arquitetura modularizada	68
6	Conclusão	71
	Referências	73
	Apêndices	77
	APÊNDICE A Config	79
A.1	Config.c	79
A.2	Config.h	81
	APÊNDICE B InsulinPump	83
B.1	IInsulinPump.h	83
B.2	InsulinPump.c	83
B.3	InsulinPump.h	86
	APÊNDICE C Lcd	87
C.1	lcd.h	87
C.2	lcdFactory.h	87
C.3	lcdFactory.c	88
C.4	lcd4bits2x16	88
C.4.1	lcd4bits2x16.h	88
C.4.2	lcd4bits2x16.c	88

APÊNDICE D Menu	93
D.1 IMenu.h	93
D.2 MenuConfigurar.h	93
D.3 MenuConfigurar.c	93
D.4 MenuExecutando.h	96
D.5 MenuExecutando.c	96
D.6 MenuInicial.h	98
D.7 MenuInicial.c	98
D.8 MenuReservatorio.h	100
D.9 MenuReservatorio.c	100
APÊNDICE E Motor	103
E.1 IMotorPasso.h	103
E.2 MotorPassoFactory.h	103
E.3 MotorPassoFactory.c	103
E.4 MotorProteus	104
E.4.1 MotorProteus.h	104
E.4.2 MotorProteus.c	104
APÊNDICE F TimerMotor	107
F.1 ITimerMotor.h	107
F.2 TimerMotorPasso.h	107
F.3 TimerMotorPasso.c	107
APÊNDICE G Principal	109
G.1 insulin_pump.c	109

1 Introdução

A Diabetes Melito é uma doença que surge quando o organismo deixa de produzir insulina ou quando essa passa a não atuar com a mesma eficácia. Atualmente existe duas classificações para ela: Diabetes Melito tipo 1 e 2. A primeira se caracteriza por ser autoimune que lesa, de forma irreversível, as células do pâncreas, produtoras de insulina e conhecidas como células beta, e seu diagnóstico se dá durante a infância do portador. Enquanto a segunda é consequência da resistência do próprio organismo contra as ações da insulina, o principal fator para se desenvolver essa resistência é a obesidade(PORTALDIABETES, 2008). Bomba de infusão de insulina é um pequeno aparelho eletrônico, do tamanho de um celular ou pager, que está ligado ao corpo do portador da doença por um finíssimo cateter com uma agulha flexível na ponta. Essa agulha é inserida no braço, coxa ou abdômen e deve ser trocada em um período de 2 ou 3 dias. Essa bomba não mede o índice glicêmico ou a quantidade de insulina a ser utilizada, essa medição é feita através do glicosímetro. A Figura 1¹ representa um aparelho comercial.



Figura 1 – Imagens de uma bomba de infusão de insulina

Seu funcionamento é bem simples, libera-se uma quantidade de insulina, programada pelo médico, durante o dia todo, simulando o funcionamento do pâncreas de uma pessoa saudável, entretanto existem cuidados a serem tomados: calcular a quantidade de carboidratos ingeridos a cada refeição e programar o aparelho para injetar uma quantidade de insulina com maior velocidade no organismo nos horários em que se faz as refeições principais. Quanto a quem pode usar, a pessoa deve cumprir alguns pré-requisitos que são:

- Conseguir medir o índice glicêmico no mínimo 4 vezes por dia;
- Durante a fase de adaptação e ajuste da dosagem a serem utilizadas pela bomba, fazer a medição glicêmica de 6 a 8 vezes por dia;
- Seguir as recomendações médicas além de manter contato e um constante *feedback* com os responsáveis pela bomba e, além de tudo, seguir a dieta recomendada, respeitando quantidades ingeridas;

¹ <http://www.diabetes.org.br/sala-de-noticias/2316-bombas-de-infusao-de-insulina>

- Ter condição financeira para custear o equipamento e o contato com os responsáveis por ele;
- Estar disposto ao uso da bomba durante o dia todo, 24 horas junto ao corpo;
- Aprender sobre contagem de carboidratos para saber seu consumo durante as refeições;
- Praticar exercícios.

Cumprindo os pré-requisitos citados temos as vantagens de seu uso que são:

- Maior flexibilidade no horário das refeições;
- Se usada corretamente o risco de hipoglicemia é reduzido, e a longo prazo as complicações devido ao diabetes também;
- Melhora o controle glicêmico;
- Melhora no controle do fenômeno do amanhecer, responsável pelo aumento do índice glicêmico durante a manhã, entre as 4 e 8 horas da manhã, causador da hipoglicemia se o diabético não calculou a dose de insulina antes de dormir, ou não se levantou durante a noite para gerenciá-la.

Mas mesmo com todas as vantagens dada devido ao uso do equipamento caso o diabético seja obeso, ingira grandes quantidades de alimento ou açúcar, ou seja, carboidratos, não praticar atividades físicas, não fazer a medição do índice glicêmico na quantidade de vezes recomendada, ou até mesmo determinar por si só a quantidade de insulina a ser utilizada, não existe vantagem no seu uso. É importante ter em mente que mesmo com toda facilidade e tecnologia existente o acompanhamento médico não deve ser deixado de lado. As principais indicações médicas para o uso do equipamento são:

- Fenômeno do amanhecer;
- Hipoglicemia;
- Diminuir a variação do índice glicêmico;
- Hiperglicemia;
- Recorrente cetose, que é o acúmulo de cetócidos, pois o fígado quebra a gordura e proteína devido à falta de insulina, pois o corpo não consegue utilizar a glicose como energia;
- Flexibilidade, especialmente para crianças pequenas;

- Gestação, viagens e atividade físicas;
- Fobia de injeção;
- Desejo do diabético ([DIABETES, 2013](#); [PORTALDIABETES, 2009](#)).

1.1 Motivação

Segundo a Sociedade Brasileira de Diabetes ([SBC, 2014](#)), diversos estudos realizados mostram que o tratamento feito através da Infusão de insulina tem diversas melhorias quando comparado com outros tratamentos existentes. Entretanto não é o mais utilizado devido ao seu alto custo, devido a importação. Logo, esse projeto vem com foco social: facilitar o acesso da população brasileira de baixa renda ao equipamento, melhorando sua qualidade de vida dos portadores da doença que se encaixem nesse perfil.

1.2 Objetivos

Esse projeto tem como objetivo principal desenvolver um software para um protótipo de uma Bomba de Infusão de insulina utilizando o microcontrolador da família PIC, PIC18F452.

1.2.1 Objetivos Secundários

Em segundo plano este trabalho foca em:

- Aprendizado sobre as características e funcionalidades disponíveis do microcontrolador escolhido;
- Aprendizado das tecnologias utilizadas como: compilador, simulador e bibliotecas disponíveis;
- Desenvolvimento das funcionalidades básicas de uma bomba de infusão de insulina;
- Aprendizado sobre a escolha e uso de um motor de passo;
- Aprendizado sobre a forma de uso de um *display* de LCD para comunicação com o usuário.

1.3 Procedimentos Metodológicos

Este trabalho foi dividido em duas etapas: pesquisa para levantamento de referências bibliográficas e desenvolvimento prático. Durante a primeira etapa, foi feita a pesquisa e levantamento bibliográfico sobre o tema em questão para que fosse possível um melhor entendimento

da plataforma, tecnologias e, claro, do problema abordado. Juntamente com essa pesquisa, foi feito um estudo sobre o microcontrolador PIC escolhido, PIC18F452, a partir do material encontrado. O estudo e desenvolvimento foi feito utilizando o compilador MikroC e sua IDE, já para os testes utilizou-se o simulador Proteus. Em seguida houve-se um levantamento de informações sobre sistemas embarcados, sistemas críticos e sistemas críticos de tempo real, e assim adquirindo-se conhecimento sobre os conceitos de desenvolvimento da área em questão, uma vez que a confiabilidade e segurança do problema proposto é importantíssimo.

O software de controle do protótipo da bomba de infusão de insulina foi desenvolvido na linguagem C, compilado para o microcontrolador já citado, PIC18F452. O sistema é responsável basicamente por gerenciar o perfil de infusão basal, calcular intervalos corretos e passos necessários para uma infusão. Desta forma, o software foi dividido nos seguintes módulos: Config, LCD, InsulinPump, Motor, Menu, TimerMotor e principal.

Essa divisão por módulos é, e foi, extremamente importante para o desenvolvimento do sistema. Somando-a à um dos conceitos mais importante desse projeto - OOC, *Object-Oriented Programming With ANSI-C* - são as chaves para a facilidade de manutenção, entendimento do código e possível evolução do projeto.

As responsabilidades dos módulos são bem claras e isoladas:

Config: Centralizar configurações gerais do sistema; LCD: Abstrair uso do periférico para o resto do sistema; InsulinPump: Abstrair funcionamento, requisitos de segurança e outras particularidades da bomba; Motor: Abstrai como se controla o motor e qual tipo está sendo utilizado para infusão; TimerMotor: Separa o gerenciador de tempo das particularidades únicas do hardware e compilador; Menu: Facilitar navegação entre menus, simulando máquina de estado; Principal: Possui o *loop* principal para navegação simples entre os menus existentes.

Os testes do software foram feitos através do simulador Proteus, desenvolvido pela Lab-center. Ferramenta tão importante quanto o OOC para o caminhar deste trabalho. Permite um alto nível de testes, devido a sua facilidade de uso, além do grande auxílio à depuração. O uso do simulador durante o desenvolvimento do software acarreta na minimização dos problemas de integração entre hardware e software, complicações estas que são praticamente impossíveis de estimar.

E, finalizando, após o desenvolvimento, executou-se uma bateria de testes para se o funcionamento da bomba estava sendo de acordo com a forma especificada, levando em conta toda a integração necessária com os periféricos existentes.

1.4 Organização do Trabalho

A organização desse trabalho tem como início considerações do estudo feito sobre sistemas embarcados. Assunto que abrange definições sobre tipos de sistemas embarcados, com-

ponentes como microcontroladores, sensores, atuadores e de interface de comunicações com o usuário. Em seguida, tem-se uma introdução do simulador Proteus, ferramenta de teste utilizada, em conjunto com o compilador. Para complementar as informações sobre o compilador, há uma explicação da organização do microcontrolador escolhido, como pinagem, contadores, barramentos e outros componentes. A partir dessas informações aborda-se o problema proposto, contextualizando-o, descrevendo requisitos e características.

Após a introdução do problema abordado e explicação de todas as tecnologias utilizadas, vem a descrição e organização do software, explicando cada módulo e forma de implementação. Assim sendo tem-se a discussão de resultados abordando detalhes, motivos e vantagens da implementação para que seja possível finalizar descrevendo a conclusão de todo estudo e projetos futuros.

2 SISTEMAS EMBARCADOS

Sistema embarcado em geral é uma combinação de *hardware* e *software* para executar uma tarefa específica diferente dos computadores do dia-a-dia que possuem inúmeros propósitos: verifica e-mail, escrever monografias, entre outros. Sistemas embarcados podem possuir ou não um sistema operacional, seja um RTOS, possui requisitos de tempo de execução, ou um Linux e, portanto, pode ser desenvolvido em um *hardware* com microcontrolador ou microprocessador ([WIKIBOOK, 2012](#)).

Segundo ([CUNHA, 2013](#)) a inteligência embarcada é uma tendência futura, cada vez mais inteligência será adicionada aos equipamentos do dia-a-dia, considera que um micro-ondas atual tem mais capacidade computacional do que tinha o projeto Apollo, que levou o homem a lua. Esta crescente utilização se dá basicamente pelo preço e consumo reduzido dos microcontroladores, além da grande flexibilidade ao atender os mais diversos problemas visto o vasto número de arquiteturas disponíveis: ARM, MIPS, Coldfire/68k, PowerPC, x86, PIC, 8051, Atmel AVR, Renesas H8, SH, V850, FR-V, M32R, Z80, Z8 e outras. Um contraste que atrai diversos desenvolvedores quando comparado com o número limitado de arquiteturas disponíveis para microprocessadores do mercado de computadores pessoais ([GERMANO, 2011](#)).

A comunicação dos microcontroladores com o meio externo, segundo ([GERMANO, 2011](#)), se dá pelos periféricos e o mais comuns são:

- Entrada de dados através de teclas, geralmente pelo de teclados feitos com varredura matricial;
- *Leds*;
- *Display's* de LCD, sendo os mais comuns os alfanuméricos como o HD44780;
- Interface serial, por exemplo RS232 e I2C;
- USB, *Universal Serial Bus*;
- TCP/IP.

Como dito anteriormente, esses sistemas estão cada vez mais no dia-a-dia das pessoas e, claro, facilitando a vida delas, mas muitas vezes não são percebidos. E cada vez mais estão mais acessíveis podendo automatizar funções até mesmo dentro das próprias casas. A Figura 2¹ mostra alguns sistemas embarcados e onde são utilizados:

¹ <http://bytesdontbite.com/2012/06/26/embedded-systems-no-bdb/>



Figura 2 – Exemplo de Sistemas embarcados

2.1 SISTEMAS EMBARCADOS DE TEMPO

O conceito Tempo Real é complexo para ser explicado, mas sua ideia básica é que se espera que o computador responda algo para o ambiente externo em tempo. Normalmente pessoas assumem que tempo real significa "muito rápido", entretanto não é verdade, tempo real simplesmente significa "rápido o suficiente" no contexto de operação do sistema. Um exemplo é a ação do motor, pode-se dizer que é "rápida", pois o sistema deve tomar decisões como - fluxo de combustível, tempo da faísca - toda vez que o motor completa um ciclo.

Sistemas de tempo real são baseados em previsibilidade e, segundo (FARINES; FRAGA; OLIVEIRA, 2000), essa previsibilidade de um sistema de tempo real é obtida quando independente de falhas, sobrecargas e variações de hardware, e assim é possível que seu comportamento seja antecipado antes de sua execução. Isso tem a finalidade de poder prever o funcionamento de um sistema de tempo real e garantir as suas restrições temporais, e para isso é necessário definir hipóteses em relação a carga e falhas em relação ao ambiente externo deste sistema (FARINES; FRAGA; OLIVEIRA, 2000). Segundo (MALL, 2009) os sistemas de tempo real são classificados em dois tipos:

- *Soft Real Time Systems*: Sistemas não críticos de tempo real, onde a ocorrência de uma falha temporal é da mesma ordem de grandeza que os resultados em que o funcionamento está correto, exemplos: Máquina de lavar e portão eletrônico de uma casa;
- *Hard Real Time Systems*: Sistemas Críticos de Tempo Real, onde a ocorrência de uma falha temporal complicam, e muito, os resultados quando comparado com seu funcionamento correto, exemplos: sistema de controle de um avião e um sistema de controle de semáforos.

2.2 SISTEMAS EMBARCADOS CRÍTICO

Sistema Crítico é um sistema no qual a confiança é fundamental, ou melhor, a questão mais importante em seu desenvolvimento. Isso porque sistemas críticos, em caso de falha, podem causar consequências gravíssimas para os humanos, economia e outras áreas. Pode-se dizer que seus indicadores são: Disponibilidade, confiabilidade, segurança e proteção. E para que essa confiança seja alcançada deve-se evitar erros durante seu desenvolvimento e realizar diversos testes para que seja possível detectar e corrigir os erros que passarem de forma que seja possível limitar os danos causados por falhar operacionais (SOMMERVILLE, 2004; FELDMANN et al., 2007; JORDAN, 2006).

Segundo (KOPETZ, 2011) as classificações dos sistemas embarcados críticos podem ser:

- *Fail Safe*: Classificação para sistemas onde o estado seguro pode ser atingido em caso de falha, como por exemplo, esgotar a bateria de uma bomba de insulina;
- *Fail Operational*: Classificação para sistemas que em caso de falhas ainda são capazes de fornecer algum tipo de serviço, mesmo que mínimo. Um exemplo é um sistema de controle de voo que, mesmo em caso de falha, é capaz de fornecer serviços e ser seguro.

Abaixo a Figura 3, mostra exemplos de sistemas embarcados críticos.



Figura 3 – Sistemas embarcados críticos

2.3 MICROCONTROLADOR

Microcontroladores são chips inteligentes que utilizam a arquitetura Harvard, RISC. É constituído basicamente por pinos de entradas e saídas e memória. Suas saídas podem ser controladas através de programação e em função do processamento de suas entradas. Sua programação pode ser feita em diversas linguagens como: C, C++, entre outras ([RADIOAMADORES, 2009](#)).

Segundo ([GANSSE, 1999](#)), o microcontrolador é a parte mais importante de um sistema embarcado e sua principal diferença quando comparada com um microprocessador é o fato de ser um sistema computacional completo que integra todos as principais partes da arquitetura de Von Neumann em um único componente, as partes citadas são:

- CPU: *Central Processor Unit*;
- Memória RAM: *Random Access Memory*;
- Portas I/O: Portas de entrada e saída.

Além de ser composto por temporizadores, memória ROM (*Read Only Memory*), conversor AD, analógico – digital, e DA, digital – analógico. Comparados com microprocessadores, os microcotroladores possuem consumo e *clock*, processamento, reduzidos, isso devido ao fato que o primeiro é destinado a tarefas que necessita uma alta capacidade de processamento como, por exemplo, os microprocessadores do nosso PC do dia-a-dia. Por padrão, os microprocessadores são utilizados em situações que os requisitos são abrangentes, com entradas e saída variadas como: sensores, atuadores e periféricos de comunicação ([LEE; SESHIA, 2011](#)).

2.3.1 PIC

PIC é um circuito integrado produzido pela *Microchip Technology Inc.* Seu nome significa: *Programmable Interface Controller*, Controlador de Interface Programável. Externamente possui uma aparência de um circuito integrado mais comuns - TTL ou CMOS -, mas na verdade contém todos os componentes de um sistema microprocessado como: CPU, *Central Processor Unit*, sua finalidade é interpretar as instruções de programa; Memória PROM, *Programmable Read Only Memory*, na qual memorizará as instruções do programa; Memória RAM, *Random Access Memory*, utilizada pra memorizar as variáveis do programa; Linhas de I/O, entrada e saída, para controlar dispositivos internos e receber informações do meio externo; entre outros ([RADIOAMADORES, 2009](#); [WIKIPEDIA, 2012b](#)).

2.3.2 SENSORES

A definição de sensor pode ser a de um transdutor capaz de alterar sua característica física interna em resposta à um fenômeno físico externo. Além disso, existem sensores consi-

derados de operação indireta que são os quais alteram suas propriedades como capacitância, resistência, ou, até mesmo, sua indutância, sob ação de algum gradeza ou evento externo (RO-SÁRIO, 2006).

Segundo (NOMADSUSP, 2012), os sensores são largamente utilizados na medicina, indústria, robótica, além de outras aplicações. Considerando que o sinal é sempre uma forma de energia, os sensores podem ser classificados em função da energia que é capaz de detectar, como:

- Sensores de luz: células solares, fotodíodos, fototransistores, tubos fotoelétricos, e outros;
- Sensores de som: microfones e hidrofone;
- Sensores de temperatura: termômetros e termopares;
- Sensores de resistência elétricas: ohmímetro.
- Outros;

2.3.3 SENSORES BIOLÓGICOS

Segundo (NOMADSUSP, 2012), os sensores citados anteriormente são corretamente chamados de sensores artificiais. Isto devido ao fato de existir sensores naturais ou biológicos, já que todos os organismos vivos possuem sensores capazes de agir da mesma forma que os artificiais. Esses sensores biológicos são células especializadas, sensíveis a:

- Luz, movimento, temperatura, vibração, pressão, campos elétricos, som, e outros aspectos físicos do ambiente;
- Grande variedade de moléculas ambientais, incluindo toxinas e nutrientes;
- Aspectos metabólicos, tais como os níveis de glicose e oxigênio;
- Até mesmo as diferenças entre proteínas do ambiente externo e do próprio organismo.

Esses sensores artificiais que imitam sensores biológicos, utilizando componentes biológicos, são chamados biossensores.

2.3.4 ATUADORES

Segundo (CHIRONIS; SCLATER, 1991), dispositivos considerados atuadores são aqueles que transformam uma forma de energia em outra, causando mudanças no ambiente em que estão atuando, ou seja, de acordo com sinais, ou impulsos, recebidos realizam ações capazes

de alterar as grandezas físicas do ambiente em questão. Eles são capazes de converter energias como: energia elétrica, hidráulica e pneumática em energia mecânica. Segue exemplos de alguns tipos de atuadores:

- Atuadores eletromagnéticos: São os motores elétricos como motores de passos, servos;
- Atuadores hidráulicos: Utilizam um fluido submetido a uma pressão para movimentar um braço, são utilizados em robô que operam grandes cargas;
- Atuadores pneumáticos: Utilizam um gás submetido a uma pressão para movimentar o braço, possuem menor custo que os hidráulicos, sendo utilizados em robôs de menor porte;

2.3.5 MOTOR DE PASSO

Motor de passo é um dispositivo eletromecânico. Sua principal propriedade é sua habilidade de transformar pulsos elétricos em movimentos, que são precisamente incrementados na posição do rotor e são denominados "passos". Esse tipo de motor é caracterizado como máquina duplamente saliente, significando que possui dentes, compostos por matérias magnéticos nas duas partes que o compõe: A parte imóvel chamada estator e a móvel rotor ([SANTOS, 2008](#); [ACARNLEY, 2002](#)).

Seu uso é interessante em situações em que precisão nos movimentos é necessária. Isso porque com ele é possível controlar: ângulo de rotação, velocidade, posição e sincronismo. Suas vantagens não são seu torque nem a capacidade de gerar movimentos de alta velocidade, mas sim a precisão em seus movimentos. Devido a essas características esse tipo de motor é amplamente utilizado em: câmeras de vídeo, robôs, brinquedos, *scanners*, impressoras, entre outros ([SANTOS, 2008](#)).

De forma simples o funcionamento de um motor de passo consiste no uso de materiais magnéticos, ou solenoides, como dito anteriormente, alinhados dois a dois, representando os polos norte e sul, que quando energizados atraem o rotor fazendo-o se alinhar as partes energizadas do estator, causando assim um pequeno movimento: o passo. Sua velocidade e sentido estão diretamente relacionados à forma com que os solenoides são acionados, o primeiro com a frequência e o segundo a ordem de acionamento ([SANTOS, 2008](#); [ACARNLEY, 2002](#); [WIKIPEDIA, 2012a](#)).

2.3.6 MOTOR DE PASSO RELUTÂNCIA VARIÁVEL – *MULTI STACK*

A fonte do fluxo magnético desse tipo de motor são as bobinas colocadas nos dentes do estator. O acionamento das bobinas é feito em sequência para incentivar o movimento, alinhamento, dos conjuntos de dentes sucessivos do estator e do rotor dando ao motor a característica de passos. Ao longo de seu eixo ele é dividido em seções isoladas magneticamente chamadas

stacks, daí o nome *multi stack*, e cada uma pode ser excitada por uma bobina separadamente chamada *phase*. Cada *stack* possui um estator, preso em sua posição pela caixa, suporte, do motor junto com as bobinas e o elemento móvel, rotor.

O rotor é uma unidade única e maciça que será utilizado para a movimentação da carga. O material do rotor é um metal elétrico laminado o que permite que o campo magnético possa mudar rapidamente sem grandes perdas. O estator de cada rotor possui um determinado número de polos e uma parte da *phase*, bobina, é enrolada em torno de cada polo para produzir o campo magnético. Os polos adjacentes são enrolados no sentido oposto assim os campos magnéticos adjacentes possuem sentidos opostos. Com isso o circuito magnético completo é considerado um polo do estator, o dente do rotor, o vão de ar entre os dentes de ambos e, por fim, um polo adjacente do estator. E esse circuito é repetido a cada par de polos do estator. As forças normais produzidas pelos polos do estator e os dentes do rotor são iguais e se anulam assim sobra apenas a força tangencial o que causa o movimento, isso pode ser visto conforme a Figura 4.

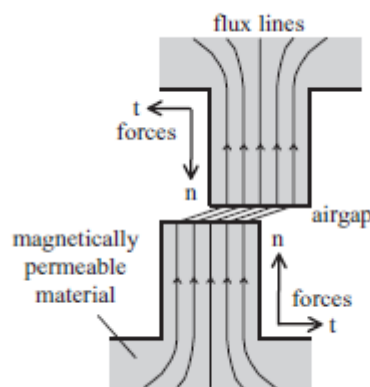


Figura 4 – Forças normais e tangenciais

A posição do rotor com relação ao estator é ajustada toda vez que as bobinas são excitadas. O ajuste ocorre, pois os dentes de ambos são alinhados o que tende a diminuir a relutância do circuito magnético, daí surgiu o nome do motor. Considerando a Figura 5 é possível perceber que para girar no sentido horário a ordem de acionamento deve ser A, B, C, A, B, C, A... e no sentido anti-horário A, C, B, A, C, B, A...

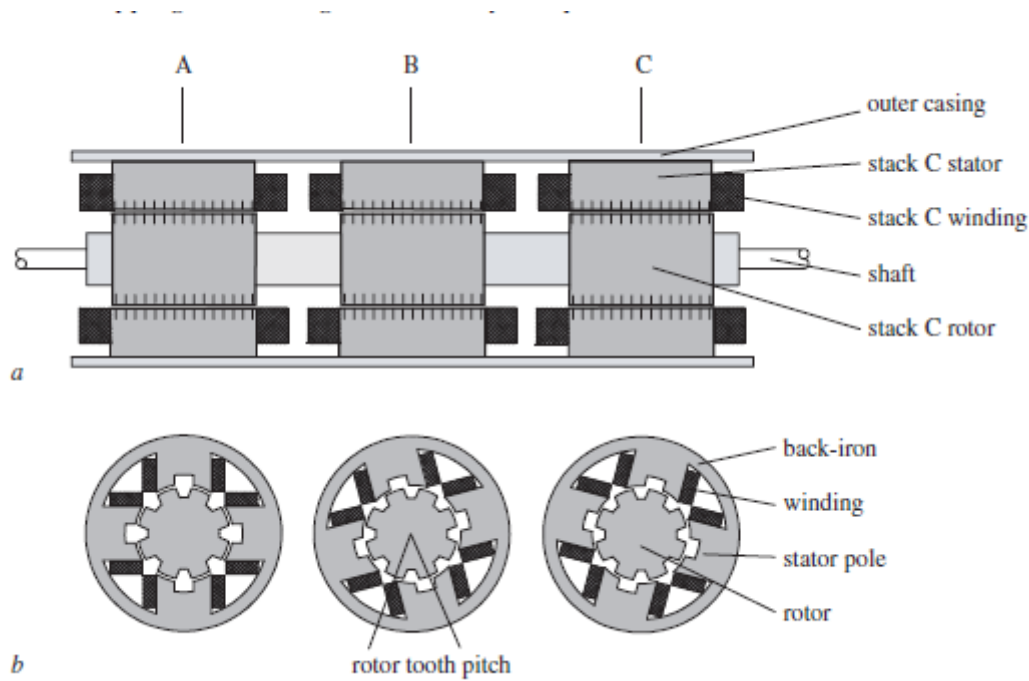


Figura 5 – Visão paralela e perpendicular das *stacks* e rotor

Segundo (ACARNLEY, 2002), existe uma pequena relação entre o comprimento do passo. Considere N o número de dentes do estator e p o número de dentes do rotor logo:

$$step\ length = 360 / (N * p) \quad (2.1)$$

2.3.6.1 DESIGN DO MOTOR

Cada polo do estator produz um campo magnético quando excitado com uma corrente DC. A performance do motor depende da força do campo magnético gerado pelas bobinas quando excitadas, logo o campo magnético está diretamente ligado ao torque do motor. A força do campo magnético está relacionada à intensidade da corrente que passa pelas bobinas, portanto em teoria aumentar a corrente para aumentar o torque seria o suficiente, entretanto existe um limitante que é o aumento da temperatura nas bobinas.

No exemplo da Figura 5 cada *stack* tem 4 polos. Uma vez que todas as quatro bobinas devem ser excitadas concorrentemente uma prática comum é interconectar as bobinas para formar apenas uma *phase*. A forma com que as bobinas são interconectadas influencia na temperatura que será dissipada pela bobina uma vez que isso está diretamente ligada à intensidade da corrente. A potência não varia conforme a interconexão.

Existem 3 formas de interconexão conforme a Figura 6. Na verdade a potência não varia conforme a interconexão, mas sim qual *driver* de controle será utilizado: baixa voltagem e alta corrente com uma conexão paralela ou alta voltagem e baixa corrente com uma conexão em série. A diferença entre as 3 interconexões pode ser vista na Figura 7 (ACARNLEY, 2002).

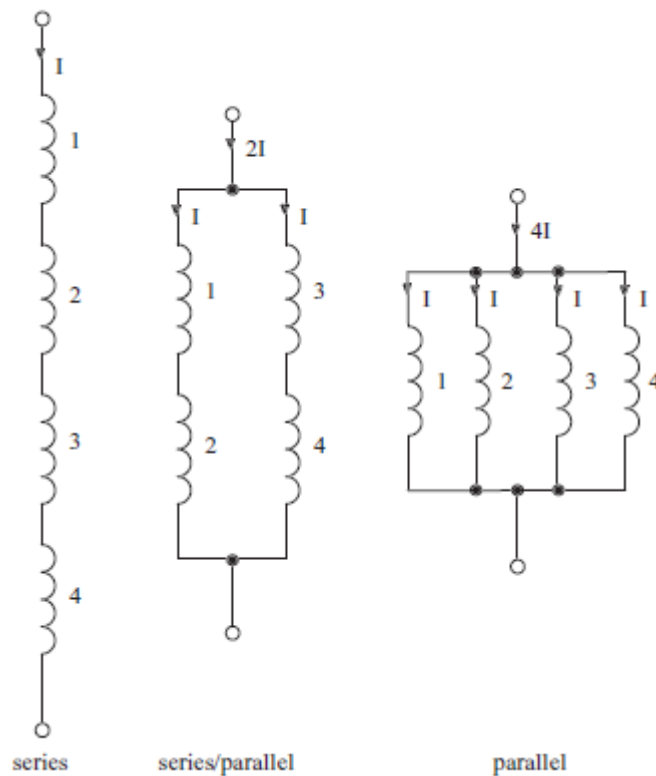


Figura 6 – Interconexão das bobinas

Connection	Rated current	Resistance	Rated voltage	Power
Series	I	$4r$	$4rI$	$4rI^2$
Series/parallel	$2I$	r	$2rI$	$4rI^2$
Parallel	$4I$	$r/4$	rI	$4rI^2$

Figura 7 – Relação das interconexões das bobinas

2.3.7 MOTOR DE PASSO RELUTÂNCIA VARIÁVEL – *SINGLE STACK*

Como o nome já diz esse motor é construído com apenas uma stack, ou melhor, uma unidade. Entretanto quanto ao funcionamento e princípios básicos é idêntico ao *multi stack*. Cada dente do estator ainda possui uma bobina separada que produz um campo magnético quando excitada por uma corrente DC.

Uma mudança é que as bobinas do lado oposto são conectadas para formar uma fase. Na Figura 8 existe 3 phases que é o número mínimo para poder rotacionar para os dois lados. A bobina no dente oposto no estator está no sentido oposto para que sejam gerados campos magnéticos em sentidos opostos. E por fim a relação de comprimento do passo se mantém conforma o motor *multi stack* (ACARNLEY, 2002).

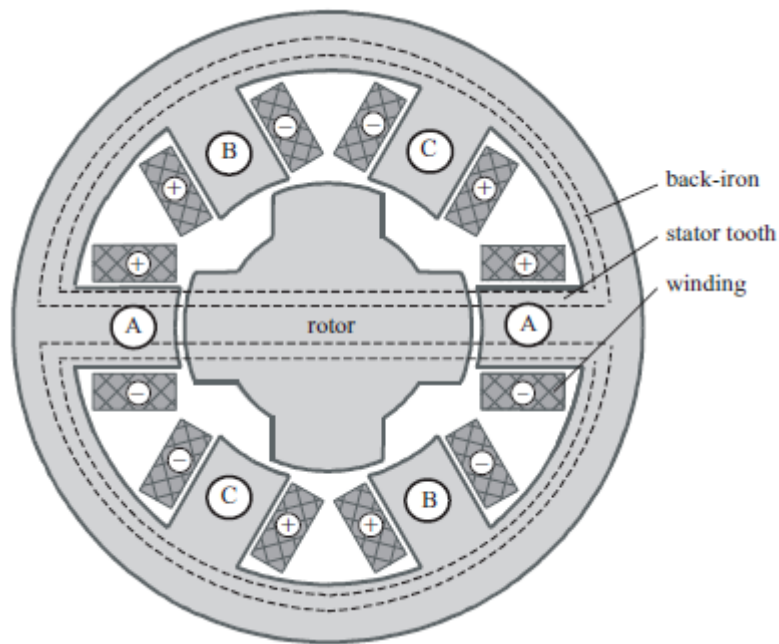


Figura 8 – Stator single stack

2.3.8 MOTOR DE PASSO HÍBRIDO

A diferença principal desse com os tipos anteriores é que o circuito magnético é excitado por uma combinação de bobinas e ímã permanente. Seu funcionamento e princípios básicos são idênticos ao *multi* e *single stack*. As bobinas ainda permanecem nos dentes do estator já o ímã compõe o eixo do rotor conforme Figura 9.

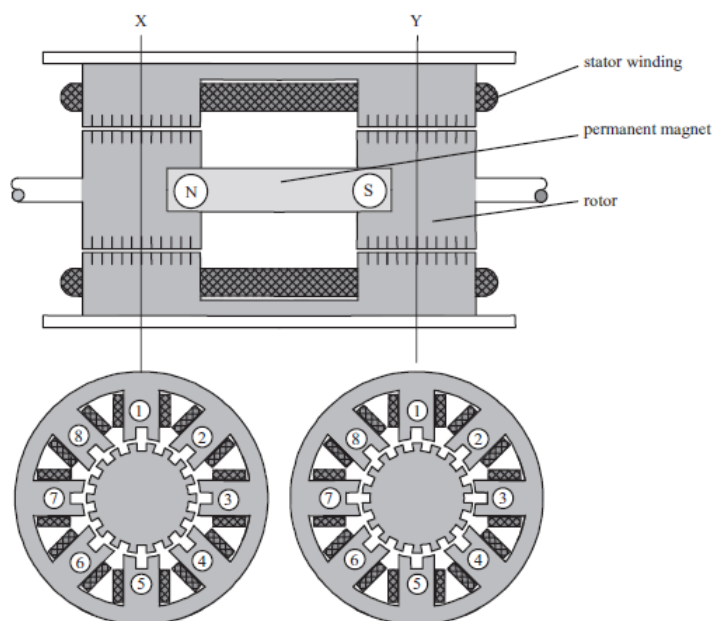


Figura 9 – Motor híbrido

Existem duas bobinas, *phases*, situadas em 4 dos 8 polos do estator, conforme Figura

9. A bobina A está nos polos 1, 3, 5, 7 e a B nos polos 2, 4, 6, 8. Polos adjacentes ainda são envolvidos pelas bobinas em sentidos opostos, portanto se a bobina A é alimentada com corrente positiva o campo magnético é direcionado para fora nas bobinas 3 e 7, mas para dentro nos polos 1 e 5 e o mesmo acontece na bobina B.

Quando se aplica corrente nas bobinas a mesma ideia de alinhamento dos dentes do rotor e *stator* acontece. Considerando a Figura 9 e o exemplo da excitação positiva na bobina A, citada anteriormente, o estator e o rotor são alinhados sob os polos 3 e 7 na seção X e polos 1 e 4 na seção Y.

Para uma rotação continua o motor necessita de uma excitação sequencial das bobinas. Se retirar a excitação de A e colocar em B o alinhamento dos dentes vai acontecer com 4 e 8 na seção X e 2 e 6 na seção Y. Isso faz com que o motor gire no sentido horário, a sequência deve ser A+, B+, A-, B-,... Para o sentido anti-horário A+, B-, A-, B+.

Segundo (ACARNLEY, 2002), a relação de comprimento do passo é similar ao de relutância variável. Existe uma relação com o número de dentes do rotor p , e com um ciclo completo de excitação. Como um esse ciclo em um motor híbrido consiste em 4 estados e produz 4 passos de movimento no rotor, logo conclui-se que:

$$step\ length = 360 / (4 * p) = 90 / p \quad (2.2)$$

2.3.9 COMPARAÇÃO ENTRE TIPOS DE MOTORES

Não é possível dizer categoricamente que um motor é melhor do que o outro em todas as situações. Os híbridos têm menor comprimento de passo, normalmente 1,8 graus, o que pode ser uma grande vantagem quando alta precisão é necessária. Eles também possuem maior torque devido ao uso do ímã permanente no rotor. E, além disso, quando nenhuma bobina esta excitada o motor híbrido ainda possui um "torque de retenção" que mantém a posição do rotor. E isso pode ser uma característica na aplicação onde a posição do rotor deve ser preservada durante uma falha de energia, mas é bom lembrar que esse torque é menor do que o torque com 1 ou mais bobinas excitadas.

Já o de relutância variável tem duas vantagens quando se trata de movimentar carga em distâncias consideráveis. A primeira é que tipicamente o comprimento de seu passo é de 15 graus, maior que o do híbrido, portanto ele precisa de menos passos para mover a mesma distância. Com a redução do número de excitações das bobinas o consumo também é reduzido, em caso de uso de baterias é uma característica muito interessante. A segunda é que por não possuir ímã permanente possui uma menor inércia para início de movimento, também diminuindo o consumo inicial (ACARNLEY, 2002).

2.3.10 LCD - *Liquid Crystal Display*

Um *display* de cristal líquido, LCD, é utilizado para exibir informações como texto, imagens e vídeos. É composto por um painel fino e, segundo (BARBACENA; AFONSO, 1996), é um componente, considerado interface de saída, muito utilizados em conjunto com microcontroladores. Exemplos de uso são: *cockpit* de aeronaves, *displays* em computadores de bordo de automóveis, dispositivos de jogos, relógios, calculadoras e outros.

Pode-se classificar os módulos de LCD em: exibição gráfica e exibição de caractere. Os módulos gráficos são encontrados com resoluções de 122x32, 128x64, 240x64 e 240x128 pixel, possuindo, geralmente, 20 pinos para conexão. Já os modelos comuns, tipo caractere, tem sua especificação em função dos números de linhas e colunas, a Figura 10 representa seus sexemplos.

Número de Colunas	Número de Linhas	Quantidade de pinos
8	2	14
12	2	14/15
16	1	14/16
16	2	14/16
16	4	14/16
20	1	14/16
20	2	14/16
20	4	14/16
24	2	14/16
24	4	14/16
40	2	16
40	4	16

Figura 10 – Tabelas de pinos LCD comum

2.4 PROTEUS DESIGN SUITE

Proteus é um suíte, conjunto de *features*, desenvolvido pela Labcenter Electronics Ltd. É um software para: simulação de microcontroladores, de circuitos eletrônicos e para desenvolvimento de placa de circuito impresso. Os componentes desse sistema são:

- *ISIS Schematic Capture*: Ferramenta utilizada para se adicionar objetos, componentes para simulação;
- *PROSPICE Mixed mode SPICE simulation*: Simulador industrial padrão SPICE3F5, combinado com simulador digital de alta velocidade. *Spice, Simulation Program with Inte-*

grated Circuit Emphasis, é um simulador para propósitos gerais para sistemas eletrônicos analógicos.

- *ARES PCB Layout*: Sistema de PCB, *Printed Circuit Board*, ou melhor, placa de circuito impresso, design de alto desempenho com posicionamento automático de componente, auto-roteamento, entre outras *features* relacionadas ao desenvolvimento de placas de circuito impresso.
- *VSM - Virtual System Modelling*: Permite simular software embarcado para microcontroladores, disponíveis em suas bibliotecas, ao lado de seu projeto de hardware ([LABCENTER, 2013](#); [WIKIPEDIA, 2013](#)).

2.4.1 ISIS SCHEMATIC CAPTURE

Como já foi dito, essa *feature* se trata da funcionalidade de simulação e montagem de circuitos eletrônicos. Permite que o circuito em questão seja depurado de forma simples e, além disso, possibilita uma integração com códigos desenvolvidos para microcontroladores suportados por ele. Seus componentes são:

- Circuitos integrados das famílias: 74ALS, 74AS, 74F, 74HC, 74HCT, 74LS, 74S e 74STD;
- Componentes analógicos;
- Medidores de corrente e tensão para serem utilizados durante a depuração;
- Geradores como fontes e *clock*, ambos ajustáveis para o valor desejado;
- Semicondutores como diodo, transistor, LEDs e outros;
- Componentes básicos como chaves, resistores, capacitores e indutores;
- Atuadores como motor DC, motor de passo e outros;
- Microcontroladores da família 80XXX, AT e PIC;
- Memórias, *displays*, CMOS e outros.

Seu uso é bem simples, pois é como se estivesse desenhando o circuito em um papel. Após ser desenhado é possível utilizar as ferramentas de auxílio para depurar e assim coletar dados como voltagem e corrente do circuito e até mesmo se o funcionamento está conforme o esperado.

2.5 MIKROC

MikroC é um *toolchain* poderoso, uma ferramenta de desenvolvimento completa para microcontroladores da família PIC. Criado de forma que disponibilize ao usuário a forma mais fácil de desenvolver suas aplicações para sistemas embarcados, sem compromisso de desempenho ou controle. MikroC permite que seja feito um rápido desenvolvimento e embarcar aplicações complexas:

- Escrever códigos em C utilizando um editor de código avançado;
- Utilizar as bibliotecas do próprio MikroC para acelerar o desenvolvimento: aquisição de dados, memória, *display*, comunicação, entre outros;
- Auxilia na evolução e desenvolvimento do código monitorando a estrutura do programa, variáveis e funções. Gera um código comentado, humanamente legível em assembler, e um padrão HEX compatível com o padrão conhecido;
- Possui um *debugger* integrado capaz de gerar relatórios detalhados, estatísticas, pilha de execução, entre outras opções que auxiliam analisar o fluxo do programa ([MIKROELETRONIKA, 2006](#)).

2.6 PIC18F452

O PIC18F452 é um microcontrolador fabricado pela empresa *Microchip Technology*. Possui tecnologia CMOS, como consequência tem um consumo baixíssimo, possui memória do tipo FLASH, um grande facilidade para desenvolvimento de protótipos, uma vez que para apagá-la não é preciso utilizar luz ultravioleta como em versões antigas, utilizavam memória EEPROM. Abaixo seguem as principais características desse microcontrolador:

- Microcontrolador de 40 Pinos;
- Memória de programa FLASH de 32Kbytes;
- Memória RAM de 1536 bytes;
- Memória EEPROM de 256 bytes;
- Processamento de até 10MIPS;
- Quatro *timers*, ou temporizadores, internos – um de 8 bits e 3 de 16 bits – TIMER0, TIMER1, TIMER2 e TIMER3;
- 2 canais capture/compare/PWM – Módulo CCP;

- Módulo *Master Synchronous Serial Port* (MSSP);
- *Unhaced Usart*;
- 8 canais A/D de 10 bits;
- Detector de baixa voltagem programável;
- Permite até 100.000 ciclos de escrita e leitura na memória FLASH;
- Permite 1.000.000 ciclos de escrita e leitura na memória EEPROM.
- Retenção de dados na FLASH por 40 anos;
- *Watchdog timer* com oscilador próprio e programável;
- Três pinos de interrupções externas: INT0, INT1 e INT2

A Figura 11 representa uma imagem do microcontrolador.



Figura 11 – PIC18F452

2.6.2 DESCRIÇÃO DOS PINOS

Dos 40 pinos desse microcontrolador 34 são pinos I/O, entrada e saída, divididos em 5 "PORT". A Figura 13 representa uma relação dos pinos do microcontrolador.

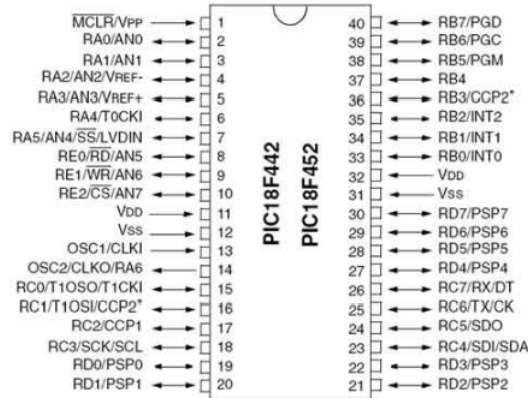


Figura 13 – Pinos PIC18F452

A divisão dos pinos I/O citada é da seguinte maneira:

- **PORTA:** São 7 pinos nomeados de RA0 a RA6. Podem ser utilizados como I/O geral ou conversor A/D, essa segundo opção tem exceção o pino RA4. Além de possuir a opção LVD, detecção de baixa tensão;
- **PORTB:** São 8 pinos nomeados de RB0 a RB7. Podem ser utilizados para I/O geral e, além disso, pode-se trabalhar com três interrupções externas, módulo CCP, pinos de gravação e debug;
- **PORTC:** São 8 pinos nomeados de RC0 a RC7. Podem ser utilizados para I/O geral, saída do oscilador do *timer*, módulo CCP, Clock e data(dados) para os modos SPI, I2C e UART;
- **PORTD:** São 8 pinos nomeados de RD0 a RD7. Podem ser utilizados para I/O geral ou como PSP para ter saída TTL, para interfaceamento com microprocessadores, por exemplo;
- **PORTE:** São 3 pinos nomeados de RE0 a RE2. Podem ser utilizados para I/O geral ou pinos de controle de acesso.

3 Bomba de infusão de insulina

3.1 Contextualização

Glicose, a principal fonte de energia do corpo humano, é absorvida a partir dos alimentos e distribuída ao corpo através da corrente sanguínea. Uma vez que está no sangue ela pode ser absorvida pelo fígado, temporariamente, utilizada pela células ou, em último caso, ser eliminada através da urina. Processos do corpo regulados pelos hormônios, que mantêm a quantidade de glicose estável na corrente sanguínea. Dos hormônios existentes os mais importante é a insulina. Sua produção é feita pelo pâncreas e sua função é controlar absorção de glicose pelas células (SBC, 2014)

Segundo (SBC, 2014) existem conjunto de doenças crônicas, chamadas diabetes, que dificultam o metabolismo da glicose e da insulina. A diagnosticção dessa doença geralmente pode ser feita através da aferição da concentração de glicose na corrente sanguínea. O resultado positivo se dá em caso de ser constatado hiperglicemia, ou seja, elevada concentração de glicose.

Dentre os tipos de diabetes o mais raro é o tipo 1. O paciente com esse tipo de doença faz com que seja necessário a aplicações diárias de insulina, consequência da secreção desse hormônio. Esse tipo da doença ocorre por causa desconhecida ou, ainda sim, pela destruição das células betas pelo pâncreas, ocasionada por um processo auto-imune (MARTINS et al., 2012). Os principais tipos tratamentos para o quadro citado são:

- Infusão Contínua Subcutânea de Insulina através de uma bomba de infusão, situação abordada neste trabalho;
- Múltiplas Doses de Insulina (MDI).

3.2 Características gerais da bomba de infusão

Segundo (MINICUCCI, 2008), para gerenciar o processo de infusão de insulina no paciente é utilizado um dispositivo eletromecânico portátil conhecido como: bomba de infusão de insulina. A ideia desse aparelho é atuar como um pâncreas artificial, ou seja, atuar de forma similar ao organismo de uma pessoa que não possui diabetes, liberando insulina durante o dia todo e no horário das refeições. A primeira forma de funcionamento, liberação de insulina entre as refeições, é chamada de basal e a segunda bolus.

A bomba geralmente é ligada a um tubo de plástico fino, cateter, que uma cânula flexível de teflon, colocada sob a pele do abdômen ou coxa. É posicionada externamente ao corpo e

possui um peso entre 80 e 100 gramas. Além das posições citadas a bomba pode ser posicionada na região lombar ou até mesmo nos membros superiores ([MINICUCCI, 2008](#)).

Segundo ([AMORIM, 2008](#)), o software embarcado, responsável pelo controle da bomba possui algumas funcionalidades como:

- avisos para monitoração da glicose;
- programação de doses de taxas basais para cada hora do dia;
- programação de doses de taxas bolus para certa quantidade de refeições;
- possibilidade de bloqueio do sistema voltados principalmente para utilização em crianças;
- possibilidade de escolha de menus operacionais;
- programação de quantidade de insulina a ser injetada;
- alarmes vibratórios e/ou sonoros para quantidade de insulina no reservatório;
- avisos para monitoração da glicose;
- ajuda de bolus que auxilia no cálculo da dose bolus necessária para correção de hiperglicemia e/ou alimentação (essa funcionalidade mais difícil de ser encontrada);
- dentre outras verificações de segurança.

Hoje em dia, a configurabilidade das funcionalidades da bomba de infusão é muito mais do que as funcionalidades básicas, o que aumentam seu custo e podem não agregar muito valor ao consumidor final. Funcionalidade essas como:

- integração com algum sistema de monitoração contínuo de glicose;
- sistema *bluetooth* onde o paciente utiliza um dispositivo externo para o controle da bomba;
- sistema de transferência de dados para um computador;
- lembretes;
- personalizações de menu;
- gráficos;
- visores coloridos;
- entre outros.

3.3 Aspectos de segurança da bomba de infusão

A bomba de infusão é considerado um sistema embarcado de tempo real crítico uma vez que deve ser extremamente seguro e fornecer respostas em prazos precisos e determinados. Considerações essas que implicam em diferenças importantes de projeto em relação a sistemas mais simples, pois complicações em sistemas desse tipo podem causar até mesmo a morte. Logo, esses dispositivos devem ser desenvolvidos com critérios de segurança robustos e requisitos funcionais e não-funcionais bem definidos (SOMMERVILLE, 2004).

É importante que o sistema seja confiável ao fornecer a quantidade correta de insulina solicitada, ou seja, ter disponibilidade sempre que for requisitada e, não menos importante, a bomba deve ser segura tratando falhas que possam suspender o fornecimento de insulina ou a infusão demasiada da mesma. Segundo (SOMMERVILLE, 2004), todas as condições anteriores retratam alguns requisitos de sistemas críticos, que no contexto deste trabalho podem ser representados por algumas funcionalidades essenciais, tais como:

- Alarmes para o nível da bateria;
- Sensor de pressão;
- Redundância de partes do hardware essenciais;
- Rotinas de verificação do funcionamento interno em geral (hardware e software);
- Implementação de rotinas que preveem iterações de forma errada por parte do paciente;
- Inacessibilidade de algumas configurações avançadas por meio do paciente;
- Alarmes para o nível de reservatório;
- Hardware de boa qualidade com as devidas certificações.

Segundo (ZHANG; JONES; JETLEY, 2010) pode-se listar situações referentes ao uso da bomba que podem causar alguma falha, podendo levar o paciente a perder a consciência no caso de uma hipoglicemia e cetoacidose para uma considerável hiperglicemia. A divisão é feita em seis grupos principais: causas operacionais, falhas de software, de hardware, causas ambientais, elétricas e químicas.

Levantamento este importantíssimo para servir como guia do projeto, de forma que se possa medir e garantir os mais altos níveis de segurança tanto no *software* quanto no *hardware*. Ainda existe classificações de riscos que podem ser energéticas, mecânicas, biológicas, químicas, ambiental e terapêutica (ZHANG; JONES; JETLEY, 2010).

Exemplos de causas operacionais que levam a situação de riscos:

- Bomba está desconectada do conjunto de infusão sem o conhecimento do paciente;

- Excessiva administração da taxa de bolus devido a várias requisições do paciente;
- Vazamento da bomba;
- Taxa atual de infusão não está de acordo com o programado;

Exemplos de falhas de *software* que levam a situação de riscos:

- *Looping* infinito;
- Acesso indevido da memória;
- Taxa incorreta de bolus recomendada pelo cálculo do sistema;
- Estouro de pilha.

Exemplos de falhas de *hardware* que levam a situação de riscos:

- Falha na memória ROM ou flash;
- Falhas do motor;
- Falha no sensor de reservatório de insulina;
- Falha no sensor de bateria;
- Falha no microcontrolador.

Exemplos de causas ambientais que levam a situação de riscos:

- Aquecimento da bomba durante o funcionamento;
- Alta diferença de pressão entre o interior da bomba e o ambiente externo;
- Uso da bomba em temperaturas fora do especificado;
- Uso da bomba em ambientes com alta umidade.

Exemplos de causas elétricas que levam a situação de riscos:

- Interferência eletromagnética vinda de outros aparelhos eletrônicos;
- Bateria desconectada;
- Vazamento de corrente pela superfície da bomba;
- Nível de bateria baixo;

- Descarga eletrostática.

Exemplos de causas químicas e/ou biológicas que levam a situação de riscos:

- Material do equipamento de baixa qualidade, problemas alérgicos;
- Infecção na região (pele) da infusão;
- Perda das propriedades bioquímicas da insulina durante a infusão, hiperglicemia;
- Precipitação química dentro do cateter, problemas alérgicos, infecções;
- Equipamento não higienizado, problemas alérgicos, infecções.

3.4 Requisitos da bomba de infusão

A bomba será implementada utilizando um microcontrolador de baixo custo da família PIC, de forma a manter as características mais importante, qualidade e segurança do dispositivo. A Figura 14 representa o modelo do contexto da bomba de infusão de insulina, com todas as entidades que deverão ser monitoradas e/ou controladas pelo software.

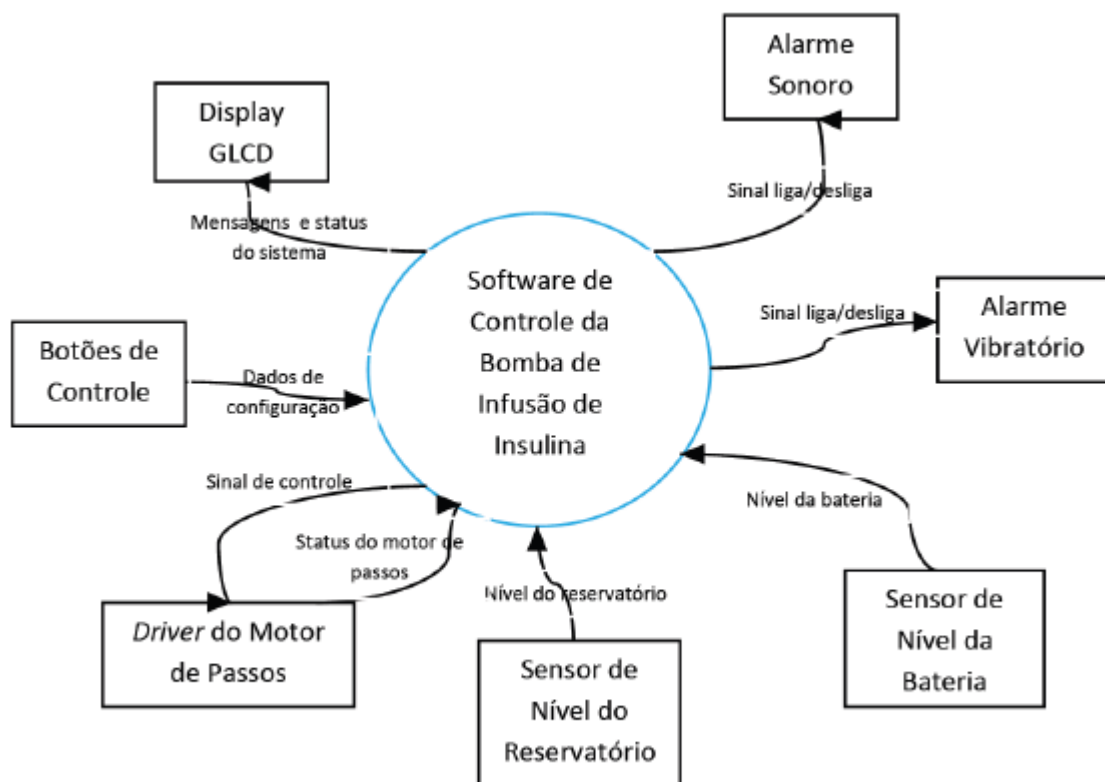


Figura 14 – Modelo de contexto da bomba de infusão de insulina

(MARTINS et al., 2012)

A divisão dos requisitos funcionais da bomba foi feita da seguinte forma:

- Módulo de interface com o usuário;
- Módulo de controle de infusão;
- Módulo de monitoramento de sensores.

Abaixo segue figuras para representar os módulos do sistema citados anteriormente. A Figura 15 representa o módulo de interface com o usuário, pode-se observar as possíveis ações que o usuário pode realizar de forma a interagir com o sistema. A Figura 16 representa o segundo módulo, controle de infusão, que tem a responsabilidade de administrar as duas formas de infusão: basal e bolus. O último, módulo de monitoramento dos sensores, está representado na Figura 17 e é responsável pela verificação do sistema como um todo.

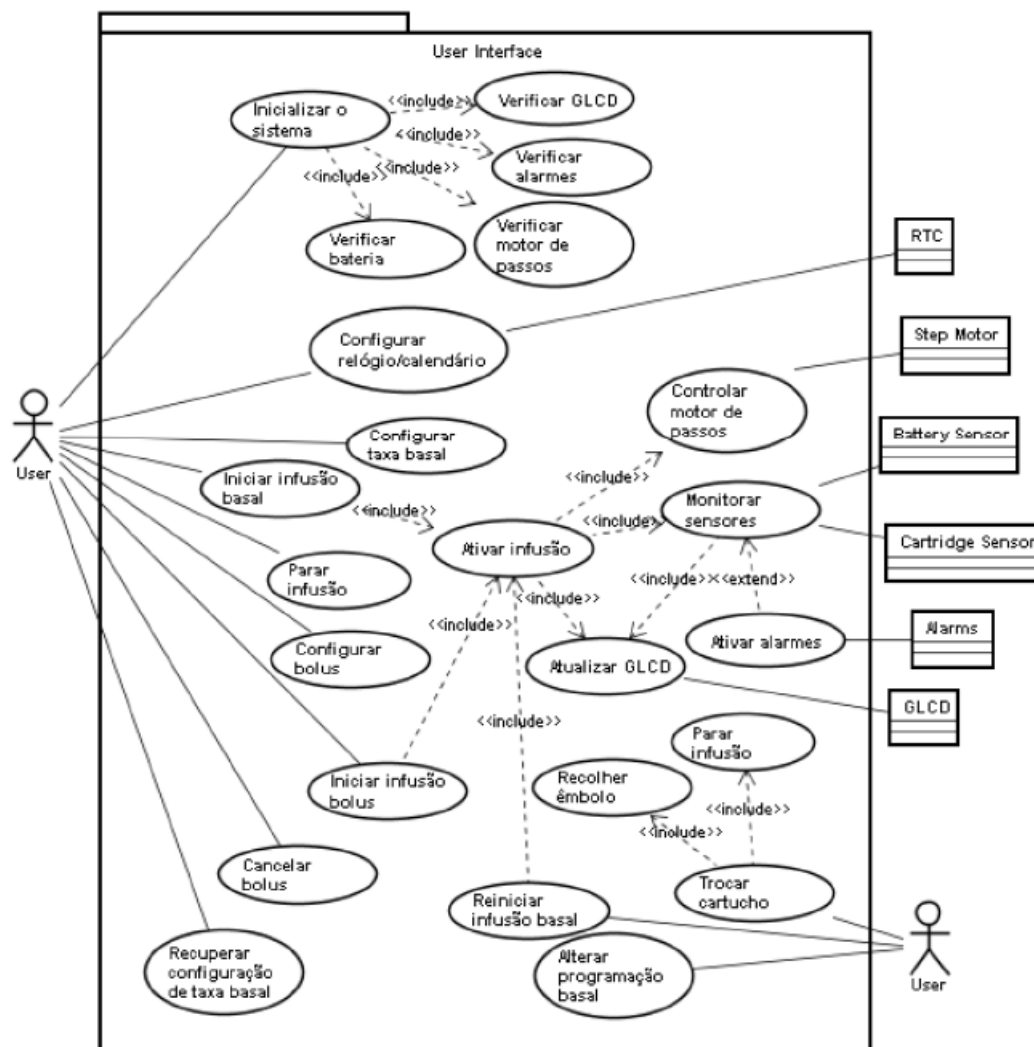


Figura 15 – Caso de uso do módulo de interface com o usuário

(MARTINS et al., 2012)

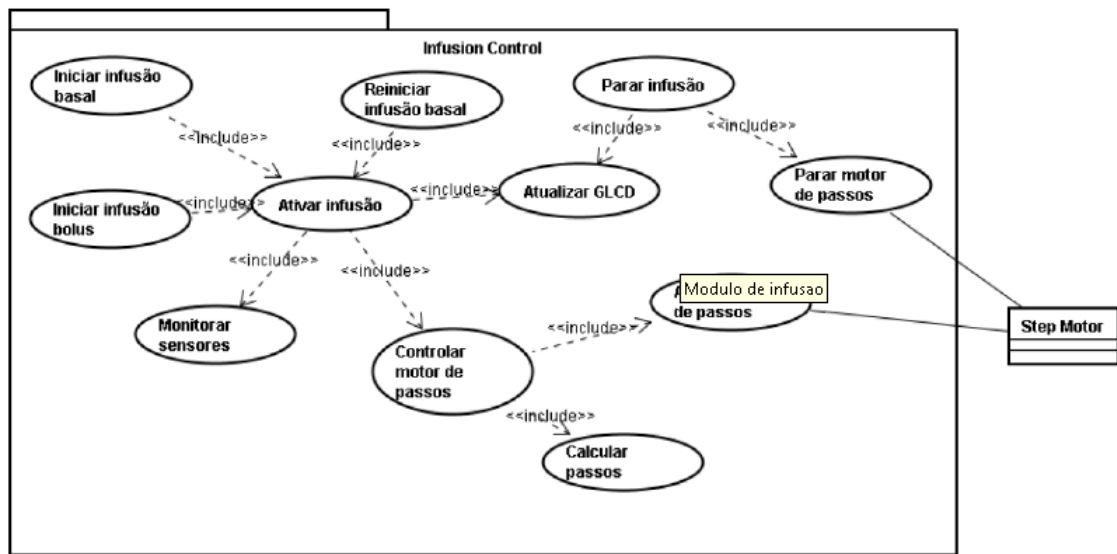


Figura 16 – Caso de uso do módulo de controle de infusão

(MARTINS et al., 2012)

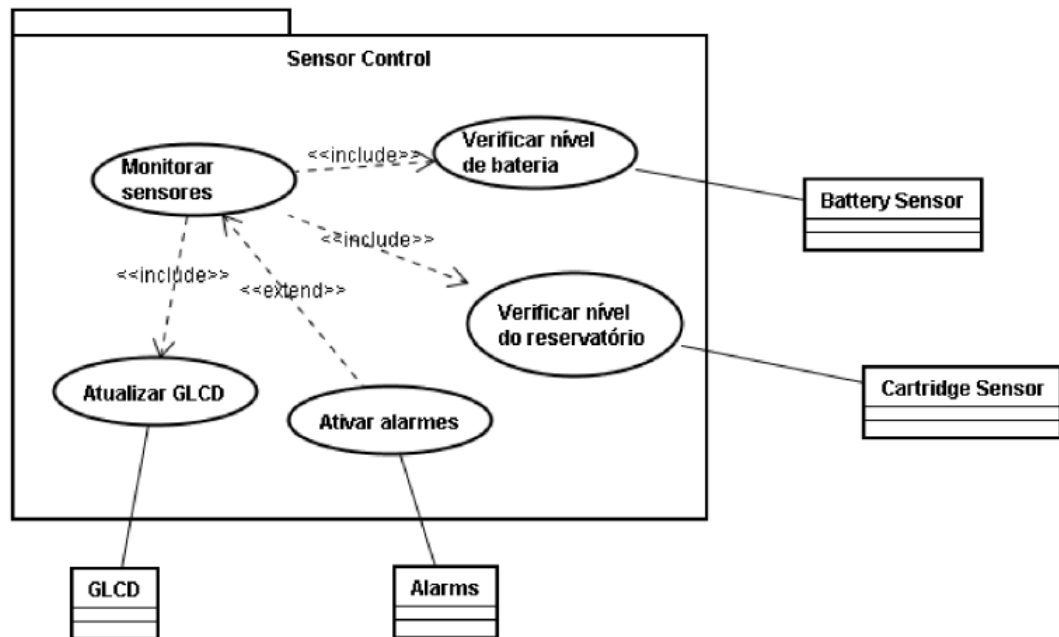


Figura 17 – Caso de uso do módulo de monitoramento dos sensores

(MARTINS et al., 2012)

4 Módulo de controle de infusão de insulina

Este trabalho tem o objetivo de estudar e desenvolver um protótipo do módulo de controle da bomba de infusão de insulina, para o desenvolvimento vamos utilizar o microcontrolador PIC18F452. Módulo este responsável por controlar o funcionamento da bomba em função dos parâmetros recebidos e pré estabelecidos. É possível configurar a quantidade a ser infundido no período de 24 horas, infusão basal, e, além disso e mais importante, o desenvolvimento foi feito em cima do simulador proteus e devido a possibilidade de mudanças possíveis o sistema foi montado de forma que deixa os módulos desacoplados. Para isso foi utilizado o conceito OOC, *Object Orientated Programming in ANSI-C*, o que possibilitou o uso de *Design Patterns*.

4.1 OOC - Object Orientated Programming in ANSI-C

Segundo (SCHREINER, 1993), OOC, *Object Orientated Programming in ANSI-C*, é uma forma de se aproveitar as diversas vantagens da Programação orientada objeto. Vantagens essas proporcionadas por linguagens como C++, java, python, smalltalk e outras. OOC, necessita da separação em dois arquivos distintos, separação semelhante a do c++: o primeiro possui a extensão por padrão ".h" e contém apenas as declarações de interfaces e classes, já o segundo, com extensão ".c", possui a devida implementação. Este é um conceito considerado mais denso do que Orientação Objeto propriamente dito, isso se deve ao fato de que o usuário implemente componentes como:

- Interface: Define-se interface como uma *struct* com ponteiros para funções, de forma que o *bind* das funções é feito na hora da criação de um objeto de uma classe que implemente essa interface;
- Classe: Da mesma forma que as interfaces, é uma *struct* em que os atributos são ponteiros para função, entretanto tem os *binds* definidos em uma função de criação(construtor) pertencente a essa classe, ou módulo;
- Construtor: Função da classe em que faz o *bind* de cada ponteiro para cada função da classe, alocações dinâmicas e outras operações que podem ser definidas de acordo com a necessidade de cada classe;
- Destrutor: Basicamente a liberação de todos os recursos alocados pelo construtor e funções que a classe utilizou;
- Public: Declaração da variável como atributo da *struct* ou no arquivo *header*;

- **Private:** Declaração da variável fora da *struct*, no arquivo que tem a implementação das funções da classe.

O conceito de polimorfismo e abstração de dados se dá pela associação de um mesmo ponteiros de função de uma classe ou interface à funções diferente, variando em função da classe mais especializada que está sendo abstraída.

Obviamente é uma simulação da consolidada programação orientada objeto e possui limitações como: Programação genérica sem validação em tempo de compilação, sintaxe mais complexa, geração de código implícito, entre outros. Entretanto ainda permite reutilização de código e, como dito anteriormente, abstração de dados.

4.2 *Design Patterns*

Na engenharia de *software*, um *Design Patterns* ou padrão de projeto é uma solução repetível geral para um problema comumente ocorre em *design* de *software*. Um padrão de projeto não é um projeto acabado que pode ser transformado diretamente em código. É uma descrição ou modelo de como resolver um problema que pode ser utilizado em diversas situações diferentes. (SHALLOWAY; TROTT, 2004).

4.2.1 Usos de *Design Patterns*

Os *Design Patterns* podem acelerar o processo de desenvolvimento, fornecendo testados e comprovados paradigmas de desenvolvimento. O *design* de *software* eficaz requer considerar as questões que não podem tornar-se visível até mais tarde na implementação. Reutilizar *Design Patterns* ajuda a prevenir situações que podem causar grandes problemas e melhora a legibilidade do código para programadores e arquitetos familiarizados com os padrões.

Muitas vezes, as pessoas só entendem como aplicar certas técnicas de *design* de *software* para determinados problemas. Estas técnicas são difíceis de aplicar a uma ampla gama de problemas. Os *Design Patterns* fornecem soluções gerais, documentadas em um formato que não requer especificidades ligadas a um problema particular (SHALLOWAY; TROTT, 2004).

Segundo (SHALLOWAY; TROTT, 2004), além disso, os padrões permitem que os desenvolvedores se comuniquem usando nomes bem conhecidos e bem compreendidos para interações de software. *Design Patterns* comuns podem ser melhorados ao longo do tempo, tornando-os mais robustos do que os projetos ad-hoc, ou seja, problemas específicos. Os *Design Patterns* podem ser divididos em:

- *Creational design patterns*: Estes *Design Patterns* tem tudo a ver com padrões de instanciação de classe. Esse padrão pode ser dividido em padrões de criação de classe e de criação de objetos. Enquanto os padrões de criação de classe usam a herança de forma

eficaz no processo de instanciação, os padrões de criação de objeto usam a delegação de forma eficaz para ter o trabalho de criação feito.

- *Structural design patterns*: Este *Design Pattern* tem tudo a ver com composição de classe e objetos. Padrão estrutural de classe usa a herança para compor interfaces. Padrão estrutural de objeto define formas de compor objetos para obter novas funcionalidades.
- *Behavioral design patterns*: São os padrões que são mais especificamente relacionadas com a comunicação entre objetos.

4.2.2 *Factory Method*

Segundo (SHALLOWAY; TROTT, 2004), *factory method* foi um dos *Design Patterns* escolhidos para esse projeto. Seus objetivos principais são:

- Definir uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar. Factory Method permite adiar a instanciação da classe para a subclasses;
- Definir um construtor *virtual*;
- Operador *new* é considerado perigoso.

Esse *Design Patterns* foi utilizado no módulo que faz uso do *display* de LCD para que a troca de seu tipo ficasse isolado do restante do sistema e que uma futura troca possa ser feita com o menor impacto possível. A Figura 18 representa sua estrutura em UML.

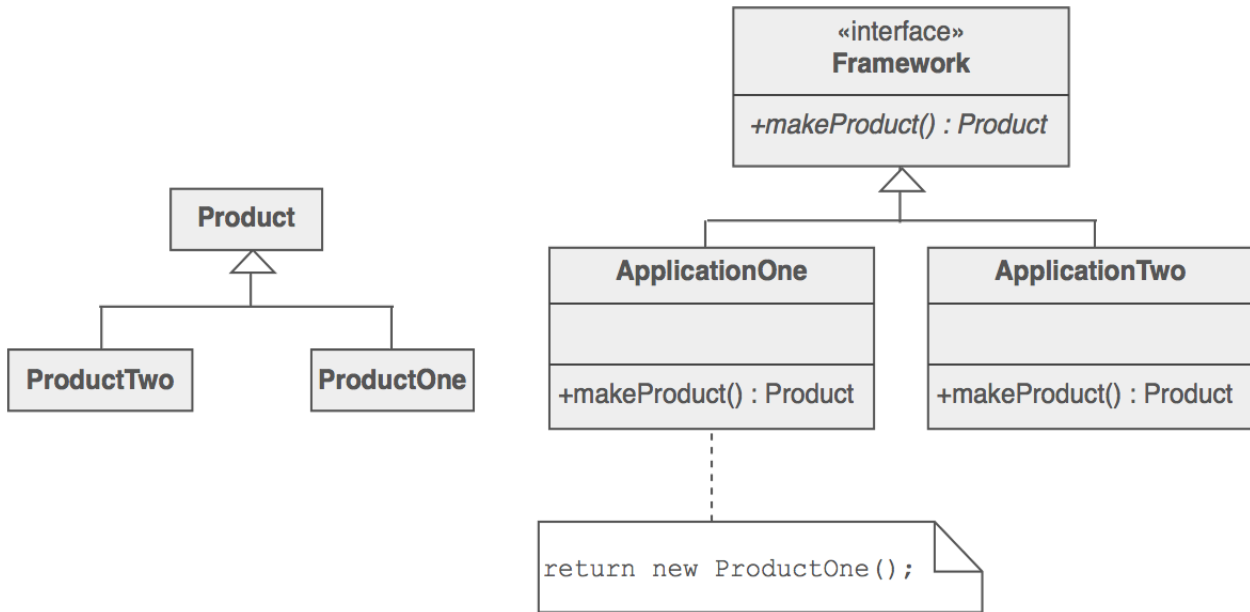


Figura 18 – Estrutura do padrão *Factory Method*
(SHALLOWAY; TROTT, 2004)

4.3 Proteus

No desenvolvimento deste TCC o simulador Proteus foi usado para montar o ambiente de teste. Para testes, o circuito teve seu desenvolvimento em função dos seguintes componentes: microcontrolador da família PIC, PIC18F452, motor de passo, *display* de LCD 2x16, botões e driver para o motor de passo. A Figura 19 representa a imagem do *Software*.

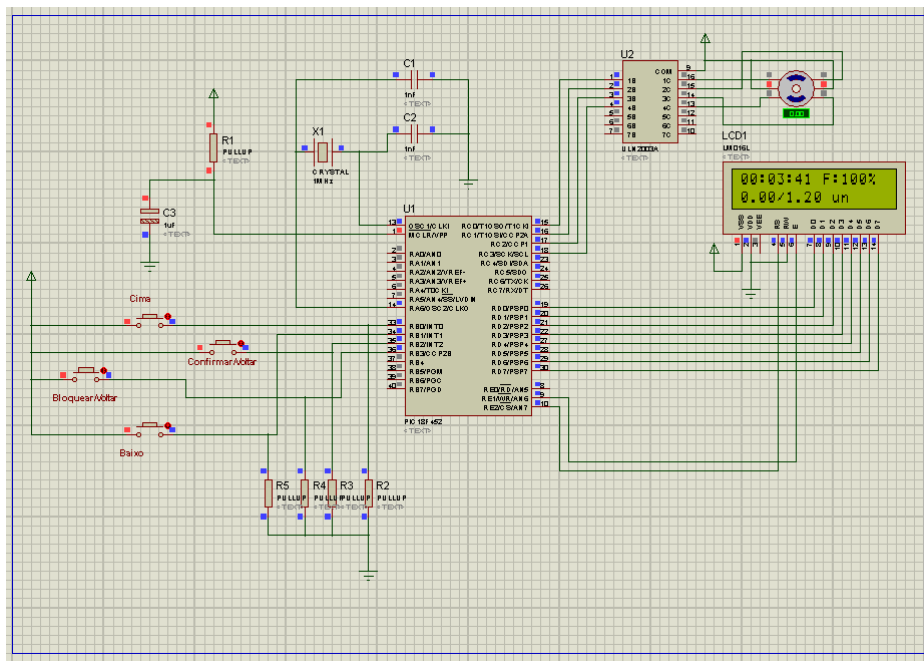


Figura 19 – *Software* Proteus

4.4 Arquitetura do Software

O *Software* foi desenvolvido de forma modular para que as funções fossem facilmente testadas, modificadas e evoluídas, a Figura 20 representa a estrutura citada. A divisão do *Software* foi feita da seguinte forma: config, insulimpump, lcd, menu, motor, timerMotor e principal.

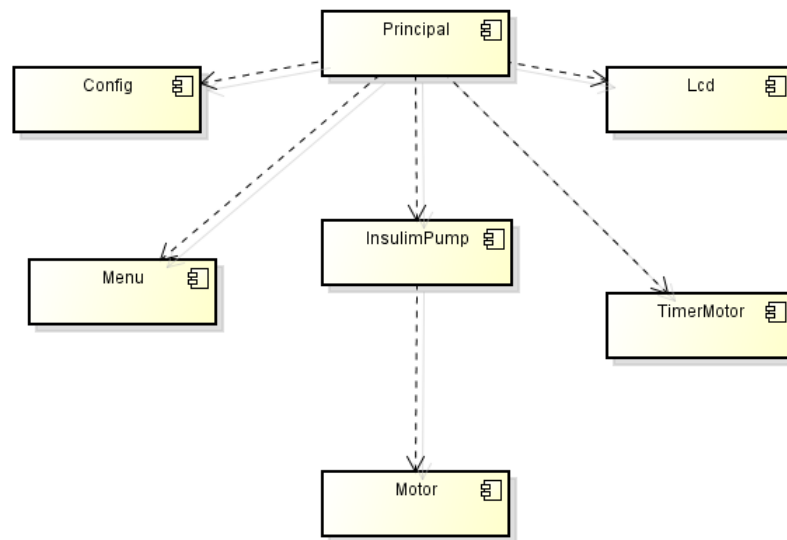


Figura 20 – Estrutura do *Software*

4.4.1 Módulo Config

Módulo feito para centralizar todas as configurações do sistema, utilizado, basicamente, por quase todos os demais módulos. Seguindo parte do conceito de OOC, pois foi possível utilizar a ideia de *private* e *public* para as variáveis. Entretanto não foi criada nenhuma *struct* de forma que representasse uma classe. A Figura 21 representa o diagrama de classe desse módulo.

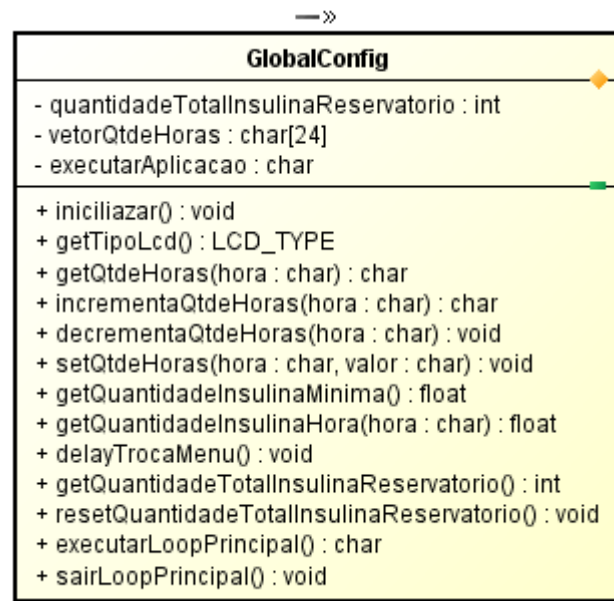


Figura 21 – Diagrama de classe GlobalConfig

Quando se diz *private* e *public* é o fato de adicionar as variáveis no arquivo `GlobalConfig.h`, *private*, ou `GlobalConfig.c`, *public*. A principal parametrização desse módulo é a quantidade que o repositório de insulina suporta. Para modificar seu valor basta alterar o *define* `QUANTIDADE_TOTAL_RESERVATORIO_INSULINA`. Esse valor representa a quantidade de insulina existente no reservatório na escala da infusão mínima da bomba.

4.4.2 Módulo InsulinPump

O módulo `InsulinPump` é responsável pela abstração das funções da bomba para o resto do sistema. Funções simples para o resto do sistema como: inicializar variáveis de controle, iniciar operação, parar operação - principalmente para configurações da bomba -, injetar e retornar a quantidade inserida naquela hora. Seguindo o conceito OOC foi criada uma "interface", `IInsulinPump`, e a "classe" concreta `InsulinPump`. A Figura 22 representa a relação e diagrama de classe dos elementos citados anteriormente.

4.4.3 Módulo Lcd

Esse módulo é responsável pelo isolamento do *display* de LCD do resto do sistema. Composto por um *factory*, que retorna um objeto de controle de acordo com o parâmetro passado, uma "interface" para possibilitar essa abstração e as classes concretas, no caso existe apenas a classe concreta para *display* 2x16 (2 linhas por 16 colunas). A Figura 23 representa a relação e diagrama de classe dos elementos citados anteriormente.

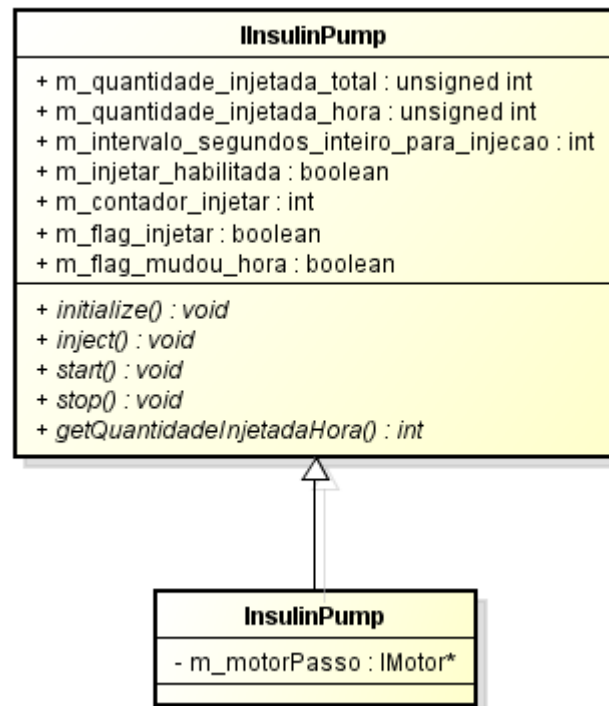


Figura 22 – Diagrama de classe InsulinPump

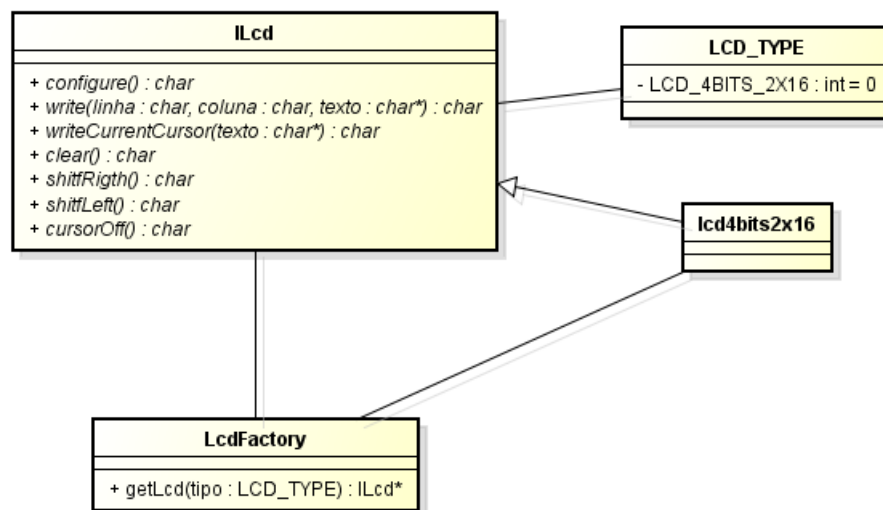


Figura 23 – Diagrama de classe Lcd

4.4.4 Módulo Menu

O módulo de menu foi criado para abstrair todos as possíveis situações e estados da bomba. Foi criado de uma forma muito simples e é equivalente a uma máquina de estado. Utiliza uma "interface" que é implementado por todos os tipos de menu existentes como: menu de configuração da bomba, menu da bomba em execução, menu de confirmação do estado do reservatório e outros. A Figura 24 representa as relações entre os componentes, ou "classes", do módulo e demonstra como é simples a criação de novos menus.

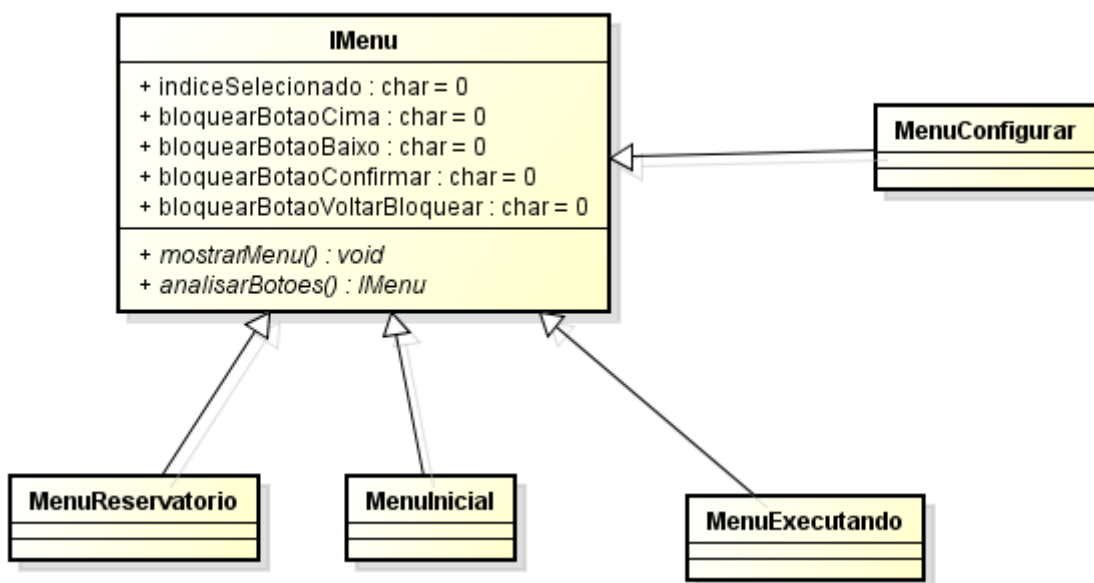


Figura 24 – Diagrama de classe Menu

4.4.5 Módulo Motor

O módulo motor, tem como responsabilidade abstrair todas as operações necessárias para o controle do motor de forma que o resto do sistema não saiba qual motor está utilizando. Isso é possível devido a "interface" criada para abstração e o uso de um *Factory* que retorna o motor que deve ser utilizado de acordo com um parâmetro global, localizado no módulo config. A figura 25 representa as relações existentes nesse módulo.

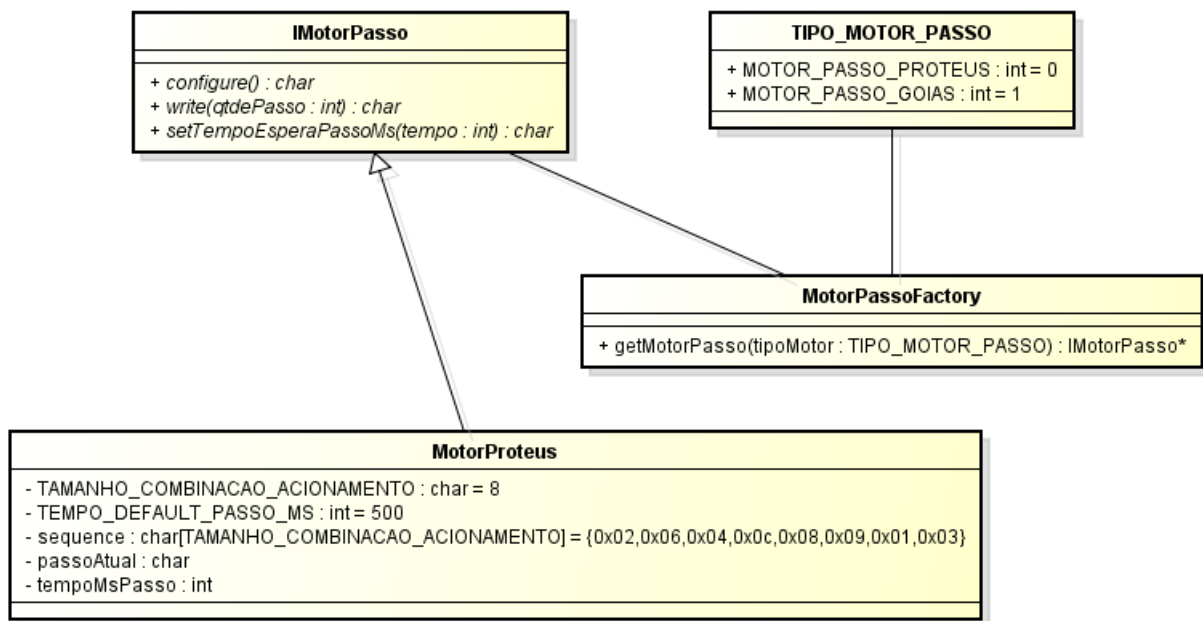


Figura 25 – Diagrama de classe Motor

4.4.6 Módulo TimerMotor

O módulo TimerMotor armazena todos os dados do contador para o uso dos motores. Abstrai a configuração do timer em função do *hardware* para o resto do sistema. Dessa forma o sistema só precisa se preocupar em: fazer configuração inicial(inicializar variáveis), iniciar timer, parar timer. A Figura 26 representa a relação citada acima.

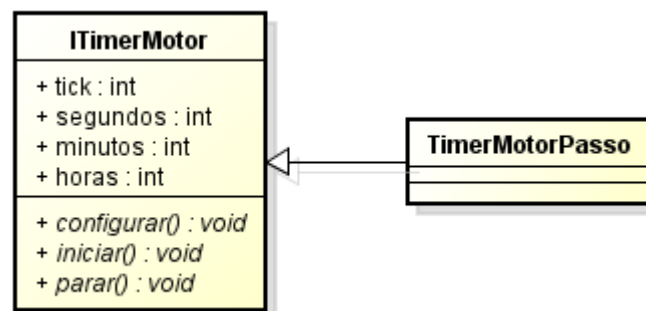


Figura 26 – Diagrama de classe TimerMotor

4.4.7 Módulo Principal

Devido a forma modular com que foi implementado o sistema esse tornou-se o módulo mais simples. Ficou responsável apenas pela interrupção, específico do hardware, chamar os métodos de inicialização dos módulos do lcd, TimerMotor e InsulinPump, e um loop principal, extremamente simples, que exibe o Menu, correspondente ao estado atual, e recebe o retorno do próprio menu em questão para navegar entre os menus existente.

5 Análise e discussão dos resultados

5.1 Simulador

O uso do simulador foi importantíssimo para o desenvolvimento desse trabalho. Ele possibilitou iniciar o desenvolvimento sem uma placa física. Utilizando-o montou-se um circuito exemplo utilizando o microcontrolador escolhido e, de forma iterativa, era possível evoluir tanto o circuito quanto o software, por exemplo: montou-se o circuito com o LCD e implementou-se o módulo de controle do mesmo, adicionou-se os componentes de menu e a análise de botões, e outros passos. Vendo esses passos é possível perceber que o desenvolvimento foi feito por funcionalidade isoladas de forma que fosse possível testar cada "parte" do software e assim minimizar os problemas do *software* como um todo e assim sempre ter um versão estável, mesmo que mínima.

Além de todas essas facilidades, a gama de testes e depuração é maior em um simulador do que no *hardware*, como: o simulador indica más práticas existentes, permite simular o uso da bomba por dias em questão de minutos, basta configurar o tempo de execução, e outros. Graças aos exemplos citados anteriormente foi possível encontrar diversos problemas que só seriam descobertos quando estivesse testando diretamente no *hardware* e mesmo assim surgiria a dúvida: É o *hardware* ou o *software*.

O circuito elétrico utilizado para os testes foi desenvolvido com base na placa Microgenios, representada pela Figura 27. Todos os componentes utilizados assim como as ligações entre eles são fiéis à placa de referência, salvo às ligações do motor que não faz parte do conjunto de referência. Portanto uma migração para a placa de Microgenios teria o funcionamento dos botões existentes e *display* de LCD de acordo com o esperado.

E, além disso, como os testes e validações do *hardware* feito através simulador são extremamente válidas e consistentes e o desenvolvimento foi baseado em uma placa consolidada, uma integração com uma placa diferente da utilizada como referência será muito mais simples. Isso deve-se ao fato através do simulador diminui-se os riscos e problemas no período de integração, onde são encontrados as maiores e inesperadas dificuldades do desenvolvimento.

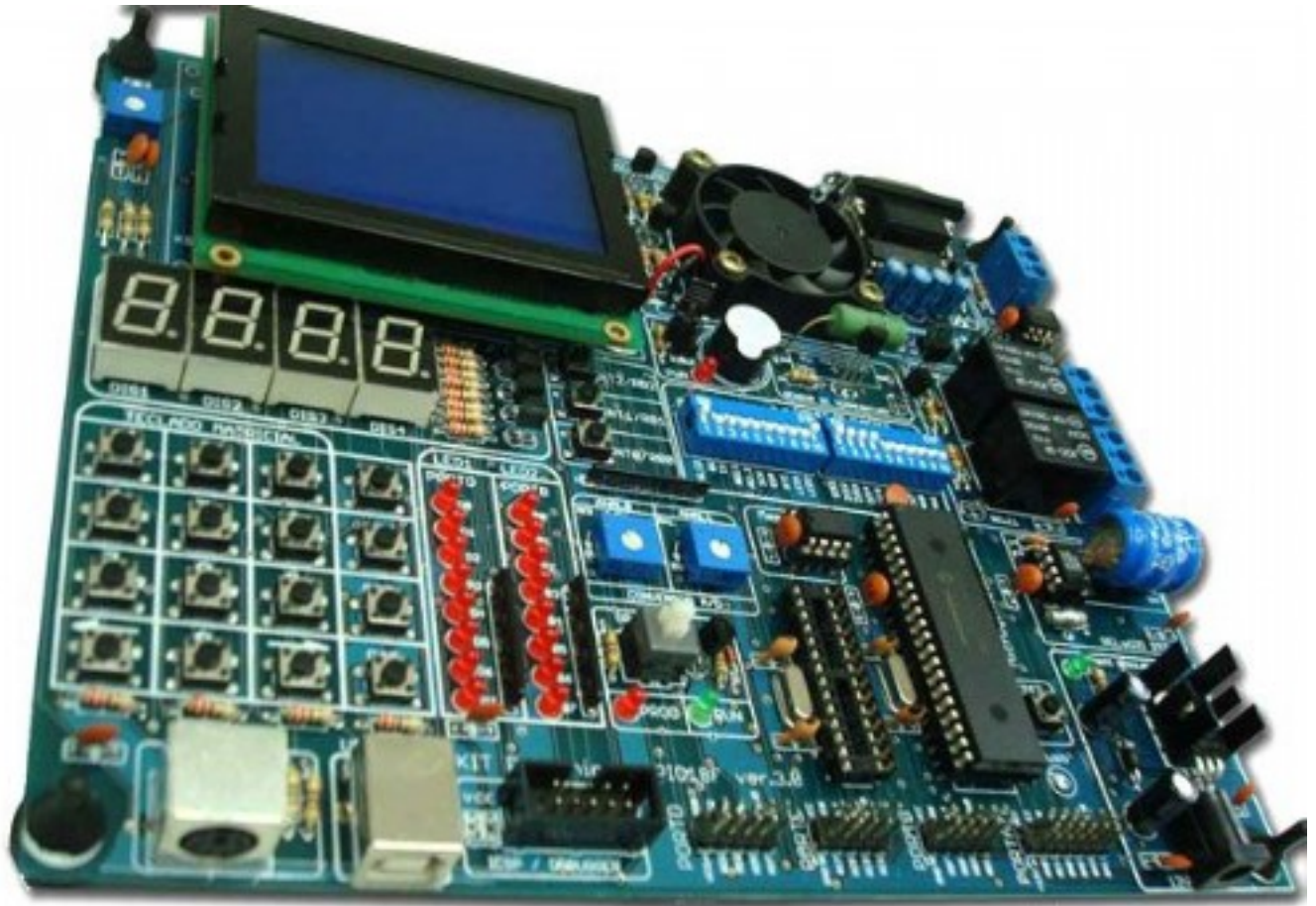


Figura 27 – Placa Microgenios PIC18F4452

5.2 Arquitetura modularizada

A organização do código é o grande destaque e vantagem desse trabalho. O desacoplamento foi o foco principal durante todo o desenvolvimento isso para facilitar mudanças no próprio *software* ou do *hardware*.

Esse desacoplamento alcançado deveu-se graças ao uso do conceito OOC, *Object Orientated Programming in ANSI-C*. Através de seu uso pode-se aproveitar das diversas vantagens que a Programação Orientada Objetos proporciona, embora seja um conceito considerado mais volumoso já que seu uso requer um conhecimento de como as linguagens OO abstraem os operadores desse paradigma, pois são implementados manualmente, como:

- *Bind*: É a associação de uma chamada a um método ou função com sua implementação, de forma mais específica é definir para quem o ponteiro da função está endereçado;
- *Interface*: Definição dada por uma *struct* com ponteiros para funções. O *bind* é feito no momento da criação do objeto da classe que implemente essa interface;

- Classe: Análogo às interfaces, é uma *struct* em que os atributos são ponteiros para função, entretanto tem os *binds* definidos em uma função de criação(construtor) pertencente a essa classe, ou módulo;
- Construtor: Função da classe responsável por realizar o *bind* de cada ponteiro para cada função da classe, alocações dinâmicas e outras operações que podem ser definidas de acordo com a necessidade de cada classe;
- Destrutor: Basicamente a liberação de todos os recursos alocados pelo construtor e funções que a classe utilizou;
- Public: Para declarar um atributo com esse operador de acesso deve-se declarar a variável como atributo da *struct* ou no arquivo header, seja da classe ou da interface;
- Private: Para esse operador a declaração da variável se dá fora da *struct*, não pode ser no *header*, pois é o arquivo conhecido pelo resto do sistema.
- Polimorfismo e abstração de dados: Resultado da associação de um mesmo ponteiros de função de uma classe ou interface à funções diferente, variando em função da classe mais especializada que está sendo abstraída.

Mas obviamente OOC é limitado e algumas complicações em sua adaptação para o paradigma OO. Validações em tempo de compilação são menos específicas, programação genérica, sintaxe mais complexa, geração de código implícito, entre outros.

Essa abordagem foi importantíssima, pois é possível fazer testes, mocks no sistema, pelo fato de simular programação orientação à objetos, possibilitando realizar testes unitários e utilizando um PC, ambiente utilizado: Ubuntu 13.10, compilador gcc, Framework C++ Qt e IDE QtCreator. Uma vez que as classes e interfaces possuem ponteiros de funções, para que o *bind* seja feito nas funções de criação, a realização do mock é muito simples, pois basta fazer com que o ponteiro da função a ser mockada seja endereçada para outra função de acordo com o teste a ser realizado. Ao realizar os testes e mocks o acesso aos periféricos, todos os acionamentos e comunicações, ou seja, tudo vinculado ao *hardware* podem redirecionados para um arquivo log de acompanhamento. A definição da arquitetura foi totalmente desenvolvida em função da possibilidade de realização de testes unitários, uma vez que para se realizar os testes é preciso um alto nível de desacoplamento. A seguir está descrito as vantagens principais de cada módulo devido a forma de implementação.

O módulo Config é o mais simples de todos, sua principal vantagem é o fato de centralizar todas as informações de configuração, fazendo com que o sistema fique mais claro. Além disso, criar casos de testes torna-se simples, pois mudando qualquer uma de suas informações já se reflete no sistema como um todo. É importante lembrar também que ele não carrega nenhuma

dependência do compilador ou do hardware, foi implementado em ANSI-C, o que permite que seja utilizado por qualquer compilador e *hardware*.

Os demais módulos foram criados para seguir a ideia de isolamento do módulo config. Para isso o que foi feito é deixar as interfaces e funções comuns independente de *hardware* e compilador utilizando apenas ANSI-C. Dessa forma, caso precise de alguma mudança mais drástica relacionada aos dois itens citados o impacto seja mínimo, precisando fazer apenas uma equivalência das funcionalidades específicas.

Tendo dito as vantagens com relação às mudanças *hardware* e compilador é importante lembrar que a expansibilidade e manutenibilidade do software ficou incrivelmente simples. A ideia foi deixar o "motor" ou "coração" da bomba ter conhecimento apenas das "interface" dos sistemas. Dessa forma os detalhes de funcionamento, requisitos de segurança, configurações específicas de hardware para controle dos periféricos e outros, podem ser modificadas sem impactar o funcionamento básico da bomba de infusão. O mais importante é que essas alterações são parametrizáveis, localizando-se no módulo config, e o software sabe quais objetos utilizar e de onde recuperá-los, em algum *Factory* na maioria dos casos. Devido a isso tudo é possível manter mais de um produto em um único código e mudanças importantes no *core* do sistema não precisa ser replicadas e correr risco de conflito entre projetos. Graças ao uso do OOC para criar algo novo basta implementar as funções da interface do módulo em questão e adicionar ao *factory*, caso exista. Segue uma breve explicação do que pode ser feito em cada módulo.

O módulo InsulinPump, responsável por abstrair particularidades de funcionalidades e segurança da bomba para o resto do ambiente, permite criar diversos tipos de bomba. Essa criação leva em conta mudanças nos dois itens citados, por exemplo, bomba europeia, americana, brasileira, que podem ter requisitos de segurança e funcionalidades distintas.

O módulo LCD, responsável por abstrair a forma de uso e qual *display* está sendo utilizado. Ele permite a troca do tipo de LCD utilizado seja simples, por exemplo, trocar o utiliza 2x16, por um 2x12.

O módulo Menu, lembra uma máquina de estado, pois retorna um próximo Menu(estado) ou ele mesmo caso a mudança não seja necessária. Para adicionar um novo menu basta colocá-lo como retorno em alguma situação dentro da função de análise de botões que todos os Menus possuem.

O módulo Motor segue a mesma linha do LCD, permite a troca do periférico por outro motor de passo ou até mesmo um outro tipo de motor. Esse módulo possui um *factory* para que seja possível recuperar o objeto de abstração do motor desejado.

Por fim, o módulo TimerMotor foi criado para desvincular o *timer* que é extremamente dependente do *hardware* e compilador. Foi o módulo mais complicado de se isolar e é o único que não está totalmente isolado devido a algumas limitações da execução de interrupções.

6 Conclusão

Este trabalho teve como base o desenvolvimento e simulação de um *software* de controle para um protótipo de bomba de infusão de insulina baseado em um microcontrolador da família PIC. Além disso, desenvolveu-se juntamente com o software de controle um módulo de comunicação com periféricos para interação com o usuário. O *software* desenvolvido é responsável por executar os perfis de infusão configurados pelo usuário, ativar o motor de passos, dar *feedback* do status da bomba e de sua execução ao paciente. Os objetivos desse trabalho foram alcançados ao desenvolver o *software* utilizando o microcontrolador de baixo custo da família PIC: PIC18F452.

A compreensão das tecnologias utilizadas, simulador e, é claro, da natureza do problema proposto foram imprescindíveis. O presente trabalho pode ser considerado uma prova de conceito do módulo desenvolvido, e pode ser continuado e aprimorado sem grandes problemas devido a sua alta modularidade e simplicidade do código. E com isso realizar testes mais precisos e factíveis com relação a realidade do problema proposto.

Considerando como projeto futuro seria os testes utilizando um *hardware*, placa da Microgenios citada anteriormente, composto pelos componentes abordados e comprovar as vantagens do uso do simulador, demonstrar que testes de *stress*, ou seja, por longos períodos pode ser simulado no PC, graças ao OOC. E, por fim, talvez menos importante, conseguir isolar completamente a dependência do compilador e *hardware* devido à forma de uso da interrupção para contagem de tempo para infusão.

Por fim, o código é aberto, licença MIT e pode ser encontrado em: https://gitlab.com/dinesh/insulin_pump.

Referências

ACARNLEY, P. P. *Stepping motors: a guide to theory and practice*. [S.l.]: Iet, 2002. Citado 4 vezes nas páginas 36, 38, 39 e 41.

AMORIM, A. C. P. d. Novas abordagens em insulinoterapia. Universidade da Beira Interior, 2008. Citado na página 50.

BARBACENA, I.; AFONSO, C. F. *DISPLAY LCD*. UNI-CAMP, 1996. Acessado em 20/06/2014. Disponível em: <ftp://ftp.dca.fee.unicamp.br/pub/docs/ea079/complementos/Lcd.pdf>. Citado na página 42.

CHIRONIS, N. P.; SCLATER, N. *Mechanisms & mechanical devices sourcebook*. [S.l.]: McGraw-Hill New York, 1991. Citado na página 35.

CUNHA, A. *O que são sistemas embarcados?* 2013. Acessado em 20/05/2014. Disponível em: <http://www.techtraining.eng.br/files/uploads/2013/04/19/artigo-sist-emb.pdf>. Citado na página 31.

DIABETES. *Bomba de infusão de insulina*. 2013. Acessado em 08/03/2013. Disponível em: <http://www.diabetes.org.br/sala-de-noticias/2316-bombas-de-infusao-de-insulina>. Citado na página 27.

FARINES, J.-M.; FRAGA, J. d. S.; OLIVEIRA, R. d. Sistemas de tempo real. *Escola de Computação*, v. 2000, p. 201, 2000. Citado na página 32.

FELDMANN, R. L. et al. A survey of software engineering techniques in medical device development. In: IEEE. *High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, 2007. HCMDSS-MDPnP. Joint Workshop on*. [S.l.], 2007. p. 46–54. Citado na página 33.

GANSSE, J. *The art of designing embedded systems*. [S.l.]: Newnes, 1999. Citado na página 34.

GERMANO, A. *Você sabe o que são sistemas embarcado?* 2011. Acessado em 20/05/2014. Disponível em: <http://www.gruponetcampos.com.br/2011/05/voce-sabe-o-que-sao-sistemas-embarcados/>. Citado na página 31.

JORDAN, P. Standard iec 62304-medical device software-software lifecycle processes. In: IET. *Software for Medical Devices, 2006. The Institution of Engineering and Technology Seminar on*. [S.l.], 2006. p. 41–47. Citado na página 33.

KOPETZ, H. *Real-time systems: design principles for distributed embedded applications*. [S.l.]: Springer, 2011. Citado na página 33.

LABCENTER. *Labcenter Eletronics*. 2013. Acessado em 05/03/2013. Disponível em: <http://www.labcenter.com/index.cfm>. Citado na página 43.

LEE, E. A.; SESHIA, S. A. *Introduction to embedded systems: A cyber-physical systems approach*. [S.l.]: Lee & Seshia, 2011. Citado na página 34.

- MALL, R. *Real-Time Systems: Theory and Practice*. [S.l.]: Pearson Education India, 2009. Citado na página 32.
- MARTINS, L. E. G. et al. Desenvolvimento de um protótipo de bomba de infusão de insulina de baixo custo: em busca de uma alternativa para a população de baixa renda portadora de diabetes. Projeto submetido ao MCTI, 2012. Citado 4 vezes nas páginas 49, 53, 54 e 55.
- MIKROELETRONIKA. *MikroC Making it Simple User's Manual*. [S.l.], 2006. Disponível em: <http://www.mikroe.com/pdf/mikroc/mikroc_manual.pdf>. Acesso em: 16.4.2013. Citado na página 44.
- MINICUCCI, W. J. Uso de bomba de infusão subcutânea de insulina e suas indicações:[revisão]. *Arq. bras. endocrinol. metab*, v. 52, n. 2, p. 340–348, 2008. Citado 2 vezes nas páginas 49 e 50.
- NOMADSUSP. *Sensores*. 2012. Acessado em 02/04/2014. Disponível em: <<http://www.nomads.usp.br/pesquisas/design/dos/Capacitacao/arquivos/sensores.pdf>>. Citado na página 35.
- PORTALDIABETES. *Iniciando insulino terapia no diabetes mellitus tipo 2*. 2008. Acessado em 04/03/2013. Disponível em: <<http://www.portaldiabetes.com.br/conteudocompleto.asp?idconteudo=3267>>. Citado na página 25.
- PORTALDIABETES. *Bombas de Infusão de Insulina*. 2009. Acessado em 10/03/2013. Disponível em: <<http://www.portaldiabetes.com.br/conteudocompleto.asp?idconteudo=3267>>. Citado na página 27.
- RADIOAMADORES. *Microcontroladores*. 2009. Acessado em 02/12/2012. Disponível em: <http://www.radioamadores.net/files/microcontroladores_pic.pdf>. Citado na página 34.
- ROSÁRIO, J. M. *Princípios de mecatrônica*. [S.l.]: Pearson Prentice Hall, 2006. Citado na página 35.
- SANTOS, V. P. de A. Motor de passo. 2008. Citado na página 36.
- SBC. *Tudo sobre diabetes*. 2014. Acessado em 20/05/2014. Disponível em: <<http://www.diabetes.org.br>>. Citado 2 vezes nas páginas 27 e 49.
- SCHREINER, A.-T. *Object oriented programming with ansi-c*. Hanser, 1993. Citado na página 57.
- SHALLOWAY, A.; TROTT, J. R. *Design patterns explained: a new perspective on object-oriented design*. [S.l.]: Pearson Education, 2004. Citado 3 vezes nas páginas 58, 59 e 60.
- SOMMERVILLE, I. *Software Engineering. International computer science series*. [S.l.]: Addison Wesley, 2004. Citado 2 vezes nas páginas 33 e 51.
- WIKIBOOK. *Embedded Systems Introduction*. 2012. Acessado em 02/12/2012. Disponível em: <http://en.wikibooks.org/wiki/Embedded_Systems/Embedded_Systems_Introductio>. Citado na página 31.
- WIKIPEDIA. *Motor de passo*. 2012. Acessado em 04/12/2012. Disponível em: <http://pt.wikipedia.org/wiki/Motor_de_passo>. Citado na página 36.

WIKIPEDIA. *Pic*. 2012. Acessado em 01/12/2012. Disponível em: <http://pt.wikipedia.org/wiki/Microcontrolador_PIC>. Citado na página 34.

WIKIPEDIA. *SPICE*. 2013. Acessado em 06/03/2013. Disponível em: <<http://en.wikipedia.org/wiki/SPICE>>. Citado na página 43.

ZHANG, Y.; JONES, P. L.; JETLEY, R. A hazard analysis for a generic insulin infusion pump. *Journal of diabetes science and technology*, SAGE Publications, v. 4, n. 2, p. 263–283, 2010. Citado na página 51.

Apêndices

APÊNDICE A – Config

A.1 Config.c

```
#include "GlobalConfig.h"

#define QUANTIDADE_TOTAL_RESERVATORIO_INSULINA 100
int quantidadeTotalInsulinaReservatorio =
    QUANTIDADE_TOTAL_RESERVATORIO_INSULINA;
char vetorQtdeHoras[24];
char executarAplicacao = 1;

void iniciliazar(){
    vetorQtdeHoras[0] = 0;
    vetorQtdeHoras[1] = 0;
    vetorQtdeHoras[2] = 0;
    vetorQtdeHoras[3] = 0;
    vetorQtdeHoras[4] = 0;
    vetorQtdeHoras[5] = 0;
    vetorQtdeHoras[6] = 0;
    vetorQtdeHoras[7] = 0;
    vetorQtdeHoras[8] = 0;
    vetorQtdeHoras[9] = 0;
    vetorQtdeHoras[10] = 0;
    vetorQtdeHoras[11] = 0;
    vetorQtdeHoras[12] = 0;
    vetorQtdeHoras[13] = 0;
    vetorQtdeHoras[14] = 0;
    vetorQtdeHoras[15] = 0;
    vetorQtdeHoras[16] = 0;
    vetorQtdeHoras[17] = 0;
    vetorQtdeHoras[18] = 0;
    vetorQtdeHoras[19] = 0;
    vetorQtdeHoras[20] = 0;
    vetorQtdeHoras[21] = 0;
```

```
vetorQtdeHoras[22] = 0;
vetorQtdeHoras[23] = 0;
}

#ifdef __cplusplus
extern "C"
#endif
LCD_TYPE getTipoLcd() {
    return LCD_4BITS_2X16;
}

char getQtdeHoras(char hora) {
    return vetorQtdeHoras[hora];
}

char incrementaQtdeHoras(char hora) {
    ++vetorQtdeHoras[hora];
    return vetorQtdeHoras[hora];
}

char decrementaQtdeHoras(char hora) {
    if (vetorQtdeHoras[hora] > 0) {
        --vetorQtdeHoras[hora];
    }

    return vetorQtdeHoras[hora];
}

void setQtdeHoras(char hora, char valor) {
    vetorQtdeHoras[hora] = valor;
}

float getQuantidadeInsulinaMinima() {
    return 0.1f;
}

float getQuantidadeInsulinaHora(char hora) {
    return getQuantidadeInsulinaMinima() * ((float)
        vetorQtdeHoras[hora]);
}
```

```
}

void delayTrocaMenu () {
    Delay_ms (50);
}

int getQuantidadeTotalInsulinaReservatorio () {
    return quantidadeTotalInsulinaReservatorio;
}

void resetQuantidadeTotalInsulinaReservatorio () {
    quantidadeTotalInsulinaReservatorio =
        QUANTIDADE_TOTAL_RESERVATORIO_INSULINA;
}

char executarLoopPrincipal () {
    return executarAplicacao;
}

void sairLoopPrincipal () {
    executarAplicacao = 0;
}
```

A.2 Config.h

```
#ifndef GLOBAL_CONFIG_H
#define GLOBAL_CONFIG_H

#include " ../lcd/lcd.h"

#define TRUE 1
#define FALSE 0
typedef char boolean;

void iniciliar ();
LCD_TYPE getTipoLcd ();
char getQtdeHoras (char hora);
char incrementaQtdeHoras (char hora);
char decrementaQtdeHoras (char hora);
void setQtdeHoras (char hora, char valor);
```

```
float getQuantidadeInsulinaMinima();  
float getQuantidadeInsulinaHora(char hora);  
void delayTrocaMenu();  
int getQuantidadeTotalInsulinaReservatorio();  
void resetQuantidadeTotalInsulinaReservatorio();  
char executarLoopPrincipal();  
void sairLoopPrincipal();  
#endif
```

APÊNDICE B – InsulinPump

B.1 IInsulinPump.h

```
#ifndef IINSULIN_PUMP_H
#define IINSULIN_PUMP_H

#include " ../ config / GlobalConfig .h"

extern unsigned int m_quantidade_injetada_total;
extern unsigned int m_quantidade_injetada_hora;
extern int m_intervalo_segundos_inteiro_para_injecao;
extern boolean m_injetar_habilitada;
extern int m_contador_injetar;
extern boolean m_flag_injetar;
extern boolean m_flag_mudou_hora;

typedef struct{
    void (* initialize ) ();
    void (* inject ) ();
    void (* start ) ();
    void (* stop ) ();
    int (* getQuantidadeInjetadaHora ) ();
} IInsulinPump;

#endif
```

B.2 InsulinPump.c

```
#include " InsulinPump .h"
#include " ../ config / GlobalConfig .h"
#include " ../ timerMotor / TimerMotorPasso .h"
#include " ../ Motor / MotorPassoFactory .h"
#include " ../ lcd / lcd .h"
#include " ../ lcd / lcdFactory .h"
```

```
#define QUANTIDADE_SEGUNDOS_EM_HORA 3600
#define QUANTIDADE_PASSOS_INFUSAO_MINIMA 8

unsigned int m_quantidade_injetada_total;
unsigned int m_quantidade_injetada_hora;
int m_intervalo_segundos_inteiro_para_injecao;
boolean m_injetar_habilitada;
int m_contador_injetar;
boolean m_flag_injetar;
boolean m_flag_mudou_hora;

// publicas
void initializePump();
void pumpInject();
void startInject();
void stopInject();
int getQuantidadeInjetadaHora();

void configuraHoraAtual();

IMotorPasso* m_motorPasso;

IInsulinPump InsulinPump = {
    &initializePump,
    &pumpInject,
    &startInject,
    &stopInject,
    &getQuantidadeInjetadaHora
};

void initializePump(){
    m_quantidade_injetada_total = 0;
    m_quantidade_injetada_hora = 0;
    m_intervalo_segundos_inteiro_para_injecao = 0;
    m_injetar_habilitada = FALSE;
    m_contador_injetar = 0;
    m_flag_injetar = FALSE;
    m_motorPasso = getMotorPasso( MOTOR_PASSO_PROTEUS );
```

```
        m_motorPasso->configure();
        m_flag_mudou_hora = FALSE;
        startInject();
    }

    void pumpInject() {
        if( m_flag_mudou_hora == TRUE ){
            configuraHoraAtual();
        }

        if( m_injetar_habilitada == TRUE ){
            if( m_flag_injetar == TRUE ){
                ++m_quantidade_injetada_total;
                ++m_quantidade_injetada_hora;
                m_flag_injetar = FALSE;
                m_motorPasso->write(QUANTIDADE_PASSOS_INFUSAO_MINIMA
                    );
            }
        }
    }

    void startInject() {
        configuraHoraAtual();
        m_injetar_habilitada = TRUE;
    }

    void stopInject() {
        m_injetar_habilitada = FALSE;
    }

    int getQuantidadeInjetadaHora() {
        return m_quantidade_injetada_hora;
    }

    void configuraHoraAtual() {
        // a cada quantos segundos intejar
        char qtdeHora = getQtdeHoras(horas);
        if( qtdeHora > 0 )
            m_intervalo_segundos_inteiro_para_injecao =
```

```
        QUANTIDADE_SEGUNDOS_EM_HORA/ qtdeHora ;  
else  
    m_intervalo_segundos_inteiro_para_injecao = 0;  
  
    m_contador_injetar = m_contador_injetar %  
        m_intervalo_segundos_inteiro_para_injecao ;  
  
}
```

B.3 InsulinPump.h

```
#ifndef INSULIN_PUMP_H  
#define INSULIN_PUMP_H  
  
#include "IInsulinPump.h"  
  
extern IInsulinPump InsulinPump ;  
  
#endif
```


APÊNDICE C – Lcd

C.1 lcd.h

```
#ifndef LCD_H
#define LCD_H

typedef struct{
    char (*configure)();
    char (*write)( char , char , char* );
    char (*writeCurrentCursor)( char* );
    char (*clear)();
    char (*shiftRigth)();
    char (*shiftLeft)();
    char (*cursorOff)();
} ILcd;

typedef enum { LCD_4BITS_2X16 = 0 } LCD_TYPE;

#endif
```

C.2 lcdFactory.h

```
#ifndef LCDFACTORY_H
#define LCDFACTORY_H

#include "lcd.h"

#ifdef __cplusplus
extern "C"
#endif

ILcd* getLcd( LCD_TYPE tipo );

#endif // LCDFACTORY_H
```

C.3 lcdFactory.c

```
#include "lcdFactory.h"
#include "lcd4bits2x16/lcd4bits2x16.h"

#ifdef __cplusplus
extern "C"
#endif
ILcd* getLcd( LCD_TYPE tipo )
{
    ILcd* retorno;

    switch( tipo )
    {
        case LCD_4BITS_2X16: retorno = &lcd4bits2x16; break;
        default: retorno = 0;
    }

    return retorno;
}
```

C.4 lcd4bits2x16

C.4.1 lcd4bits2x16.h

```
#ifndef LCD4BITS2X16_H
#define LCD4BITS2X16_H

#include "../lcd.h"

extern ILcd lcd4bits2x16;

#endif // LCD4BITS2X16_H
```

C.4.2 lcd4bits2x16.c

```
#include "Lcd4Bits2x16.h"

// LCD module connections
sbit LCD_RS at RE2_bit;
sbit LCD_EN at RE1_bit;
```

```

sbit LCD_D4 at RD4_bit;
sbit LCD_D5 at RD5_bit;
sbit LCD_D6 at RD6_bit;
sbit LCD_D7 at RD7_bit;

sbit LCD_RS_Direction at TRISE2_bit;
sbit LCD_EN_Direction at TRISE1_bit;
sbit LCD_D4_Direction at TRISD4_bit;
sbit LCD_D5_Direction at TRISD5_bit;
sbit LCD_D6_Direction at TRISD6_bit;
sbit LCD_D7_Direction at TRISD7_bit;

char configureLcd4Bits2x16();
char writeLcd4Bits2x16( char, char, char* );
char writeCurrentCursorLcd4Bits2x16( char* );
char clearLcd4Bits2x16();
char shiftrigthLcd4Bits2x16();
char shiftrightLcd4Bits2x16();
char cursorOffLcd4Bits2x16();

ILcd Lcd4Bits2x16 = {
    &configureLcd4Bits2x16,
    &writeLcd4Bits2x16,
    &writeCurrentCursorLcd4Bits2x16,
    &clearLcd4Bits2x16,
    &shiftrightLcd4Bits2x16,
    &shiftrightLcd4Bits2x16,
    &cursorOffLcd4Bits2x16
};

```

/*

obs: a emissao do registrador ADCON1 nos programas, acarreta o nao acionamento adequado do PORTA e consequentemente provoca o mal funcionamento do programa.

O pino RA4 do PIC eh dreno aberto, isto implica que devemos conectar resistor de pull-up quando formos trabalhar este pino como I/O de uso geral.

O PORTE tambem possui a funcao de conversores analogicos ou

digitais também. Devemos configurar o ADCON1 para trabalharmos o PORTE como IO de uso geral.

**/*

```
char configureLcd4Bits2x16(){
    trise = 0; // pinos D como saida
    ADCON1 = 0x06; // Todos os pinos A/D como I/O uso geral
    Lcd_Init();
    // Lcd8_Config(&PORTE,&PORTD,2,1,0,7,6,5,4,3,2,1,0);
    return 0;
}

char writeLcd4Bits2x16( char linha , char coluna , char* texto ){
    trisd = 0; // pinos D como saida
    Lcd_Out( linha , coluna , texto );
    trisd = 255; // pinos D como entrada
    return 0;
}

char writeCurrentCursorLcd4Bits2x16( char* texto ){
    trisd = 0; // pinos D como saida
    Lcd_Out_CP( texto );
    trisd = 255; // pinos D como entrada
    return 0;
}

char clearLcd4Bits2x16(){
    trisd = 0; // pinos D como saida
    Lcd_Cmd( _LCD_CLEAR );
    trisd = 255; // pinos D como entrada
    return 0;
}

char shiftRigthLcd4Bits2x16(){
    trisd = 0; // pinos D como saida
    Lcd_Cmd(_LCD_SHIFT_RIGHT);
    trisd = 255; // pinos D como entrada
    return 0;
}
```

```
char shiftLeftLcd4Bits2x16(){
    trisd = 0; // pinos D como saida
    Lcd_Cmd(_LCD_SHIFT_RIGHT);
    trisd = 255; // pinos D como entrada
    return 0;
}

char cursorOffLcd4Bits2x16(){
    trisd = 0; // pinos D como saida
    Lcd_Cmd(_LCD_CURSOR_OFF);
    trisd = 255; // pinos D como entrada
    return 0;
}
```


APÊNDICE D – Menu

D.1 IMenu.h

```
#ifndef IMENU_H
#define IMENU_H

static char indiceSelecionado = 0;
static char bloquearBotaoCima = 0;
static char bloquearBotaoBaixo = 0;
static char bloquearBotaoConfirmar = 0;
static char bloquearBotaoVoltarBloquear = 0;

typedef struct stIMenu{
    void (*mostrarMenu)();
    struct stIMenu* (*analisaBotoes)();
} IMenu;

#endif
```

D.2 MenuConfigurar.h

```
#ifndef MENU_CONFIGURAR_H
#define MENU_CONFIGURAR_H

#include "IMenu.h"

extern IMenu MenuConfigurar;

#endif
```

D.3 MenuConfigurar.c

```
#include "MenuConfigurar.h"
#include "../lcd/lcd.h"
```

```

#include " ../ lcd / lcdFactory .h"
#include " ../ config / GlobalConfig .h"
#include " MenuInicial .h"
#include " ../ config / GlobalConfig .h"

void mostrarMenuConfiguracao ();
IMenu* analisarBotoesMenuConfiguracao ();

IMenu MenuConfigurar = {
    &mostrarMenuConfiguracao ,
    &analisarBotoesMenuConfiguracao
};

void mostrarMenuConfiguracao () {
    ILcd* lcd = getLcd( getTipoLcd() );
    char convercao [7];
    float quantidade;

    sprintf(convercao , "Hora:_%d" , indiceSelecioneado+1);
    lcd->write(1,1 , convercao);
    quantidade = ((float) getQtdeHoras(indiceSelecioneado)) *
                  getQuantidadeInsulinaMinima();
    sprintf(convercao , "%.2f_un" , quantidade);
    lcd->write(2,1 , convercao);
}

IMenu* analisarBotoesMenuConfiguracao () {
    ILcd* lcd = getLcd( getTipoLcd() );
    // pra cima
    if(button(&PORTB,0,1,0)){
        bloquearBotaoCima = 1;
    }
    else if(bloquearBotaoCima && button(&PORTB,0,1,1)){
        incrementaQtdeHoras(indiceSelecioneado);
        bloquearBotaoCima = 0;
    }
    // Baixo
    if(button(&PORTB,1,1,0)){
        bloquearBotaoBaixo = 1;
    }

```



```
}  
else if(bloquearBotaoBaixo && button(&PORTB,1,1,1)){  
    decrementaQtdeHoras(indiceSelecioneado);  
    bloquearBotaoBaixo = 0;  
}  
  
// confirmar  
if( button(&PORTB,2,1,0) ){  
    bloquearBotaoConfirmar = 1;  
}  
else if(bloquearBotaoConfirmar && button(&PORTB,2,1,1)){  
    bloquearBotaoConfirmar = 0;  
    if(indiceSelecioneado >= 23) {  
        indiceSelecioneado = 0;  
        lcd->clear();  
        delayTrocaMenu();  
        return &MenuInicial;  
    }  
    else {  
        ++indiceSelecioneado;  
    }  
}  
  
// voltar-bloquear  
if( button(&PORTB,3,1,0) ){  
    bloquearBotaoVoltarBloquear = 1;  
}  
else if(bloquearBotaoVoltarBloquear && button(&PORTB,3,1,1)  
) {  
    bloquearBotaoVoltarBloquear = 0;  
    if(indiceSelecioneado > 0) {  
        --indiceSelecioneado;  
    }  
    else {  
        lcd->clear();  
        indiceSelecioneado = 0;  
        delayTrocaMenu();  
        return &MenuInicial;  
    }  
}
```

```

    }

    return &MenuConfigurar;
}

```

D.4 MenuExecutando.h

```

#ifndef MENU_EXECUTANDO_H
#define MENU_EXECUTANDO_H

#include "IMenu.h"

extern IMenu MenuExecutando;

#endif

```

D.5 MenuExecutando.c

```

#include "MenuExecutando.h"
#include "../lcd/lcd.h"
#include "../lcd/lcdFactory.h"
#include "../config/GlobalConfig.h"
#include "MenuInicial.h"
#include "../timerMotor/TimerMotorPasso.h"
#include "../insulinPump/InsulinPump.h"

void mostrarMenuExecutando();
IMenu* analisarBotoesExecutando();

IMenu MenuExecutando = {
    &mostrarMenuExecutando,
    &analisarBotoesExecutando
};

void mostrarMenuExecutando() {
    int porcentagem = 100*(getQuantidadeTotalInsulinaReservatorio
        () - m_quantidade_injetada_total)/
        getQuantidadeTotalInsulinaReservatorio();
    ILcd* lcd = getLcd( getTipoLcd() );
    unsigned char converter[15];
}

```

```

memset(&converter , 0x00 , sizeof(converter));

sprintf(converter , "%02d:%02d:%02d_F:%02d%%" , horas , minutos ,
        segundos , porcentagem);
lcd->write(1,1 , converter);

sprintf(converter , "%.2f/%.2f_un" , (float)((float)InsulinPump.
        getQuantidadeInjetadaHora()*getQuantidadeInsulinaMinima())
        ,
        getQuantidadeInsulinaHora(
        horas));
lcd->write(2,1 , converter);
}

IMenu* analisarBotoesExecutando(){
    ILcd* lcd = getLcd( getTipoLcd() );
    // confirmar
    if(button(&PORTB,2,1,0)){
        bloquearBotaoConfirmar = 1;
    }
    if(bloquearBotaoConfirmar && button(&PORTB,2,1,1)){
        lcd->clear();
        bloquearBotaoConfirmar = 0;
        delayTrocaMenu();
        InsulinPump.stop();
        return &MenuInicial;
    }

    // voltar-vloquear
    if(button(&PORTB,3,1,0)){
        bloquearBotaoVoltarBloquear = 1;
    }
    if(bloquearBotaoVoltarBloquear && button(&PORTB,3,1,1)){
        lcd->clear();
        bloquearBotaoVoltarBloquear = 0;
        delayTrocaMenu();
        InsulinPump.stop();
        return &MenuInicial;
    }
}

```

```

    delayTrocaMenu();
    return &MenuExecutando;
}

```

D.6 MenuInicial.h

```

#ifndef MENU_INICIAL_H
#define MENU_INICIAL_H

#include "IMenu.h"

extern IMenu MenuInicial;

#endif // LCD4BITS2X16_H

```

D.7 MenuInicial.c

```

#include "MenuInicial.h"
#include "../lcd/lcd.h"
#include "../lcd/lcdFactory.h"
#include "../config/GlobalConfig.h"
#include "MenuConfigurar.h"
#include "MenuExecutando.h"
#include "../insulinPump/InsulinPump.h"

void mostrarMenuInicial();
IMenu* analisarBotoesMenuInicial();

IMenu MenuInicial = {
    &mostrarMenuInicial,
    &analisarBotoesMenuInicial
};

void mostrarMenuInicial() {
    ILcd* lcd = getLcd( getTipoLcd() );

    if( indiceSelecioneado == 0 ){
        lcd->write(1,1, "_Iniciar");
    }
}

```

```
    else{
        lcd->write(1,1, "_Iniciar");
    }

    if( indiceSelecionado == 1 ){
        lcd->write(2,1, "-_Config._basal");
    }
    else{
        lcd->write(2,1, "_Config._basal");
    }
}

IMenu* analisarBotoesMenuInicial(){
    ILcd* lcd = getLcd( getTipoLcd() );
    // pra cima
    if( button(&PORTB,0,1,0) ){
        bloquearBotaoCima = 1;
    }
    if( bloquearBotaoCima && button(&PORTB,0,1,1) ){
        bloquearBotaoCima = 0;
        if( indiceSelecionado == 0 )
            indiceSelecionado = 1;
        else if( indiceSelecionado == 1 )
            indiceSelecionado = 0;
    }
    // Baixo
    if( button(&PORTB,1,1,0) ){
        bloquearBotaoBaixo = 1;
    }
    if( bloquearBotaoBaixo && button(&PORTB,1,1,1) ){
        bloquearBotaoBaixo = 0;
        if( indiceSelecionado == 0 )
            indiceSelecionado = 1;
        else if( indiceSelecionado == 1 )
            indiceSelecionado = 0;
    }

    // confirmar
    if( button(&PORTB,2,1,0) ){
```

```

        bloquearBotaoConfirmar = 1;
    }
    if(bloquearBotaoConfirmar && button(&PORTB,2,1,1)){
        bloquearBotaoConfirmar = 0;
        lcd->clear();
        if(indiceSelecioneado == 0){
            delayTrocaMenu();
            InsulinPump.start();
            return &MenuExecutando;
        }
        else if(indiceSelecioneado == 1) {
            indiceSelecioneado = 0;
            delayTrocaMenu();
            return &MenuConfigurar;
        }
    }

    delayTrocaMenu();
    return &MenuInicial;
}

```

D.8 MenuReservatorio.h

```

#ifndef MENU_RESERVATORIO_H
#define MENU_RESERVATORIO_H

#include "IMenu.h"

extern IMenu MenuReservatorio;

#endif

```

D.9 MenuReservatorio.c

```

#include "MenuReservatorio.h"
#include "../lcd/lcd.h"
#include "../lcd/lcdFactory.h"
#include "../config/GlobalConfig.h"
#include "MenuInicial.h"
#include "../timerMotor/TimerMotorPasso.h"

```

```
char reservatoriCheioSim = 0;

void mostrarMenuReservatorio();
IMenu* analisarBotoesMenuReservatorio();

IMenu MenuReservatorio = {
    &mostrarMenuReservatorio,
    &analisarBotoesMenuReservatorio
};

void mostrarMenuReservatorio(){
    ILcd* lcd = getLcd( getTipoLcd() );
    unsigned char converter[15];
    memset(&converter, 0x00, sizeof(converter));

    sprintf(converter, "%02d:%02d:%02d_F:%.0f%%", horas, minutos,
        segundos, 100);
    lcd->write(1,1, "Insulina_cheia");
    if(reservatoriCheioSim == 0){
        lcd->write(2,1, "(_)_sim, _(-)_nao");
    }
    else{
        lcd->write(2,1, "(-)_sim, _(*)_nao");
    }
}

IMenu* analisarBotoesMenuReservatorio(){
    ILcd* lcd = getLcd( getTipoLcd() );
    // pra cima
    if( button(&PORTB,0,1,0) ){
        bloquearBotaoCima = 1;
    }
    if(bloquearBotaoCima && button(&PORTB,0,1,1)){
        bloquearBotaoCima = 0;
        if(reservatoriCheioSim == 0)
            reservatoriCheioSim = 1;
        else if(reservatoriCheioSim == 1)
            reservatoriCheioSim = 0;
    }
}
```

```
    }  
    // Baixo  
    if (button(&PORTB,1,1,0)){  
        bloquearBotaoBaixo = 1;  
    }  
    if (bloquearBotaoBaixo && button(&PORTB,1,1,1)){  
        bloquearBotaoBaixo = 0;  
        if (reservatoriCheioSim == 0)  
            reservatoriCheioSim = 1;  
        else if (reservatoriCheioSim == 1)  
            reservatoriCheioSim = 0;  
    }  
  
    // confirmar  
    if (button(&PORTB,2,1,0)){  
        bloquearBotaoConfirmar = 1;  
    }  
    if (bloquearBotaoConfirmar && button(&PORTB,2,1,1)){  
        if (reservatoriCheioSim == 1)  
            resetQuantidadeTotalInsulinaReservatorio();  
  
        lcd->clear();  
        bloquearBotaoConfirmar = 0;  
        delayTrocaMenu();  
        return &MenuInicial;  
    }  
  
    delayTrocaMenu();  
    return &MenuReservatorio;  
}
```


APÊNDICE E – Motor

E.1 IMotorPasso.h

```
#ifndef IMOTORPASSO_H
#define IMOTORPASSO_H

typedef struct{

    char (*configure)();
    char (*write)( int );
    char (*setTempoEsperaPassoMs)( int );

} IMotorPasso;

typedef enum { MOTOR_PASSO_PROTEUS = 0, MOTOR_PASSO_GOLIAS = 1
} TIPO_MOTOR_PASSO;

#endif // IMOTORPASSO_H
```

E.2 MotorPassoFactory.h

```
#ifndef MOTORPASSOFACTORY_H
#define MOTORPASSOFACTORY_H

#include "IMotorPasso.h"

#ifdef __cplusplus
extern "C"
#endif

IMotorPasso* getMotorPasso( TIPO_MOTOR_PASSO tipoMotor );

#endif // MOTORPASSOFACTORY_H
```

E.3 MotorPassoFactory.c

```
#include "MotorPassoFactory.h"
#include "MotorProteus/MotorProteus.h"

#ifdef __cplusplus
extern "C"
#endif

IMotorPasso* getMotorPasso( TIPO_MOTOR_PASSO tipoMotor )
{
    IMotorPasso* retorno;

    switch( tipoMotor )
    {
        case MOTOR_PASSO_PROTEUS: retorno = &motorProteus; break;
        case MOTOR_PASSO_GOIAS: retorno = 0; break;
        default: retorno = 0;
    }

    return retorno;
}
```

E.4 MotorProteus

E.4.1 MotorProteus.h

```
#ifndef MOTORPROTEUS_H
#define MOTORPROTEUS_H

#include "../IMotorPasso.h"

extern IMotorPasso motorProteus;

#endif // MOTORPROTEUS_H
```

E.4.2 MotorProteus.c

```
#include "MotorProteus.h"

const char TAMANHO_COMBINACAO_ACIONAMENTO = 8;
const int TEMPO_DEFAULT_PASSO_MS = 500;

char configureMotorProteus();
```

```

char writeMotorProteus( int qtdePasso );
char setTempoEsperaPassoMsMotorProteus( int tempoMs );

char sequence[TAMANHO_COMBINACAO_ACIONAMENTO] = {0x02,0x06,0x04
    ,0x0c,0x08,0x09,0x01,0x03};
char passoAtual;
int tempoMsPasso;

IMotorPasso motorProteus = {
    &configureMotorProteus ,
    &writeMotorProteus ,
    &setTempoEsperaPassoMsMotorProteus
};

void delayEntrePassosDoMotor() {
    Delay_ms( 50 );
}

char configureMotorProteus() {
    TRISC = 0x00;
    passoAtual = 0;
    tempoMsPasso = TEMPO_DEFAULT_PASSO_MS;
}

char writeMotorProteus( int qtdePasso ){

    //      const int teste = tempoMsPasso;
    for(; qtdePasso > 0; --qtdePasso )
    {
        PORTC = sequence[passoAtual];
        delayEntrePassosDoMotor();
        passoAtual = (passoAtual+1)%
            TAMANHO_COMBINACAO_ACIONAMENTO;
    }
}

char setTempoEsperaPassoMsMotorProteus( int tempoMs ){
    tempoMsPasso = tempoMs;
}

```


APÊNDICE F – TimerMotor

F.1 ITimerMotor.h

```
#ifndef ITIMERMOTOR_H
#define ITIMERMOTOR_H

extern volatile int tick;
extern volatile int segundos;
extern volatile int minutos;
extern volatile int horas;

typedef struct{
    void (*configurar)();
    void (*iniciar)();
    void (*parar)();
} ITimerMotor;

#endif
```

F.2 TimerMotorPasso.h

```
#ifndef TIMERMOTORPASSO_H
#define TIMERMOTORPASSO_H

#include "ITimerMotor.h"

extern ITimerMotor timerMotorPasso;

#endif
```

F.3 TimerMotorPasso.c

```
#include "ITimerMotor.h"
```

```
volatile int tick;
volatile int segundos;
volatile int minutos;
volatile int horas;

void configurarTimerMotorPasso();
void iniciarTimerMotorPasso();
void pararTimerMotorPasso();

ITimerMotor timerMotorPasso = {
    &configurarTimerMotorPasso,
    &iniciarTimerMotorPasso,
    &pararTimerMotorPasso
};

void configurarTimerMotorPasso() {
    tick = 0;
    segundos = 0;
    minutos = 0;
    horas = 0;

    TMR0L = 192;
    INTCON = 0xA0;
}

void iniciarTimerMotorPasso() {
    TOCON = 0xC4;           // Enable watchdog timer
}

void pararTimerMotorPasso() {
    TOCON = 0x00;           // Disable watchdog timer
}
```

APÊNDICE G – Principal

G.1 insulin_pump.c

```

#include "src/lcd/lcdFactory.h"
#include "src/config/GlobalConfig.h"
#include "src/insulinPump/InsulinPump.h"
#include "src/timerMotor/TimerMotorPasso.h"
#include "src/menu/MenuReservatorio.h"

//verifica se o contador de segundos chegou no intervalo que
//deve ocorrer a infusao
void verifica_infusao(){
    /*incrementa o contador que e utilizado pra infundir a
    insulina*/
    ++m_contador_injetar;
    /*caso tenha chegado no intervalo que deve ocorrer a infusao
    */
    if(m_intervalo_segundos_inteiro_para_injecao > 0 &&
        m_contador_injetar >=
        m_intervalo_segundos_inteiro_para_injecao){
        m_flag_injetar = TRUE;
        m_contador_injetar = m_contador_injetar %
            m_intervalo_segundos_inteiro_para_injecao;
    }
}

void interrupt() {
    tick++;
    if (tick >= 10){
        ++segundos;
        verifica_infusao();
        if(segundos > 59){
            segundos = 0;
            ++minutos;
        }
    }
}

```

```

        if (minutos > 59) {
            minutos = 0;
            m_contador_injetar = 0;
            m_quantidade_injetada_hora = 0;
            m_flag_mudou_hora = TRUE;
            ++horas;
            if (horas > 23) {
                horas = 0;
            }
        }
    }

    tick = 0;
}

TMR0L = 192;                // on time
INTCON = 0x20;
}

void main() {

    unsigned char converter[15];
    ILcd* lcd = getLcd( getTipoLcd() );
    IMenu* menu = &MenuReservatorio;

    InsulinPump.initialize();
    timerMotorPasso.configurar();
    timerMotorPasso.iniciar();

    iniciliazar();

    lcd->configure();        // Initialize LCD

    lcd->clear();
    lcd->cursorOff();

    while( executarLoopPrincipal() ) {
        menu->mostrarMenu();
        menu = menu->analisarBotoes();
        InsulinPump.inject();
    }
}

```



```
}
```

```
}
```