HILGAD MONTELO DA SILVA

Simulação com Hardware In the Loop Aplicada a Veículos Submarinos Semi-Autônomos

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

São Paulo
2008

HILGAD MONTELO DA SILVA

Simulação com Hardware In the Loop Aplicada a Veículos Submarinos Semi-Autônomos

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

Área de Concentração:
Engenharia Mecatrônica

Orientador:
Prof. Dr. Celso Massatoshi Furukawa

São Paulo
2008

FICHA CATALOGRÁFICA

Montelo, Hilgad
Simulação com hardware In the loop aplicada a veículos submarinos semi-autônomos / H. Montelo. -- São Paulo, 2008.
92 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Submersíveis não tripulados 2.Arquitetura de software (Simulação) 3.Simulink 4.Matlab I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos II.t.

AGRADECIMENTOS

À minha família e amigos pelo incentivo permanente e torcida pelo sucesso na conclusão dessa dissertação.

A todos os professores do programa que não mediram esforços para transmitir e ensinar os seus conhecimentos ao longo dos dois anos de jornada. Às assistentes de programa que sempre se mostraram presentes e prestativas, na divulgação de informações e distribuição de material acadêmico, o meu reconhecimento e agradecimento.

Ao CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico, por haver concedido a bolsa de estudos para a realização deste mestrado.

À USP - Universidade de São Paulo que introduziu este programa de pós-graduação, inovando no ambiente do ensino acadêmico.

Sumário

LISTA DE FIGURAS 8
LISTA DE TABELAS 9
RESUMO 10
1. INTRODUÇÃO 12
1.1 Motivação...12
1.2 Visão Geral 14
1.3 Apresentação..16
2. OBJETIVOS 18
3. REVISÃO BIBLIOGRÁFICA 19
3.1 Veículos Submarinos Semi-Autônomos 19
3.2 Modelos Matemático de um UUV...23
3.2.1 Modelo Cinemático 24
3.2.2 Modelo Dinâmico 24
3.3 Simulação com Hardware In the Loop 25
4. METODOLOGIA..30
4.1 Simulação com Hardware in the Loop..30
4.1.1 Considerações de Hardware 30
4.1.2 Considerações de Software...32
4.2 Sistemas Operacionais de Tempo Real 33
5. MATERIAIS E MÉTODOS...35
5.1 Plataforma de Desenvolviimento..35
5.2 Ambiente Operacional de Tempo Real 37
5.3 Framework de Desenvolvimento em Sistemas de Tempo Real.................37
5.4 Modelagem e Simulação...40
5.5 Dinâmica do UUV...42
5.6 Sistema de Navegação 46
5.7 Controladores de baixo nível...46
5.8 Implementando um ambiente de Simulação com Hardware in the loop.......50
5.9 Conversor Simulink/RTW - Constellation 53
6. RESULTADOS 56

6.1 Modelo Dinâmico do UUV adaptado para execução no ambiente Vx Works ..56
6.2 Controlador do UUV adaptado para execução no computador embarcado 56
6.3 Simulações com o controlador embarcado 56
6.3.1 Simulação 1: Descolamento em direção a um ponto específico 57
6.3.2 Simulação 2: Controle de Profundidade 58
6.3.3 Simulação 3: Controle de Direção 59
6.3.4 Simulação 4: Controle de Velocidade 60
6.4 Integridade do contexto em ambiente de tempo real 61
7. CONCLUSÕES 62
REFERÊNCIAS BIBLIOGRÁFICAS 64
Apêndice A 69
Apêndice B 70
Apêndice C 75

LISTA DE FIGURAS

Figura 1. Subsistemas típicos encontrados em um UUV. 15
Figura 2. Veículo submarino com os 6 (seis) graus de liberdade associados ao veículo submarino com três modos de translação (*Swaying, Heaving, Surging*) e três modos de rotação (*Yaw, Pitch, Roll*). 23
Figura 3. Diagrama de blocos de um típico sistema com um simulador com hardware in the loop. 27
Figura 4. Placa padrão PC-104 com processador Geode 300 MHz. 36
Figura 5. Bússola digital magnética com 3 eixos. 36
Figura 6. Bloco correspondente ao modelo simulink em ambiente RTI's Constellation. 38
Figura 7. Sistema de coordenadas de um UUV – Blocos Simulink (à esquerda) e componente Constellation (à direita). 39
Figura 8. Modelo dinâmico ou planta de um UUV – Blocos Simulink (à esquerda) e componente Constelation (à direita). 39
Figura 9. Diagrama de blocos do MatLab/Simulink para a simulação do UUV. 41
Figura 10. O UUV Hornet consiste de dois propulsores horizontais, um propulsor vertical, uma câmera CCD e uma lanterna submersível de halogênio montados em uma estrutura tubular de PVC 42
Figura 11. Demonstração de corpo rígido com sistema de coordenadas. 44
Figura 12. Diagrama de blocos do controlador de profundidade PID. 47
Figura 13. Comportamento do controlador de modo deslizante no plano de erro. 48
Figura 14. Diagrama de blocos do controlador de direção em modo deslizante. 49
Figura 15. Diagrama de blocos de um controlador de velocidade PID. 49
Figura 16. Visão geral da arquitetura de simulação com hardware in the loop aplicada ao desenvolvimento e construção de Veiculo Submarino Autônomo. 51
Figura 17. Visualização do caminho percorrido pelo UUV. 57
Figura 19. Performance do controlador de direção em modo deslizante. 59
Figura 20. Performance do Controlador de velocidade PID. 60
Figura 21. Visualização de tempo de execução e troca de contexto entre tarefas.... 61

LISTA DE TABELAS

Tabela 1. Potenciais aplicações para um veículo submarino....................................20

Tabela 2. Alguns veículos submarinos fabricados na década de 90.........................21

Tabela 3. Configurações de alguns veículos submarinos..21

RESUMO

Veículos Submarinos Não Tripulados (UUVs – Unmanned Underwater Vehicles) possuem muitas aplicações comerciais, militares e científicas devido ao seu elevado potencial e relação custo-desempenho considerável quando comparados a meios tradicionais utilizados para a obtenção de informações provenientes do meio subaquático. O desenvolvimento de uma plataforma de testes e amostragem confiável para estes veículos requer o projeto de um sistema completo além de exigir diversos e custosos experimentos realizados no mar para que as especificações possam ser devidamente validadas. Modelagem e simulação apresentam medidas de custo efetivo para o desenvolvimento de componentes preliminares do sistema (software e hardware), além de verificação e testes relacionados à execução de missões realizadas por veículos submarinos reduzindo, portanto, a ocorrência de potenciais falhas. Um ambiente de simulação preciso pode auxiliar engenheiros a encontrar erros ocultos contidos no software embarcado do UUV além de favorecer uma maior introspecção dentro da dinâmica e operação do veículo. Este trabalho descreve a implementação do algoritmo de controle de um UUV em ambiente MATLAB/SIMULINK, sua conversão automática para código compilável (em C++) e a verificação de seu funcionamento diretamente no computador embarcado por meio de simulações. Detalham-se os procedimentos necessários para permitir a conversão dos modelos em MATLAB para código C++, integração do software de controle com o sistema operacional de tempo real empregado no computador embarcado (VxWORKS) e a estratégia de simulação com Hardware In The Loop (HIL) desenvolvida - A principal contribuição deste trabalho é apresentar de forma racional uma estrutura de trabalho que facilite a implementação final do software de controle no computador embarcado a partir do modelo desenvolvido em um ambiente amigável para o projetista, como o SIMULINK.

***Palavras-Chave**: Veículo Submarino, Hardware in the loop, Sistema Embarcado de Tempo Real, Modelo Dinâmico, Sistemas de Controle.*

ABSTRACT

Unmanned Underwater Vehicles (UUVs) have many commercial, military, and scientific applications because of their potential capabilities and significant cost-performance improvements over traditional means of obtaining valuable underwater information The development of a reliable sampling and testing platform for these vehicles requires a thorough system design and many costly at-sea trials during which systems specifications can be validated. Modeling and simulation provide a cost-effective measure to carry out preliminary component, system (hardware and software), and mission testing and verification, thereby reducing the number of potential failures in at-sea trials. An accurate simulation environment can help engineers to find hidden errors in the UUV embedded software and gain insights into the UUV operation and dynamics. This work describes the implementation of a UUV's control algorithm using MATLAB/SIMULINK, its automatic conversion to an executable code (in C++) and the verification of its performance directly into the embedded computer using simulations. It is detailed the necessary procedure to allow the conversion of the models from MATLAB to C++ code, integration of the control software with the real time operating system used on the embedded computer (VxWORKS) and the developed strategy of Hardware in the loop Simulation (HILS). The Main contribution of this work is to present a rational framework to support the final implementation of the control software on the embedded computer, starting from the model developed on an environment friendly to the control engineers, like SIMULINK.

Keywords*: Underwater Vehicle, Hardware In-the-Loop Simulation, Embedded Real-Time System, Dynamic Model, Control System.*

1. INTRODUÇÃO

Sistemas embarcados desempenham um importante papel no controle de típicos sistemas mecânicos. Considere o projeto de um sistema de piloto automático para um pequeno veículo submarino não-tripulado, por exemplo. Um pequeno erro no projeto pode levar a uma elevada ou total perda do veículo durante um simples teste de submersão. Mesmo a simulação de um sistema antes dos testes principais nem sempre ajuda devido à ausência de condições reais relacionadas ao meio e ao tempo mapeadas em termos de sinais analógicos e/ou digitais. Tal dilema tem levado a adoção de diferentes estratégias de testes e avaliação dos resultados no desenvolvimento do produto.

Testes tradicionais, muitas vezes referenciados como testes estáticos, consistem da avaliação de funcionalidades de um componente particular onde a ele são fornecidas entradas conhecidas e saídas mensuráveis. O aumento da pressão pelo lançamento de produtos mais rapidamente ao mercado e o apelo pela redução dos ciclos de desenvolvimento associados ao projeto têm ocasionado um aumento da necessidade de testes dinâmicos, onde o comportamento dos componentes é avaliado à medida que são submetidos a interações com o restante do sistema, ora de forma real ou simulada. Testes dinâmicos minimizam riscos relacionados à segurança e custos, além de abranger maiores condições de testes quando comparados aos testes estáticos. A aplicação desta estratégia para testes dinâmicos é conhecida como simulação com hardware in the loop (*HIL*).

1.1 Motivação

Simulação pura é muitas vezes utilizada para entender o comportamento de um sistema ou para predizer uma saída sob diferentes influências internas e/ou externas. Porém, se a simulação está sendo usada como uma base necessária para provar a viabilidade de controle, o risco de investimento pode ser fortemente reduzido utilizando-se uma abordagem baseada em simulação com hardware in the loop. Para a maioria dos sistemas reais, existem características que são desconhecidas ou muito complexas para modelar somente através de simulação pura. Se, por exemplo, é desejado estabelecer controle sobre o sistema de posicionamento de um veículo submarino, seria bastante arriscado ter que construir

todo o hardware no início sem considerar o sistema como um todo. Neste caso, uma boa prática de engenharia seria iniciar com uma simulação pura e, à medida que os componentes forem estabelecidos (muito possivelmente com o auxilio da própria simulação), eles podem ser fabricados e recolocados no laço de controle. Uma vez que os componentes físicos são adicionados ao laço de controle, características não modeladas podem ser investigadas e o controle pode ser refinado.

O uso de simulação com hardware in the loop diminui gastos e quantidade de iterações para a fabricação de maquinário e suas partes, além de tornar o desenvolvimento mais eficiente.

Antes o custo de construção de sistemas baseados em hardware in the loop eram bastante elevados, variando de US$ 50.000 a US $1.000.000. Hoje, máquinas baseadas em processadores Pentium® por exemplo, com clock na ordem de GHz já apresentam o poder de processamento necessário a um baixo custo, além das interfaces analógicas ou digitais de entrada/saída que vêm diminuindo cada vez mais o preço. Desse modo já é possível adquirir um sistema completo envolvendo computador, software e dispositivos de entrada/saída por valores na faixa de US$ 6000.

Assim, com custos menores, a simulação com hardware in the loop pode considerar melhores alternativas existentes no mercado para os processadores embarcados e dispositivos de entrada e saída. Processadores Digitais de Sinais (do inglês: *DSP – Digital Signal Processors*) são de baixo custo e, atualmente, existem muitos softwares de desenvolvimento que geram códigos de máquina para eles.

A simulação com hardware in the loop surge para facilitar o trabalho do engenheiro de controle. Sistemas podem ser desenvolvidos rapidamente utilizando apenas o modelo conceitual por um único engenheiro de controle. Neste ambiente de desenvolvimento, o engenheiro de controle não necessariamente é responsável por implementar o sistema na perspectiva da programação; ele simplesmente modela interações e fluxos através de diagramas de blocos. O engenho de controle pode focar-se inteiramente sobre características do projeto, evitando levar horas de escrita e/ou depuração de códigos ou mesmo ter que lidar com as complexidades inerentes aos sistemas de tempo real. Eventualmente o projeto precisará ser transferido para o sistema "alvo" (*target*). Desde que o algoritmo de controle é conhecido, passa a ser conhecida também sua carga computacional, podendo-se

portanto selecionar um processador adequado para a execução das tarefas. Além disso, há a vantagem de que o projeto avaliado servirá como "*template*" para a especificação e validação do software de controle. Técnicas de validação de componentes, seja de forma unitária ou relacionadas ao sistema de programação, são facilmente aplicadas ao código utilizando o controlador do simulador como uma referência. Neste estágio, o risco de desenvolvimento pode ser apenas afetado pelos prazos, não pela tecnologia.

A técnica de simulação com hardware in the loop aplica-se a todos os sistemas, sejam eles grandes ou pequenos, processos industriais e mesmo durante o desenvolvimento de novos produtos. Onde quer que exista interação entre simulação e o mundo real, existe uma oportunidade para a abordagem de simulação com hardware in the loop.

1.2 Visão Geral

Um Veículo Submarino Não Tripulado (UUV - *Unmanned Underwater Vehicles*) consiste de uma coleção de complexos subsistemas (navegação, controle, atuadores, alimentação, etc) integrados dentro de uma plataforma embarcada, como mostra a Figura 1. O desenvolvimento de um sistema completa ou parcialmente autônomo em conjunto com a busca por águas mais profundas, provoca um aumento na complexidade de cada um destes módulos ou subsistemas e interconexões originando, portanto, as seguintes necessidades:

- Habilidade de desenvolver testes incrementais e de integração de componentes envolvendo hardware e/ou software por diferentes equipes sob diferentes condições físicas;
- Uma maneira viável de avaliar os diferentes algoritmos de controle e a performance completa do sistema;
- Um mecanismo para rever as missões realizadas simplificando a analise de dados pós-operação;
- Um ambiente de treinamento para operadores de UUVs que minimize custos e riscos;
- Um rápido e eficiente desenvolvimento das tarefas de controle, iniciando na construção do modelo e finalizando com o hardware in the loop para promover a otimização do projeto estrutural do controlador.

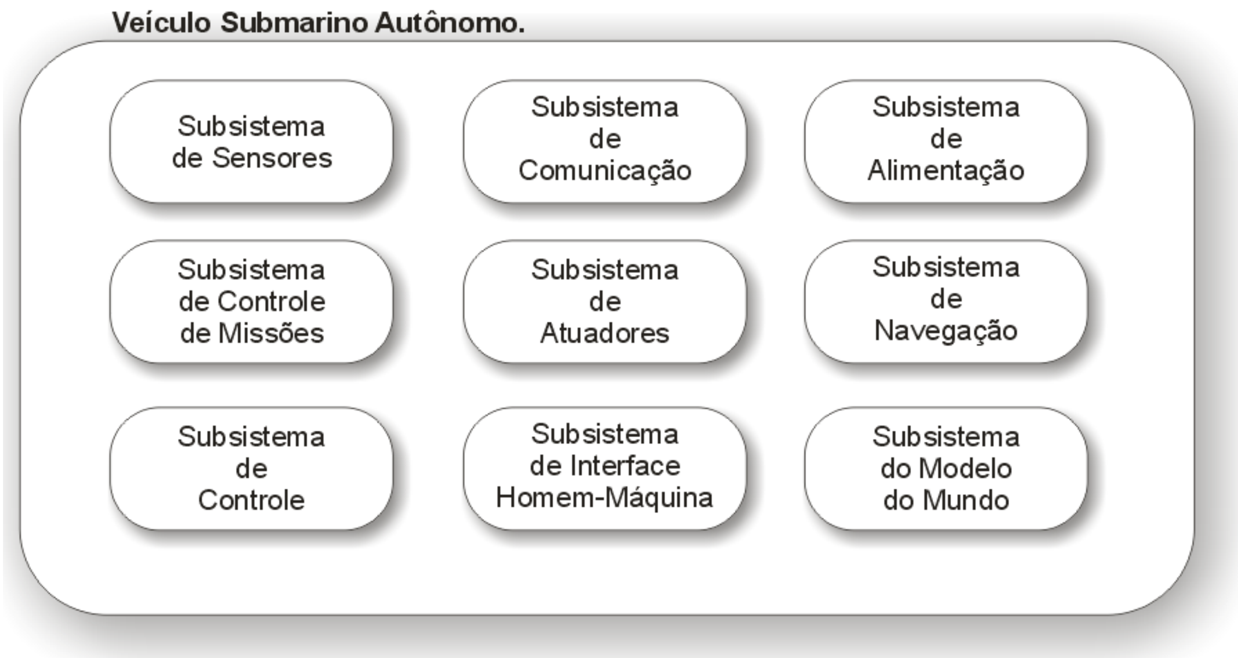

Figura 1. Subsistemas típicos encontrados em um UUV.

O trabalho consiste de um ambiente de hardware e software distribuído que endereça estas necessidades, permitindo que subsistemas reais e simulados possam trabalhar de forma conjunta e completamente interoperável durante todas as fases de simulações e testes a que é submetido o veículo. Pode ser executado da seguinte forma:

- Inteiramente em modo simulado para questões de demonstração e/ou avaliação do sistema;
- Usando partes simuladas (abstração do hardware) e partes reais dos diferentes componentes do sistema, favorecendo testes e integração do sistema;
- Utilizando todos os componentes reais de forma a promover a infra-estrutura básica para operações de campo.

A habilidade de unir subsistemas reais e simulados introduz dificuldades devido à alta interação, por exemplo, de sensores, atuadores e componentes físicos encontrados no ambiente. Especificamente:

- Quando a operação em tempo real é impossibilitada (pela complexidade da simulação ou pela sobrecarga durante o processo de comunicação);

- Dados devem permanecer consistentes através dos componentes necessários ao sistema durante a operação;
- A arquitetura de simulação com hardware in the loop deve manter um estado simulado do mundo que deve ser calibrada com o mundo real favorecendo a análise de dados.

A arquitetura de hardware in the loop aplicada a veículos submarinos consiste de um projeto colaborativo entre diferentes ciências e áreas de conhecimento, incluindo conceitos encontrados na engenharia naval, mecatrônica, oceanografia, técnicas de produção e exploração associadas a tecnologias off-shore.

No seu essencial, este trabalho propõe uma metodologia para a modelagem e organização de bibliotecas de modelos tendo como finalidade a realização de experiências com *hardware-in-the-loop*. É comumente aceito que modelos mais complexos exigem desempenhos computacionais e tempos de simulação maiores, sendo apropriada a simulação em tempo não real. O esforço incide então na procura de modelos de baixa complexidade, que poderão ser semi-empíricos, mas que traduzam de forma eficaz os comportamentos dos componentes, muito especialmente os mais importantes para o teste de controladores.

1.3 Apresentação

O texto apresentado a seguir é organizado com a seguinte estrutura:

- Capítulo 2: Declaração dos objetivos deste trabalho;

- Capítulo 3: Apresenta uma revisão bibliográfica sobre o desenvolvimento e pesquisa de veículos submarinos não tripulados, modelos matemáticos de UUVs e as aplicações de simuladores com hardware in the loop;

- Capítulo 4: Apresenta a metodologia utilizada para o maior desenvolvimento sobre o tema do trabalho, avaliando os diferentes componentes e comportamentos - hardware e software – e o impacto que cada um pode trazer ao resultado final;

- Capítulo 5: Indica os materiais e métodos utilizados para uma adequada formulação do problema, avaliação dos dados e informações e obtenção dos resultados sobre a tecnologia de hardware in the loop. Apresenta a utilização e integração de ferramentas desenvolvidas para a geração de código para execução em sistemas embarcados e de tempo real, além de permitir o acompanhamento de seus estados;

- Capítulo 6: Apresenta alguns resultados obtidos a partir das implementações referenciando, quando necessário, abordagens tomadas no capítulo anterior;

- Capítulo 7: Conclusões e sugestões para trabalhos futuros;

2. OBJETIVOS

Este trabalho teve como objetivo o desenvolvimento de ambiente capaz de

- Avaliar a utilização de recursos computacionais exigidos pelo software de controle;
- Possibilitar a seleção de uma plataforma de execução mais adequada com base na análise de medidas dos tempos necessários para execução de cada ciclo referente aos algoritmos de controle;
- Permitir a realização de simulações com a presença ou não do veículo submarino, redirecionando, quando necessário, o conjunto de entradas e saídas representadas por sensores a atuadores do sistema;
- Construir de forma transparente, o conjunto de código necessário para a execução em ambiente de tempo real, equivalente ao modelo de controle de entrada;
- Permitir a validação do software de controle por meio de comparações entre seu comportamento quando executado em ambiente real e simulado.

Não são considerados, neste trabalho, o desenvolvimento ou construção de um modelo de controle para veículos submarinos, sendo este, porém, utilizado como entrada necessária para iniciar todo o processo de análise e validação que o ambiente se propõe a realizar. Como o foco deste trabalho se concentra nas etapas de codificação e testes do algoritmo de controle no computador que será embarcado, a partir de um projeto de controlador feito por terceiros, utilizou-se como caso para estudo o modelo dinâmico e o algoritmo de controle de um UUV já desenvolvido, HORNET (BRAUNL, et al., 2004), disponíveis na literatura. Pode-se assim comparar os resultados das simulações realizadas com os resultados publicados, para verificar se o controlador embarcado implementado funcionou como esperado.

Também não foi considerada a elaboração de um modelo de carga de processamento ocasionado por eventos relacionados à comunicação e troca de mensagens que possam ocorrer entre o veículo submarino e outros computadores de superfície, que pode gerar picos de processamento consideráveis de forma aleatória.

3. REVISÃO BIBLIOGRÁFICA

3.1 Veículos Submarinos Semi-Autônomos

Impulsionada pela crescente demanda por operações submarinas nos campos de telecomunicações e extração de óleo, a robótica submarina tem se desenvolvido rapidamente. Robôs submarinos, ou UUVs têm há muito estado presentes em operações comerciais, militares e em pesquisa oceanográfica. No caso dos veículos semi-autônomos, apesar de se manter um cabo umbilical conectando-o a uma base de operações, o sistema de controle embarcado tem capacidade de realizar manobras complexas com pouca ou nenhuma interferência humana. Quando conveniente, o umbilical permite que um operador assuma o controle manual do veículo, além de oferecer um canal de comunicação on-line.

Neste contexto, veículos submarinos não tripulados (UUVs) semi-autônomos têm se mostrado como uma nova tendência em um meio tradicionalmente dominado pelos chamados ROVs (*Remotely Operated Vehicles*), que são comandados à distância através de um cabo umbilical (YUH, 2000).

Avanços tecnológicos recentes indicam ainda que operações como inspeção de equipamentos, canalizações e cabos submarinos serão em breve realizadas rotineiramente por UUVs. Este fato motiva o desenvolvimento de algoritmos de manobra e controle para a execução deste tipo de missão. Os obstáculos técnicos a serem vencidos em robótica submarina diferem muito daqueles presentes em ambientes aéreos. A rápida atenuação de ondas eletromagnéticas dentro da água impede a comunicação por rádio, a navegação com auxílio de GPS e a obtenção de imagens de vídeo além de uma pequena distância (ALMEIDA, 1999). Contornar estas limitações representa um desafio para a evolução e para o emprego prático de robôs submarinos.

Veículos submarinos semi-autônomos são objetos de intensa pesquisa em laboratórios ao redor do mundo. Em breve estes veículos realizarão rotineiramente missões de inspeção de equipamentos, tubulações e cabos, mapeamento de regiões submarinas, atendendo às demandas da indústria petrolífera e de telecomunicações.

Um grande desafio para o projeto de UUVs está em sua navegação e controle. Estes requerem informações confiáveis sobre a localização e atitude do veículo, o que é obtido através de complexos sistemas de fusão sensorial. O controle de alto-nível, permitindo a programação de combinações de tarefas como navegação, seguimento de fundo ou retorno à base, ainda constitui objeto de intensa pesquisa.

Devido à dificuldade de se encontrar parâmetros precisos para os modelos dinâmicos de veículos submarinos, é muito comum a prática de métodos de controle robusto, em particular o controle deslizante. Aplicações de controle deslizante de primeira ordem em UUVs são descritas em (HEALEY, et al., 1993), (CRISTI, et al., 1990) e (RODRIGUES, et al., 1996). Pan-Mook e colaboradores (PAN-MOOK, et al., 1999) apresentam a utilização de controle quasi-deslizante no plano vertical como solução para realização de controle baseado em sensor para seguimento de fundo, utilizando sonares que fornecem leituras em intervalos de tempo longos e irregulares.

Além do controle deslizante, diversos métodos vêm sendo investigados para o controle de UUVs, como o PID (JALVING, 1994), o controle Fuzzy (DEBITETTO, 1995), controle Fuzzy-deslizante (LEPAGE, et al., 2000) e controle por redes neurais artificiais (KAWANO, et al., 2002).

A arquitetura de sensores em UUVs e o problema de controle baseado em sensor para seguimento de fundo são abordados em (SANTOS, 1995) e (CREUZE, 2002). Martins-Encarnação trata da geração de trajetórias e a coordenação de um veículo autônomo submarino e um veículo autônomo de superfície (MARTINS-ENCARNAÇÃO, 2002).

A Tabela 1 apresenta algumas áreas de aplicação em que é possível a utilização de UUVs (SMITH, 1995), (KOK, 1984), (ADAM, 1985), (YORGER, 1991), (BLIDBERG, 1991), (BELLINGHAM, 1993), (BELLINGHAM, 1996):

Área	Aplicação
Ciência	- Mapeamento marítimo.
	- Respostas rápidas a eventos oceanográficos e geotérmicos.
	- Retirada de amostras para análise geológica.
Ambiente	- Inspeção de estruturas submarinas (dutos, represas, etc).

	- Monitoramento de longo prazo (vazamentos radiativos, poluição).
Militar	- Busca e desarmamento de minas submarinas.
	- Sensoriamento submarino off-board.
Indústrias de mineração e petróleo	- Verificação e avaliação de recursos oceânicos.
	- Prospecção de petróleo em águas profundas.
	- Construção e manutenção de estruturas marítimas.
Outras aplicações	- Inspeção interna de tanques de navios.
	- Inspeção de plantas de energia nuclear.
	- Resgate/Coleta de dispositivos submersos.
	- Inspeção e instalação de cabos de força e comunicação submarinos.
	- Entretenimento.

Tabela 1. Potenciais aplicações para um veículo submarino.

A Tabela 2 apresenta uma série de veículos submarinos desenvolvidos na década de 90, com seus respectivos objetivos, fabricantes e profundidades alcançadas (KOK, 1984), (ROBINSON, 1986), (TUCKER, 1986), (ASHLEY, 1993).

Ano	Veículo	Objetivo	Profundidade	Fabricante
1990	UROV-2000	Bottom survey	2000 m	JAMSTEC, Yokosuka, Japan
1991	AROV	Busca e mapeamento	N/A	SUTEC, Linkoping, Sweden
1992	Phoenix	Testes	10 m	Naval Postgraduate School, Monterey, CA
1993	Ocean Voyager II	Missão Científica	6000 m	Florida Atlantic University, Boca Raton, FL.
1993	Odyssey II	Missão Científica	6000 m	MIT Sea Grant, Cambridge, MA
1994	OTTER	Testes	1000 m	MBARI, CA.
1994	Explorer	Inspeção de tubulações	1000 m	Shenyang Institute of

				Automation, China
1995	Autosub-1	Monitoramento ambiental	750 m	Southampton Oceanography Centre, UK.
1996	Theseus	Missão sob o gelo do mar ártico	1000 m	ISE, Canadá.
1997	VORAM	Testes	200 m	Korea Research Inst. of Ships & Ocean Engr., Korea
1998	AMPS	Militar	200 m	Pacific Missile Range Facility, Kekaha, HI
1998	Solar AUV	Testes	N/A	Autonomous Undersea Systems Institute, NH
1998	SIRENE	Prospecção submarina	4000 m	DESIBEL, European project led by IFREMER, France
1999	SAUVIM	Intervenção militar/científica	6000 m	ASL, University of Hawaii, Honolulu, HI

Tabela 2. Alguns veículos submarinos fabricados na década de 90.

A Tabela 3; a seguir; apresenta o resumo das configurações apresentadas por alguns desses veículos (ADAM, 1991), (WANG, 1993), (ADAKAWA, 1995) , (BRUTZMAN, 1996):

Veículo	Sistema Operacional	Propulsores	Sistema de sensoriamento
Phoenix	OS-9	6	Sonar de altitude, sonar para prevenir colisão, giroscópio, etc.
Ocean Voyager II	VxWorks	1	Velocímetro sônico, sensor de pressão, sonar, modem de RF, altímetro, etc.
Odyssey II	OS-9	1	Altímetro, sensor de temperatura, sonar para lidar com obstáculos, modem acústico, pinger, etc.
OTTER	VxWorks	8	Sensor de inclinação, sensor de movimento, sensor de pressão, sistema de posicionamento sônico, etc.

Tabela 3. Configurações de alguns veículos submarinos.

3.2 Modelos Matemático de um UUV

De acordo com a prática comum em Robótica Submarina, as equações do modelo dinâmico a seis graus de liberdade de um veículo são representadas com o auxílio de um sistema de coordenadas global e um local (FOSSEN, 2002) - Como mostra a Figura 2 - Este padrão é proposto pela SNAME (*Society of Naval Architects and Marine Engineers*).

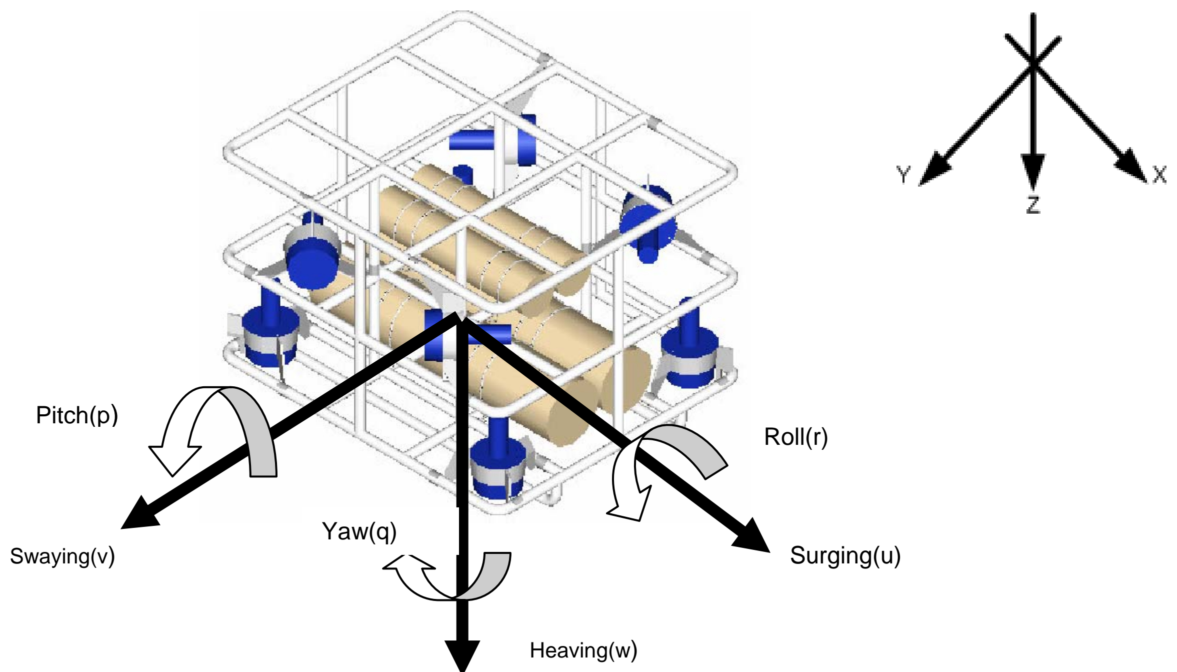

Figura 2. Veículo submarino com os 6 (seis) graus de liberdade associados ao veículo submarino com três modos de translação (*Swaying, Heaving, Surging*) e três modos de rotação (*Yaw, Pitch, Roll).*

O sistema de coordenadas local é em geral localizado no centro de massa do veículo, e é definido de tal forma que seus eixos coincidam com seus eixos principais de inércia. Desta forma, tira-se vantagem da simetria do veículo conduzindo a um modelo mais simples. Este sistema tem componentes de movimento dadas pelo vetor de velocidades lineares $\mathbf{v_1} = [u, v, w]^T$ e angulares

$\mathbf{v_2} = [p, q, r]^T$. Desse modo, a equação 1 representa o vetor geral de velocidades para o veículo.

$$\mathbf{v} = [\mathbf{v}_1^T, \quad \mathbf{v}_2^T] = [u, v, w, p, q, r]^T \tag{1}$$

O vetor de posição $\eta_1 = [x, y, z]^T$ e orientação $\eta_2 = [\phi, \theta, \psi]^T$ é expresso em relação ao sistema absoluto, onde sua forma geral é dada pela equação 2.

$$\eta = [\eta_1^T, \quad \eta_2^T] = [x, y, z, \phi, \theta, \psi]^T \tag{2}$$

na qual ϕ representa o ângulo de rolagem, θ o ângulo de arfagem e ψ a guinada.

3.2.1 Modelo Cinemático

A relação entre as velocidades no sistema de coordenadas local e absoluto é dada pela equação 3.

$$\begin{bmatrix} \dot{\boldsymbol{\eta}}_1 \\ \dot{\boldsymbol{\eta}}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{J}_1(\eta_2) & 0 \\ 0 & \mathbf{J}_2(\eta_2) \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v_2} \end{bmatrix} \tag{3}$$

em que $\mathbf{J}_1(\eta_2)$ é a matriz jacobiana de rotação que fornece as componentes de velocidade linear v_1 no sistema absoluto e $\mathbf{J}_2(\eta_2)$ é a matriz jacobiana que relaciona a velocidade angular v_2 com a atitude do veículo no sistema absoluto.

3.2.2 Modelo Dinâmico

A dinâmica de veículos submarinos é obtida com base nos princípios do movimento de corpos rígidos. Seu movimento é governado por componentes de inércia, por acelerações de *Coriolis*, forças centrífugas e forças hidrodinâmicas, que são provocadas por transferências de energia entre o fluido e o veículo devido ao deslocamento relativo entre eles. Para levar em conta a inércia do fluido ao redor do veículo, ou seja, a transferência de energia cinética entre o veículo e o fluido, emprega-se o conceito de massa adicionada, que corresponde ao fluxo adicional de massa, considerado devido ao movimento de um fluido (ROBERSON, et al., 1997).

As equações não-lineares de movimento referentes aos seis graus de liberdade do veículo podem ser expressas de forma compacta através da equação matricial, equação 4 (FOSSEN, 2002):

$$\mathbf{M}\dot{\mathbf{v}} + \mathbf{C}(\mathbf{v}) + \mathbf{D}(\mathbf{v})\mathbf{v} + g(\eta) = \tau \quad (4)$$

em que $\mathbf{M} = \mathbf{M_{RB}} + \mathbf{M_A}$ é a matriz de inércia com massa adicionada, $\mathbf{C} = \mathbf{C_{RB}} + \mathbf{C_A}$ é a matriz de *Coriolis* e termos centrípetos, com massa adicionada, D é a matriz de amortecimento, g é o vetor de forças e momentos gravitacionais e τ é o vetor de forças e torques dos atuadores, para os quais utilizam-se modelos lineares.

3.3 Simulação com Hardware In the Loop

Hardware-in-the-loop Simulation (HILS) refere-se a uma simulação onde alguns dos componentes são reais e não simulados. É considerada a técnica mais segura e de mais baixo custo para teste de componentes reais em ambientes virtuais. A maioria dos componentes reais é substituída por modelos matemáticos e os componentes a testar são inseridos na malha fechada. Uma das razões para inserir componentes numa simulação surge, muitas vezes, pela inexistência de um conhecimento cabal das suas características, ou as estas são muito complexas, ou então quando existe a necessidade de teste dos próprios componentes reais, como é o caso do teste de controladores. Outra das razões advém do próprio processo de projeto de novos sistemas de controle que cumpram determinadas especificações. Pode-se, por exemplo, partir de uma simulação pura em tempo real, com os modelos de todos os componentes do sistema, de modo a cumprir determinadas especificações; à medida que os componentes vão sendo fabricados, vão substituindo os respectivos modelos matemáticos na simulação, de tal forma que as características dos outros componentes poderão ser re-ajustadas de modo a cumprir as especificações iniciais; este processo permite, por exemplo, reduzir o número de iterações na fabricação de peças para os diferentes componentes do sistema.

A indústria aeroespacial está entre as primeiras a desenvolver este tipo de simulação com o objetivo de desenvolver formas viáveis de realizar testes sobre sistemas de controle de vôo, muito embora, desde então, as aplicações passaram a

ser bastante diversificadas, como indicado por (MACLAY, 1997), (ALLES, et al., 1994), (KEY, 1987). No domínio subaquático, simuladores *HIL* são utilizados para estudo de controle e visualização (BRUTZMAN, et al., 1992), capacidade de sensoriamento (YURODA, et al., 1996) e testes de subsistemas (DUNO, et al., 1994).

A tecnologia de simulação com hardware in the loop pode ser encontrada em quase todos os lugares onde são necessários testes mais realísticos com componentes de um sistema antes de sua construção final. Por exemplo:

- Kim e colaboradores apresentam um sistema de cirurgia invasiva com operações hápticas através de simulação com hardware in the loop (KIM, et al., 2002);

- Senta e Colaboradores implementam um sistema de controle para discos rígidos, simuladores de forças que interagem, sobre veículos automotivos (SENTA, et al., 2002);

- Cita-se ainda uma coleção de trabalhos que descrevem um software de simulação de tráfego (PRASETIAWAN, et al., 1999) (ZHANG, et al., 2003), (ZHEN, et al., 2004).

A simulação *HIL* é, também, uma ferramenta bastante útil para avaliação e desenvolvimento de controladores, oferecendo um risco nulo na experimentação de diferentes técnicas e metodologias de controle sem necessidade da plataforma real para teste (ANAKWA, et al., 2002). Desta forma, é possível poupar investimentos e evitar consequências perigosas resultantes de erros no projeto inicial dos controladores, permitindo, assim, a identificação e eliminação desses erros. O uso da *HILS* para as diferentes regiões de operação, incluindo modos com falhas, permite a seleção das estratégias adequadas de controle na plataforma real, pois facilita a repetição de testes de desempenho.

A Figura 3 mostra as diferentes possibilidades de interação de componentes reais com versões simuladas de outros componentes num sistema de controle típico.

Como se pode observar, existem alguns caminhos na malha que não são acessíveis; por exemplo, não é possível um processo real ser monitorado através de sensores simulados ou atuadores virtuais atuarem em processos reais.

Em alguns casos, podem ser empregadas técnicas denominadas de *load simulation*, que consistem na utilização de um processo simulado para avaliação de desempenho de atuadores reais (JANSSON, et al., 1994). Por outro lado, o monitoramento de processos simulados através de sensores reais, embora com aplicações não tão evidentes, pode ser utilizado quando se quer avaliar o desempenho de um determinado sensor; neste caso é necessária uma interface adequada, que pode envolver um atuador real para converter os resultados da simulação em quantidades mensuráveis pelo sensor.

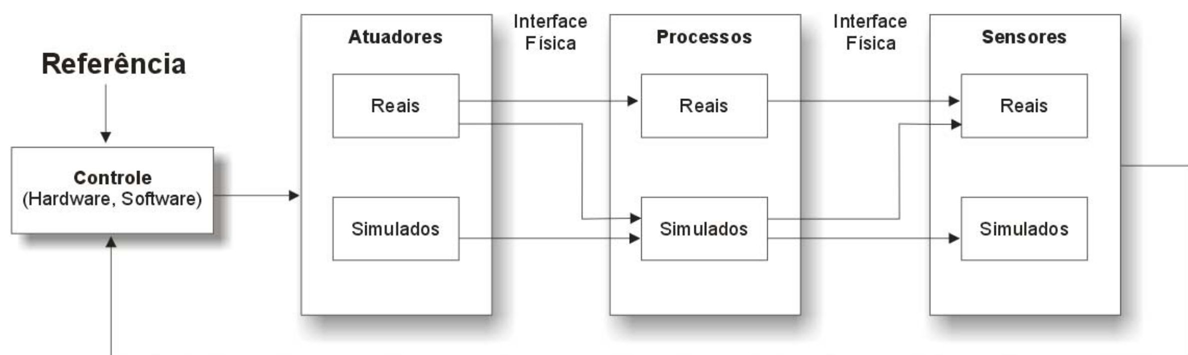

Figura 3. Diagrama de blocos de um típico sistema com um simulador com hardware in the loop.

Uma condição necessária para a realização de experiências de HILS é a capacidade de simulação em tempo real dos modelos matemáticos dos componentes.

3.3.1 Simulação em Tempo Real

A simulação em tempo real (*STR*) pode ser definida como aquela em que as variáveis de um sistema simulado representam fielmente não só os valores do sistema real, mas também os instantes temporais associados a esses valores. A simulação em tempo real envolve problemas diferentes das outras simulações no que diz respeito aos métodos numéricos a utilizar. Na simulação em tempo não real, o objetivo principal dos métodos numéricos é a diminuição do tempo de simulação, para uma dada precisão dos valores obtidos para as variáveis. Estes métodos

recorrem a sofisticados mecanismos que ajustam o passo de integração e/ou controlam a ordem do método para manter a estabilidade numérica e a obtenção da precisão pretendida. Na simulação em tempo real com *HIL*, o passo de integração é fixo pois, devido às contingências da simulação, o hardware de simulação pode ter de se comunicar com periféricos de hardware durante a simulação. A comunicação entre a simulação e os periféricos ocorre com um determinado tempo de amostragem e a simulação tem de fornecer os resultados dentro desse intervalo de amostragem. Desta forma, o principal objetivo da simulação em tempo real é garantir que o tempo de cálculo, para obtenção dos valores para as variáveis, é inferior ao passo de integração.

3.3.2 Técnicas e métodos numéricos na STR

Na simulação em tempo real são usados métodos de integração de passo fixo que podem ser explícitos ou implícitos (HEATH, 1997), (LAMBERT, 1991) ou (SHAMPINE, 1994). Os métodos de integração de passo fixo explícitos têm particulares dificuldades com os sistemas rígidos (sistemas em que a relação entre os valores próprios, correspondentes às dinâmicas rápidas e às dinâmicas lentas, é grande), pois o passo de integração é limitado por problemas de estabilidade numérica. Por exemplo, quando se utiliza o método explícito de Euler, o passo de integração tem que ser inferior à menor constante de tempo do sistema, para garantir a estabilidade do método (HEATH, 1997); se o passo for demasiado grande a trajetória pode oscilar e divergir. A solução para estes comportamentos instáveis passa, normalmente, pela utilização de métodos implícitos. Contudo, o uso destes métodos implica a resolução, em cada passo, de grandes sistemas de equações não lineares cuja dimensão é pelo menos igual ao número de variáveis de estado do sistema. Para a resolução destes sistemas de equações não lineares são usados métodos numéricos iterativos. O processo iterativo termina quando a precisão desejada é atingida e, portanto, não é possível prever o número de iterações necessário em cada passo. Além disso, nos sistemas de equações de estrutura variável (resultantes, por exemplo, da modelagem de sistemas híbridos), os sistemas de equações a resolver podem ser diferentes em diferentes regiões de operação do sistema modelado e, portanto, não se pode tirar partido da estrutura do modelo na simulação em tempo real (SCHIELA, et al., 2002). Os métodos implícitos resolvem o

problema da estabilidade numérica e permitem a utilização de passos de integração maiores, sendo a restrição ao passo de integração definida através da precisão que se pretende obter. Em aplicações industriais os sistemas a simular são, muitas vezes, modelados por grandes sistemas de DAEs rígidas. Na maioria dos casos a rigidez é devida a componentes com dinâmicas rápidas (por exemplo componentes hidráulicos, elétricos, controladores, etc.) quando comparadas com as escalas temporais de interesse (por exemplo o movimento de peças mecânicas). Como se depreende do exposto, os métodos explícitos e os métodos implícitos têm vantagens e desvantagens quando usados na simulação em tempo real. Nos últimos anos foram apresentadas alternativas que exploram as vantagens de ambos os métodos para permitir a simulação em tempo real de sistemas mais complexos, nomeadamente os sistemas multi-domínio que resultam, normalmente, em sistemas de equações rígidas. Uma das técnicas, chamada *multirate integration*, consiste em utilizar a discretização explícita com passos de integração diferentes, ajustados às dinâmicas lentas e às dinâmicas rápidas. A principal vantagem resulta da inexistência de métodos iterativos. Esta técnica pode ser usada em sistemas onde as equações diferenciais possam ser divididas em subsistemas, pouco acoplados ou desacoplados, cujas dinâmicas tenham diferentes escalas temporais (BUZDUNGAN, et al., 1999). Nos subsistemas de dinâmica lenta é usado um menor número de passos de integração, do que nos subsistemas de dinâmicas rápidas.

Algumas das ultimas abordagens e técnicas para simulação com hardware in the loop podem ser encontradas em sites de companhias que produzem produtos ligados a sistemas de controle, como a Mathworks (MATHWORKS, 2007), dSpace (DSPACE, 2007) e National Instruments (NATIONAL, 2007), por exemplo.

4. METODOLOGIA

4.1 Simulação com Hardware in the Loop

O processo de definição de um sistema de controle e simulação com hardware in the loop (*HIL*) apresenta desafios adicionais quando comparado ao processo de simulação simples ou de uma futura implementação. Por exemplo, o controlador deve ser executado em algum tipo de computador de tempo real de forma a garantir o escalonamento de entrada e saída (*I/O - Input ou Output*), obter as derivadas em intervalo de tempo fixo e processar todos os algoriítmos de controle dentro do laço controle principal em uma determinada taxa amostragem. Requisitos de hardware como o poder de processamento e interfaceamento com circuitos que disponibilizam sinais analógicos ou digitais devem ser levados em consideração. Além disso, requisitos de software e questões relacionadas à programação devem ser discutidas, necessariamente, antes do design, pois em muitos casos, o software a ser desenvolvido está fortemente acoplado (e em alguns casos, dependente) ao hardware.

4.1.1 Considerações de Hardware

A estrutura de um simulador *HIL* tipicamente consiste dos seguintes componentes de hardware:

- **Computador para desenvolvimento do software**. Computador que comporta softwares como editores de texto, compiladores, depuradores de código e, ocasionalmente, simuladores. Como por exemplo: Tornado C/C++ IDE (*Integrated Development Enviroment*) (TORNADO, 2005), Constellation Real Time Framework (CONSTELLATION, 2005), MatLab (MATLAB, 2000) ou Simulink (SIMULINK, 2000);

- **Planta**. É o sistema a ser controlado;

- **Computador de Execução do Software**. Computador sobre o qual será executado o código gerado pelo computador de desenvolvimento e interage diretamente com a planta. Ele pode ser tão complexo quanto um sistema dedicado de múltiplos processadores ou tão simples quanto um

DSP (*Digital Signal Processor*) embarcado ou microcontrolador, como por exemplo: um PIC (PIC, 2001), Basic Stamp (STAMP, 2004) ou qualquer outro baseado na especificação ARM (SEAL, 2005);

- **Circuitos Condicionadores de Sinal**. Consistem de filtros, amplificadores atenuadores ou diferentes formas de sinais, localizados entre a planta e o computador que comporta o software de execução. Composto, basicamente, de componentes discretos (resistores, capacitores, op-amps, etc).

No projeto de um sistema de controle e simulação com *HIL*, o computador para desenvolvimento de software executa editores de texto, compiladores e até mesmo geradores de código automático, como é o caso do Mathwork's Real Time Workshop (RTW, 2000). Desde que o desenvolvimento seja conduzido em *off-line*, ou seja, o computador de desenvolvimento e computador de execução estão fisicamente separados, requisitos mínimos são impostos a máquina de execução e o código pode ser compilado em um período razoável de tempo. No caso em que o computador de desenvolvimento de software também é utilizado para a execução, significantes requisitos são colocados sobre o processador devido ao gerenciamento de entradas e saídas de tempo real e a demanda, em geral, de interfaces de usuário.

A planta deve ser interfaceada pelo computador de execução do software, exigindo em muitos casos a conversão de sinais analógicos para digitais e vice-versa. Diversas placas para o padrão PC estão disponíveis que apresentam características digitais de I/O, A/D (Analógico/Digital), D/A (Digital/Analógico), temporizadores e contadores. O ambiente de desenvolvimento de software com suporte a estas placas de entrada e saída reduz o tempo de desenvolvimento e permite ao desenvolvedor focar no desempenho do sistema de controle. Além disso, pode ser vantajoso implementar o software sobre microcontroladores embarcados. Estes chips oferecem soluções completas em um único circuito integrado e podem conter conversores A/D “on-board”, capacidade de processamento digital de sinais, interrupções de hardware e saídas com modulação em largura de pulso (*PWM – Pulse Width Modulation*). Produtos de empresas como a Texas Instruments, National

Semiconductors e Microchip Technology podem ser programados em linguagem de programação de “alto nível” (como C, por exemplo) e executar código de tempo real em pequenos pacotes.

Uma vez que uma interface de hardware viável foi selecionada, um circuito condicionador de sinais passa a ser necessário. Filtros analógicos são, preferencialmente, construídos em hardware de forma a aliviar a carga computacional sobre o DSP e a CPU (*Central Processing Unit*). Entretanto o hardware condicionador de sinais passa a ser uma solução menos flexível do que uma solução em software devido à dependência dos componentes discretos. No mínimo, um filtro passa-baixa sobre os sensores analógicos de uma planta limitarão os ruídos de alta freqüência associados com as conversões A/D e interferências eletromagnéticas induzidas por meio de cabos ligados aos sensores.

4.1.2 Considerações de Software

O desenvolver do software do sistema de controle freqüentemente corresponde a mais complexa e demorada fase do projeto. Precauções podem ser tomadas durante a seleção de efetivos sistemas operacionais e ferramentas de desenvolvimento, como por exemplo, se alocação de recursos complexos do sistema podem ser monitorados automaticamente ou mesmo a possibilidade de manipulação de características intrínsecas associadas com a escrita de código de tempo real.

Os sistemas operacionais sobre os quais irá executar o software embarcado estão divididos em dois tipos: Sistemas operacionais de tempo real e sistemas operacionais orientados a eventos. Sistemas operacionais de tempo real como VxWorks® (VXWORKS, 2005), QNX® (QNX, 1996), LabView Real Time® (LRT, 2001) e xPC® (XPC, 2002) executam código em intervalos de tempos específicos. Os recursos computacionais necessários para executar uma seção particular de código são arranjados pelo sistema operacional de modo que a execução daquele código tenha sido completada antes que a próxima seção seja escalonada ara execução. Desse modo, garantias podem ser feitas em sistemas operacionais de tempo real como quão rápido o código pode ser executado ou com que prioridade cada seção de código tem sobre a outra. Por exemplo, computadores que controlam processos industriais tal como estações de força nuclear que

requerem que as válvulas do sistema de resfriamento sejam abertas ou fechadas em períodos fixos. Se uma destas válvulas não foi aberta no período de tempo especificado porque o sistema operacional escalonou e executou outra tarefa menos crítica, podem ocorrer implicações sérias relacionadas à segurança da planta.

Sistemas operacionais orientados a eventos como, por exemplo, Microsoft Windows® 95/98/NT/2000/XP, Linux (BOVET, et al., 2002) e BSD (FREEBSD, 1999) são direcionados a responder apenas a entradas externas como aquelas iniciadas por um usuário (toque do teclado, clique do mouse, etc). Entretanto, é possível obter um pseudo comportamento tempo real por meio da utilização temporizadores e interrupções para forçar o sistema a executar um trecho de código a taxas fixas de tempo. Porém, os resultados em boa parte dos casos ainda são imprevisíveis, devido ao fato que os serviços associados às interrupções não podem garantir que aquele código seja completado durante o tempo especificado. De fato, estes sistemas operacionais são tipicamente gráficos em natureza e freqüentemente multitarefa, implicando que múltiplas aplicações ou tarefas gráficas possam demandar tempo do processador durante pontos críticos na operação da planta.

4.2 Sistemas Operacionais de Tempo Real

Controladores desenvolvidos com métodos gráficos como aqueles usados com o Simulink ou MatrixX (MATRIxX, 2003) não são facilmente portados para sistemas operacionais orientados a eventos, e controladores de tempo real e escalonamento são difíceis para programar. Para fazer isto, projetistas devem estar aptos à codificação de software para executar sobre o hardware, exigindo sincronização de entrada e saída e ao mesmo tempo resolvendo equações diferenciais em tempo real. Segundo (CALVO, 2002), uma coisa é entender como um solucionador de equações diferenciais pelo método de Runge-Kutta opera, outra é implementá-lo para execução em tempo real. Desse modo, projetistas podem facilmente gastar várias horas codificando e perder importantes conceitos do projeto do controlador e sua operação em tempo real.

Muitos pacotes de software têm tentado gerar automaticamente o código de controle para execução sobre o hardware, porém estes pacotes possuem significantes limitações. Por exemplo, o MatLab vem divulgando o RTW (*Real Time*

Workshop) já a alguns anos. Este software trabalha com blocos de controle gráficos criados no Simulink e compila-os para a execução sobre vários ambientes como o DOS (*Disk Operating System*) em modo protegido ou VxWorks. Infelizmente, o RTW inclui apenas alguns poucos módulos de interfaces (*drivers*) - quando comparado ao montante de dispositivos e padrões que existem no mercado - exigindo que o usuário desenvolva a maioria deles. Enquanto esta solução alivia a carga para o solucionador (*solver*) de equações diferenciais, ela não permite uma fácil mobilidade, por parte dos usuários, entre a simulação do controlador para o hardware atual. Além disso, o RTW confia ao sistema operacional da máquina *target* (o mesmo que computador de execução de software) para a alocação de recursos de tempo real e entrada e saída do sistema, promovendo flexibilidade, porém fazendo múltiplas concessões para permitir a compatibilidade com múltiplos sistemas operacionais.

Em geral, uma plataforma de desenvolvimento de código deve fornecer um conjunto básico de características para o usuário. Uma interface gráfica intuitiva é requerida para apresentar claramente os fluxos dos sinais, ferramentas de análise, suporte a simulação (simular a resposta de um modelo passo a passo, plotagem das freqüências de resposta, etc) e, talvez, mais importante, fornecer um link direto com o sistema operacional (seja ele DOS, VxWorks, QNX ou qualquer outro) no qual o gerenciamento direto de recursos possa ser obtido. Felizmente, O pacote de software *Constellation*, da RTI (*Real Time Innvations*), provê, além de ferramentas de integração, um framework baseado em componentes que permitem descrever e implementar processos complexos, diminuindo as dificuldades existentes entre algoritmos de controle, sistema operacional e hardware; além de cobrir as deficiências apresentadas pelo RTW no que diz respeito a sua conexão e geração de código para o sistema operacional de tempo real VxWorks.

Toma-se, então, como estratégia a utilização de um modelo de controle para veículos submarinos já implementado, testado e disponibilizado pelos autores (BRAUNL, et al., 2004), que permita avaliar os resultados com os objetivos do trabalho.

5. MATERIAIS E MÉTODOS

5.1 Plataforma de Desenvolvimento

A plataforma de desenvolvimento consiste, especificamente, da montagem e utilização de hardware com características semelhantes às apresentadas pelo projeto do veículo submarino que se encontra em desenvolvimento nos laboratórios da Engenharia Mecatrônica da Escola Politécnica de São Paulo (LIMA, et al., 2003), (LIMA, 2003a), (ALMEIDA, et al., 1999), permitindo o aproveitamento de recursos e soluções em pesquisa com temas relacionados.

Para a montagem do sistema embarcado, utiliza-se uma placa padrão PC-104 (PC-104, 2003), apresentada na Figura 4 e com as seguintes especificações:

- Processador Geode GX1 – com clock de 300 MHz;
- Memória do sistema: SDRAM SODIMM com 128 MB;
- Solid Disk State (SSD) CompactFlash;
- Watchdog Timer;
- Bateria de Lítio;
- 15 níveis de Interrupção;
- Suprimento de força de +5V (4.75V a 5.25V);
- Peso bruto de 0.25 Kg;
- Barramento padrão ISA.
- Temperatura de Operação variando de 0 a 60°C;
- Placa de comunicação de rede padrão ethernet: 10/100Base-T;
- Porta Serial (padrão RS-232) e Paralela;

Figura 4. Placa padrão PC-104 com processador Geode 300 MHz.

Através da porta de comunicação serial RS-232 disponibilizada pela placa, realiza-se a comunicação com uma bússola digital programável de três eixos (OS5000, 2008) responsável pela indicação das posições referentes aos movimentos de roll, pitch e heading, avaliados pela simulação com hardware in the loop. Esse sensor foi utilizado como caso exemplo para interfacear o software do controlador embarcado com um sensor real, e testar os módulos que permitem selecionar a fonte dos sinais de entrada usados em uma simulação – sinais armazenados em arquivos ou sinais gerados por sensores reais. Essa funcionalidade do ambiente de simulação desenvolvido está descrita na seção 5.8.

Figura 5. Bússola digital magnética com 3 eixos.

A Figura 5 apresenta a bússola magnética que possui como características principais:

- Temperatura de Operação: -20 a 70°C;
- Precisão: 0,1 graus;
- Dimensões: 25,4 mm X 25,4 mm
- Frequência de amostragem: de 0,01 Hz a 40 Hz;
- Suprimento de força:
  - Tensão: 3,3 a 5 Volts;
  - Corrente: 20 mA.

5.2 Ambiente Operacional de Tempo Real

VxWorks é um sistema operacional de tempo real fabricado pela Windriver Systems. Como a maioria dos sistemas operacionais de tempo real (VXWORKS, 2005). Ele inclui um kernel com suporte a multi-tarefas e escalonamento preemptivo. Muito popular em aplicações embarcadas é utilizado em varias arquiteturas de hardware, incluindo MIPS, PowerPC, SH-4, x86, ARM, StrongARM, xScale.

5.3 Framework de Desenvolvimento em Sistemas de Tempo Real

O Constellation, consiste de um framework de tempo real orientado a objetos (CONSTELLATION, 2005), com capacidade de interfaceamento e geração de código partindo de um modelo projetado em MATLAB/Simulink até a obtenção de código em linguagem de programação nos padrões ANSI C++. Favorece a componentização de software, método no qual é elevada a taxa de reutilização e especialização de código, principalmente, orientado a objetos.

5.3.1 Conversão do Modelo Matlab/Simulink em Componentes de Software

É possível, dado um modelo MATLAB/Simulink, convertê-lo num único componente, sendo ele, no ambiente Constellation, representado graficamente por um único bloco composto de entradas e saídas, como mostra a Figura 6.

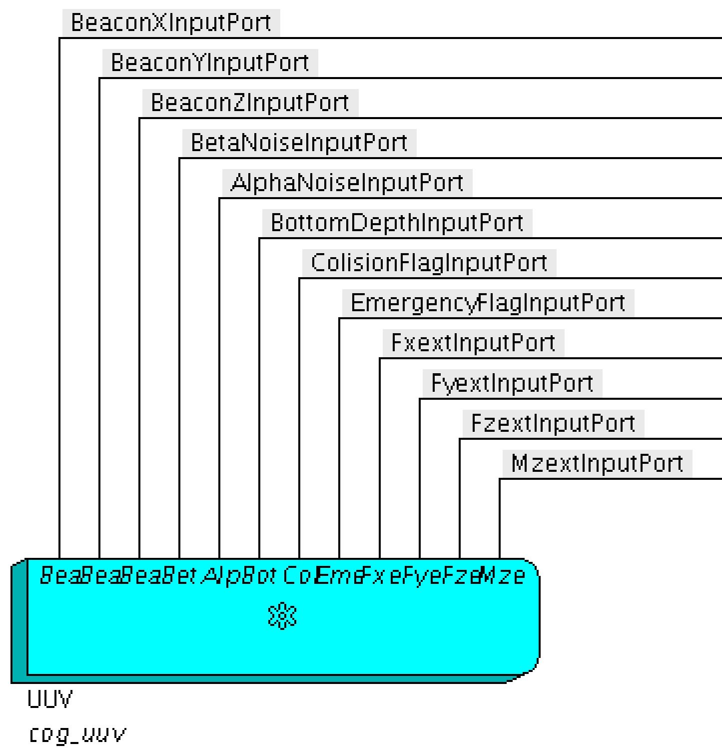

Figura 6. Bloco correspondente ao modelo simulink em ambiente RTI's Constellation.

Esta abordagem, apesar de ser bem mais rápida, torna o sistema de difícil desenvolvimento e depuração. Desse modo, o processo de conversão foi direcionado em converter cada bloco Simulink num respectivo componente de software orientado a objetos do ambiente Constellation. As figuras 7 e 8 apresentam, respectivamente, diagramas de blocos em ambiente Matlab/Simulink e seu equivalente em termos de componentes em ambiente Constelation do sistema de coordenadas e planta de um UUV, por exemplo.

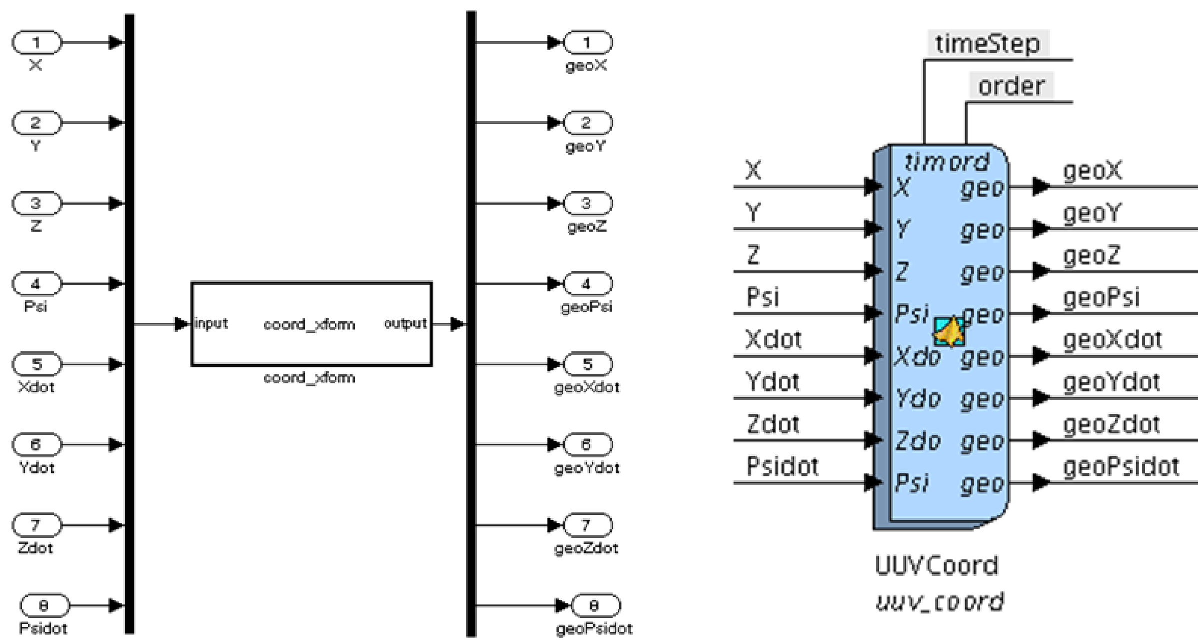

Figura 7. Sistema de coordenadas de um UUV – Blocos Simulink (à esquerda) e componente Constellation (à direita).

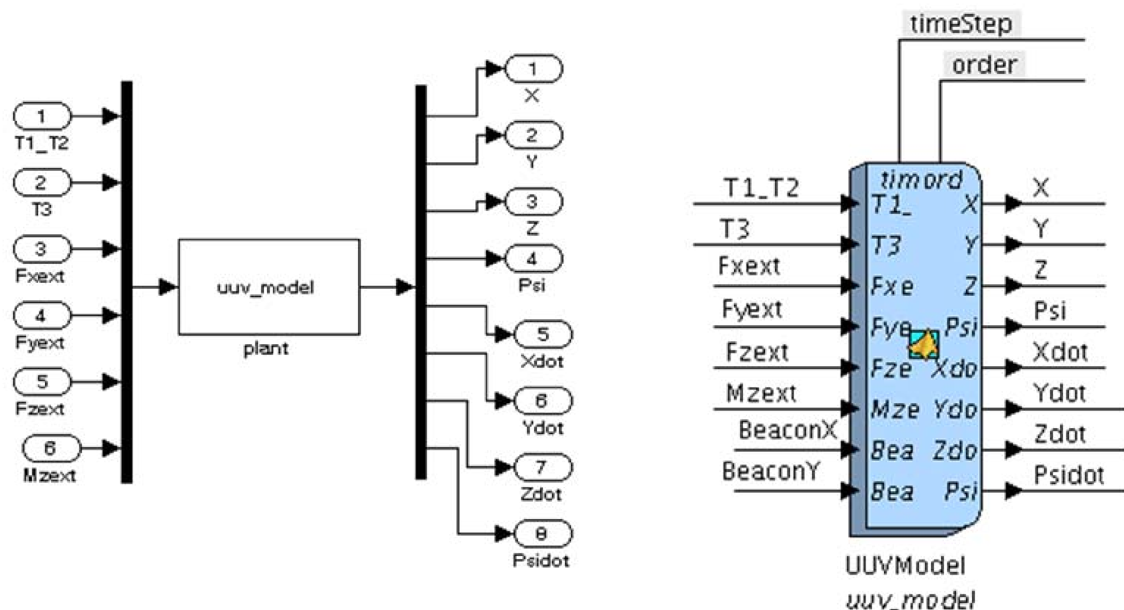

Figura 8. Modelo dinâmico ou planta de um UUV – Blocos Simulink (à esquerda) e componente Constelation (à direita).

Para cada bloco Simulink são então gerados e obtidos seus respectivos componentes. Esta abordagem mesmo não sendo obrigatória, acelera o desenvolvimento e aumenta o reaproveitamento do código dos componentes.

Após o processo de componentização, passa a ser necessário o mapeamento de entradas e saídas dos componentes existentes, de forma a obedecer o mesmo fluxo apresentado pelo digrama MATLAB/Simulink. É possível adotar uma série de estratégias de interfaceamento entre componentes (utilização de serviços publicados, execução direta de métodos ou funções associadas a cada um dos componente ou acesso direto às variáveis e/ou atributos de cada um). O acesso direto aos atributos, por oferecer maior desempenho e facilitar o entendimento, constitui-se num padrão para as interações entre os componentes. No ambiente de programação MATLAB/Simulink, tais atributos estão organizados em memória na forma de matrizes.

Por fim, com todos os blocos MATLAB/Simulink mapeados em seus respectivos componentes, é utilizado um gerador de código que consiste basicamente de um módulo de software que, entre outras funções, realiza comunicação com o RTW nativo do MATLAB, interpreta classes, métodos e atributos contidos na descrição de cada componente do ambiente Constellation e constrói o código para o ambiente de tempo real, que será compilado pelo VxWorks para a arquitetura do computador de controle.

5.4 Modelagem e Simulação

A Figura 9 apresenta o digrama de simulação desenvolvido em MatLab/Simulink criado para modelar a dinâmica do UUV, refinar o projeto dos controladores e examinar a viabilidade do projeto através da execução de testes de missões. O modelo utilizado é baseado no descrito em (BRAUNL, et al., 2004) para o UUV Hornet.

O Hornet foi utilizado por se tratar de um UUV com modelo de controle adequado aos objetivos deste trabalho, oferecendo, portanto, um modelo de controle de média complexidade (o que se encaixa no cronograma de atividades pretendidas), de fácil entendimento e inteiramente disponível pelos autores.

Figura 9. Diagrama de blocos do MatLab/Simulink para a simulação do UUV.

Figura 10. O UUV Hornet consiste de dois propulsores horizontais, um propulsor vertical, uma câmera CCD e uma lanterna submersível de halogênio montados em uma estrutura tubular de PVC

O Hornet, Figura 10, consiste de dois propulsores horizontais, um propulsor vertical, uma câamera colorida CCD e uma lanterna para mergulho de halogênio. O sinal de vdeo e sinais de controle de voltagem para os dois propulsores sem-escovas DC são alimentados por meio do cabo umbilical através de um par trançado de fios de cobre blindados. Possui um suprimento de força de 110 volts enviados pelo sistema inversor de sua base de operação na superfície, trabalhando com 24 volts e cerca de 10 ampères.

.

5.5 Dinâmica do UUV

Um dos primeiros passos no desenvolvimento de uma simulação apurada é a modelagem das equações dinâmicas do UUV. Neste caso, as coordenadas referentes ao corpo rígido são descritas pelos seguintes 6 (seis) graus de liberdade e suas respectivas derivadas, conforme mostra a Figura 11:

x(*surging*) - Movimento linear com relação ao eixo longitudinal;

y(swaying) - Movimento linear com relação ao eixo transversal;

z(*heaving*) - Movimento linear com relação ao eixo vertical;

ϕ(*roll*) - Movimento rotacional sobre o eixo X.

θ(*pitch*) - Movimento rotacional sobre o eixo Y.

ψ(*yaw*) - Movimento rotacional sobre o eixo Z.

As entradas do sistema são definidas da seguinte forma:

$\mathbf{T}_1$- Força propulsora aplicada pelo propulsor lateral;

$\mathbf{T}_2$- Força propulsora aplicada pelo propulsor frontal;

$\mathbf{T}_3$- Força propulsora aplicada pelo propulsor vertical;

$\mathbf{F}_{ext(x,y,z,y)}$ – Distúrbios externos (correnteza da água, etc..);

$\mathbf{F}_{d(x,y,z)}$ – Forças de arrasto hidrodinâmico linear devido ao movimento do veículo definido pela equação 4 (ROBERSON, et al., 1997):

$$\mathbf{F}_d = \mathrm{C}_d * \rho * \mathrm{A}_p * \frac{\mathbf{V}_0^2}{2} \tag{4}$$

Onde:

$C_d = Coeficiente\ de\ arrasto$

$\rho = densidade\ da\ água$

$A_p = Área\ de\ arrasto\ projetada$

$\boldsymbol{V_0} = Velocidade\ da\ superfície\ de\ arrasto$

Outros parâmetros importantes:

$m - massa\ do\ veículo = 90\ kg$

$D - Separação\ entre\ os\ propulsores\ \boldsymbol{T}_1\ e\ \boldsymbol{T_2} = 0{,}28\ m$

$\boldsymbol{W}_{auv} -\ Peso\ do\ veículo\ na\ água\ (considerando\ empuxo\ 0)$

I_z - Momento de inércia sobre o eixo-z = $0{,}004\ kg * m^2$

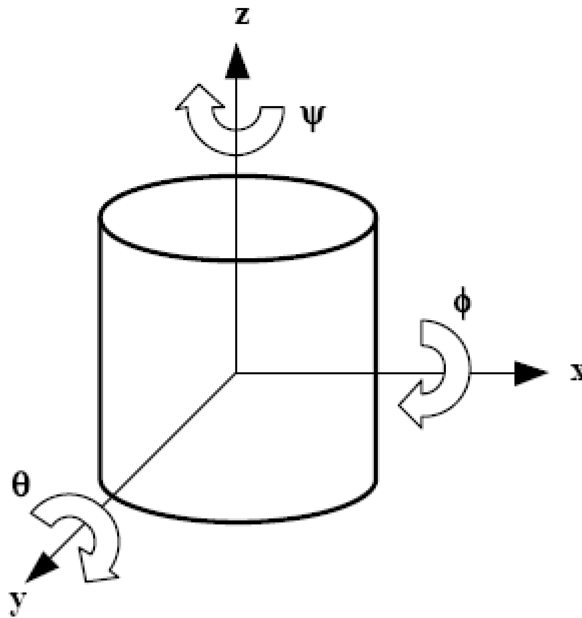

Figura 11. Demonstração de corpo rígido com sistema de coordenadas.

Antes de iniciar a derivação das equações de movimento, é importante notar que $y,\ \phi\ e\ \theta$ são todos graus de liberdade que não serão controlados.

Os ângulos $\phi\ e\ \theta$ não são de interesse para o controlador, devido a configuração dos propulsores e da quilha utilizada. Isto na realidade é uma das vantagens primárias do modelo, na qual *roll* e *pitch* podem ser removidos da equação de movimento. Isto porque se assume que $\mathbf{T_1}, \mathbf{T_2}\ e\ \mathbf{T_3}$ agem no centróide do veículo. De fato, este não é o caso. Entretanto, devido ao elevado peso da quilha, acoplada com um largo módulo de empuxo e uma larga separação entre o centro de empuxo de veículo $\mathbf{C_b}$ e o centro de gravidade $\mathbf{C_g}$, é possível assumir que o veículo possuirá um elevado momento $\mathbf{GZ}$, permitindo que permaneça suficientemente estável em *roll* e *pitch*, da mesma forma que incorre sobre os distúrbios nominais em outros graus de liberdade do veículo. Portanto, a única preocupação são os 4 (quatro) graus de liberdade requeridos para manter a posição do veículo no espaço: x, y, z, ψ, três dos quais são controlados: x, z, ψ.

As equações dinâmicas de movimento podem ser derivadas da seguinte maneira (OLGAC, et al., 1991).

Tomando a soma de todas as forças nas direções dos eixos X, Y e Z e resolvendo as respectivas acelerações apresentadas nas equações 5, 6 e 7, respectivamente:

$$\ddot{x} = \frac{(\mathbf{T}_1 + \mathbf{T}_2) + \mathbf{F}_{dx} + \mathbf{F}_{xext}}{m} + \dot{y}\,\dot{\psi}\,, \tag{5}$$

$$\ddot{y} = \frac{\mathbf{F}_{dy} + \mathbf{F}_{yext}}{m} + \dot{x}\,\dot{\psi}, \tag{6}$$

$$\ddot{z} = \frac{\mathbf{T_3} + \mathbf{T}dz_2 - \mathbf{W}_{auv} + \mathbf{F}_{zext}}{m}, \tag{7}$$

onde:

$$\mathbf{F}_{d(x,y,z)} \; são \; definidos \; pela \; equação \; 4.$$

$$\mathbf{F}_{xext} = \mathbf{F}_{yext} = \mathbf{F}_{zext} = 0 (assumindo \; nenhuma \; correnteza \; externa \; ou \; distúrbio).$$

Tomando a soma dos momentos sobre o eixo Z, temos:

$$\ddot{\psi} = \frac{(\mathbf{T_2} - \mathbf{T}_1)\left(\frac{D}{2}\right) + \mathbf{M}_{zext}}{I_z}, \tag{8}$$

onde:

$$\mathbf{M}_{zext} = 0 \; (assumindo \; a \; ausência \; de \; distúrbios \; sobre \; o \; eixo \; Z).$$

Para os propósitos da simulação, estas equações de movimentos comportam 8 (oito) variáveis de estado e 3 (três) entradas de sistemas independentemente controladas, apresentadas na equação 9, a seguir.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \psi \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\psi} \end{bmatrix}, \qquad u = \begin{bmatrix} \mathbf{F_x} \\ \mathbf{M_z} \\ \mathbf{F_z} \end{bmatrix} = \begin{bmatrix} \mathbf{T_1} + \mathbf{T_2} \\ (\mathbf{T_2} - \mathbf{T_1})\left(\frac{D}{2}\right) \\ \mathbf{T_3} \end{bmatrix}, \tag{9}$$

onde u é em função de $\mathbf{T}_1$, $\mathbf{T}_2$ e $\mathbf{T}_3$ (as entradas para o modelo da planta dos três controladores de propulsão).

No modelo auv_plant (ver Figura 9), estas equações de movimento foram implementadas em um bloco *S-Function* do MATLAB.

5.6 Sistema de Navegação

Dois blocos no modelo Simulink apresentado compreendem o sistema de navegação. O bloco chamado “coord_xform” transforma os estados de saída do corpo rígido do modelo da planta para coordenadas com relação a uma referência geográfica. No plano horizontal, a seguinte matriz de transformação de transformação é utilizada.

$$\mathbf{X_{BG}} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix}, \tag{10}$$

onde, no plano vertical, as coordenadas de referencia podem ser diretamente superpostas uma pela outra.

O bloco “nav_system” do diagrama (ver Figura 9) simula o sistema de navegação por meio da solução do posicionamento do veiculo com respeito a uma unidade acústica submersa, um *beacon,* configurada pelo usuário. Ruídos durante a navegação podem ser adicionados para simular erros de cálculo. Deste bloco, os estados apresentados na equação 11 são saídas para o planejador de missões e módulos de controle.

$$\mathbf{X}_{nav} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} \alpha \\ \psi \\ \mathrm{R} \\ \mathrm{Z} \\ \mathrm{H} \end{bmatrix}. \tag{11}$$

5.7 Controladores de baixo nível

O conjunto de controle de baixo nível consiste, para este modelo, de três controladores: profundidade, velocidade e direção. Cada um deles está dedicado a receber comandos do planejador da missão e gerar saídas dos propulsores com base nos erros entre a profundidade e direção atuais e comandadas.

O mais simples dos três controladores, o controlador de profundidade, recebe comandos de profundidade do planejador da missão, a profundidade atual do veículo (z) do sistema de navegação e provoca uma saída no propulsor ($\mathbf{T}_3$). Através da equação 7 é possível perceber que o relacionamento entre a profundidade e o propulsor $\mathbf{T}_3$ é simples e linear. Portanto, um controlador PID (Proporcional-Integral-

Derivativo) é mais que suficiente (OGATA, 1997) e seu diagrama de blocos correspondente apresentado na Figura 12.

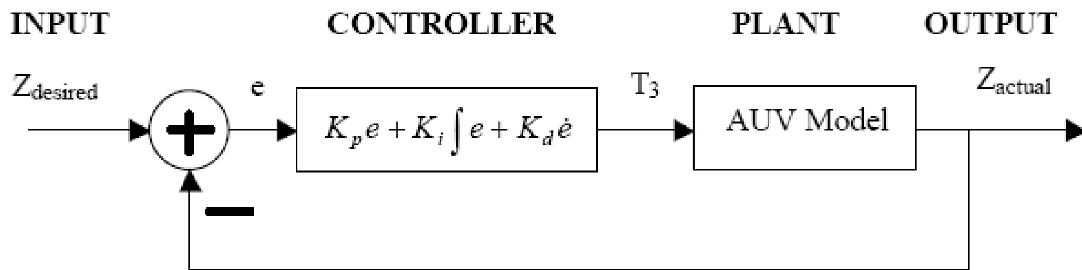

Figura 12. Diagrama de blocos do controlador de profundidade PID.

Onde:

$Z_{desired}$ = Comando de profundidade de planejador da missão

Z_{actual} = Profundidade atual obtida do sistema de navegação

e = Erro de profundidade

T_3 = Saída do propulsor vertical

K_p = Ganho proporcional

K_i = Ganho integral

K_d = Ganho derivativo

Os controladores de velocidade e direção são um pouco mais complexos devido a sua natureza não-linear. Como resultado, as duas tarefas devem ser combinadas dentro de um único controlador híbrido, cuja função é tanto de um controlador de correção de direção (quando o erro de direção está presente), quanto de um controlador PID de velocidade (quando o veículo está em curso). Os projetistas do Hornet optaram por realizar a correção da direção através de um controlador do tipo *Slide-Mode*, isto é, *Modo-Deslizante* (DECARLO, et al., 1996), que é ativado se o erro de direção excede os limites estabelecidos da camada de deslizamento (no caso, +/- 3 graus). Um controlador de modo deslizante trabalha tentando deixar o erro (e) e a taxa de erro ($\dot{e}$) em zero, ao longo de uma função de deslizamento definida pela equação 12,

$$\mathbf{s} = \mathbf{h}^T e, \tag{12}$$

onde:

h = um vetor bi-dimensional com os ganhos do controlador (Muda a inclinação de deslizamento da reta)

e = um vetor bi-dimensional contendo o erro e a taxa de erros

Como pode ser visto na Figura 13, quando (e) e ($\dot{e}$) saem fora da linha de deslizamento, o controlador move de volta as linhas para fazer com que o sinal de controle seja o máximo positivo ("*reaching*"). Isto permite ao controlador compensar situações desconhecidas geradas por sistemas não-lineares, forçando o sistema a permanecer dentro do domínio onde ele se torna linear e pode, então, iniciar o deslizamento em direção a origem (ao longo da linha de deslizamento). Isto é especialmente útil se pouco ou nada é conhecido sobre os aspectos não-lineares do modelo do sistema. Uma vez sobre a linha, o controlador continua a reduzir (e) e ($\dot{e}$) comportando-se como um controlador linear PD (BRAUNL, et al., 2004).

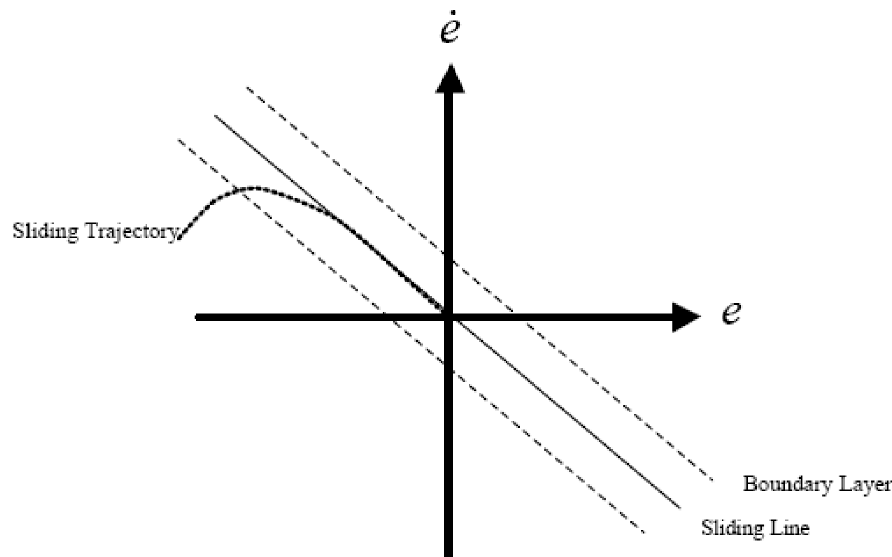

Figura 13. Comportamento do controlador de modo deslizante no plano de erro.

Adicionalmente, como mostra a Figura 13, uma camada limitante pode ser adicionada para reduzir a vibração, deixando "mais densa" a linha de deslizamento. No caso do controlador híbrido, o controlador de direção em modo deslizante desacopla dentro do intervalo de +/- 3 graus na camada limitadora, permitindo que o controlador de velocidade PID mova o veículo para frente, com base nos comandos

do planejador da missão. Desse modo, o controle de direção é apenas iniciado apenas quando o veículo está fora do curso. Estes dois controladores são apresentados nas figuras 9 e 10, respectivamente.

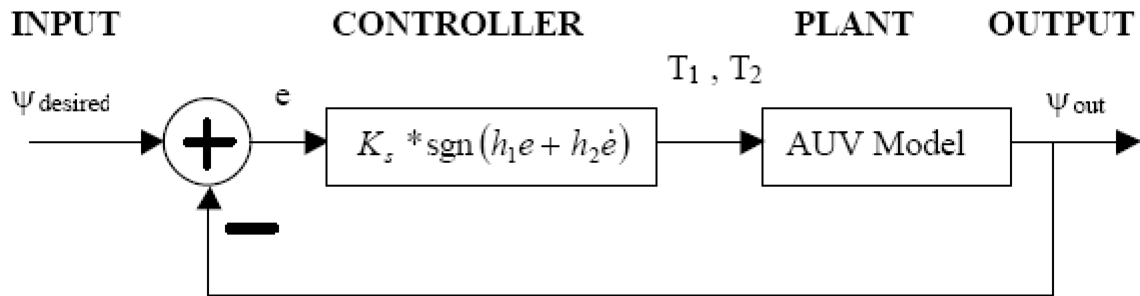

Figura 14. Diagrama de blocos do controlador de direção em modo deslizante.

Na Figura 14, tem-se:

$\psi_{desired}$ = Comando de direção do planejador da missão

K_s = +/- Limites sobre a saída do controlador (limites do propulsor)

h_1= Ganho de erro

h_2= Ganho na taxa de erro

$\mathbf{T_1}$ e $\mathbf{T_2}$= Saída dos propulsores horizontais

Note que controlador em modo deslizante apresentado na Figura 15 não contém estimativas das não linearidades do sistema e, portanto comporta-se como um controlador *bang-bang*, que tenta atualizar os valores do erro e da taxa de erro para zero (OLGAC, et al., 1991).

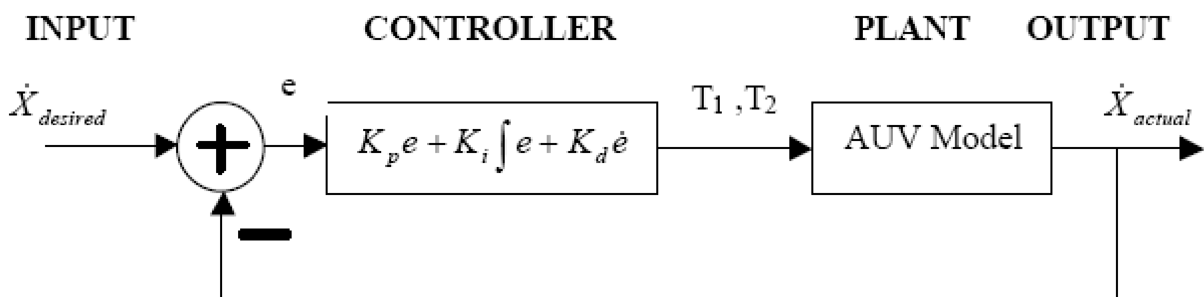

Figura 15. Diagrama de blocos de um controlador de velocidade PID.

Na Figura 15, tem-se:

$\dot{\mathbf{X}}_{desired}$ = Velocidade desejada enviada pelo planejador da missão.

e = Erro de velocidade.

$\mathbf{T}_1$ e $\mathbf{T}_2$= Saída dos propulsores horizontais

K_p = Ganho proporcional

K_i = Ganho integral

K_d = Ganho derivativo

É importante observar que, desde que a determinação da velocidade do veículo é baseada na taxa de mudança em *range* (R) até a unidade acústica submersa, a velocidade do veículo nem sempre é precisa. Neste caso, utiliza-se a informação da velocidade do propulsor, ao invés da velocidade desejada.

5.8 Implementando um ambiente de Simulação com Hardware in the loop

A Figura 16 apresenta uma visão geral da arquitetura de hardware in the loop utilizada para auxiliar no desenvolvimento e construção do veiculo submarino autônomo.

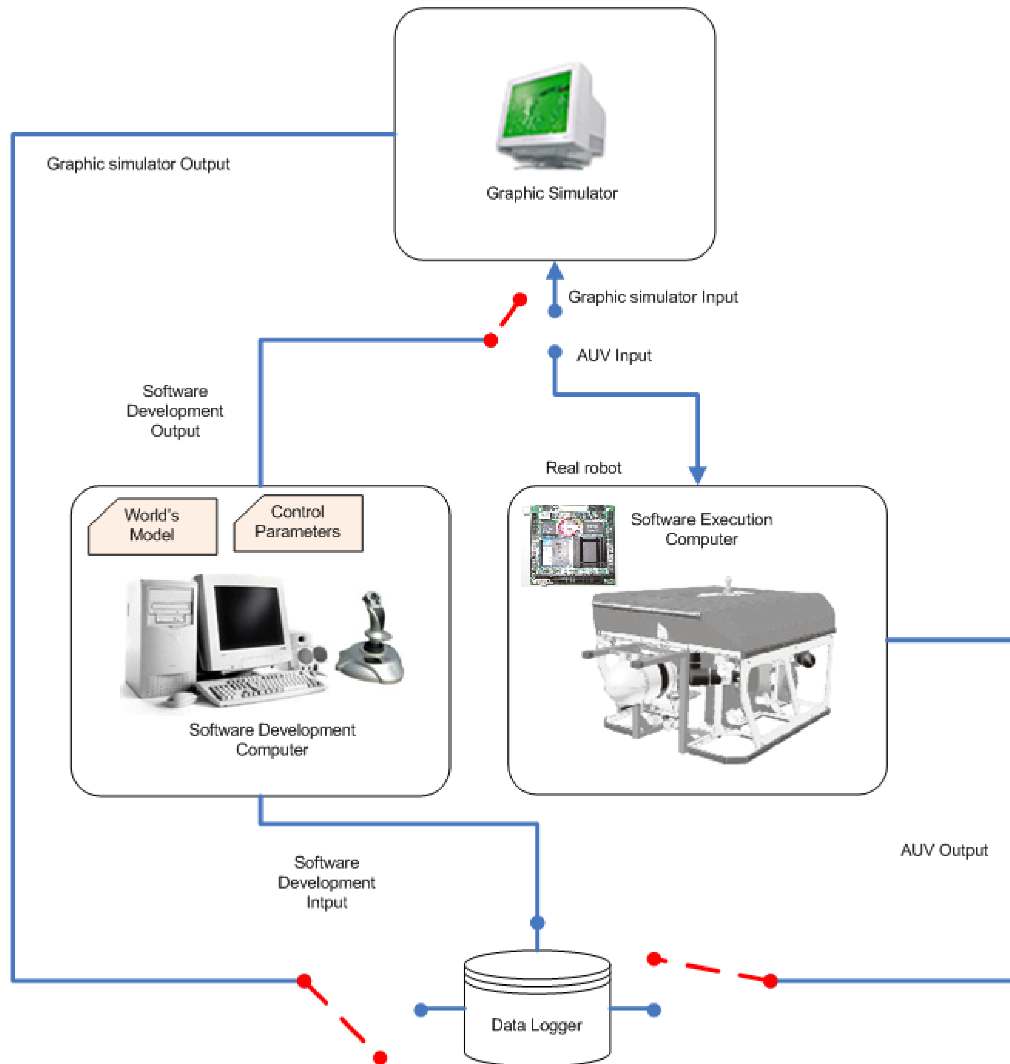

Figura 16. Visão geral da arquitetura de simulação com hardware in the loop aplicada ao desenvolvimento e construção de Veiculo Submarino Autônomo.

A seguir, os componentes que compõem a arquitetura são brevemente descritos:

- **Real Robot**. Representa o Veículo Submarino Autônomo real, consistindo de componentes de software e hardware. Suas entradas consistem das forças vetoriais geradas pelo meio (correntes, empuxo, etc) e o vetor de torque aplicado pelos propulsores e planos de controle;

- **World Model**. Consiste do modelo do mundo físico no qual será inserido veículo. O mundo subaquático é composto pela topografia das profundezas do oceano, a presença de agentes marinhos (peixes ou mesmo outros robôs) e objetos genéricos como: estruturas off-shore, cabos, caixas, etc. O

modelo do mundo possui também a representação de diferentes fenômenos físicos como correntezas e ondas, por exemplo.

- **Parâmetros de controle**. Variáveis principais e auxiliares usadas para a realização adequada dos algoritmos de controle empregados no veículo. Parâmetros como velocidade, aceleração, gravidade, forca empuxo, sentido e direção da correnteza, por exemplo.
- **Data Logger**. Registro das estruturas de dados obtidos durante testes e operação, sejam eles total ou parcialmente simulados. Através deste componente é possível rever dados da operação, validar e verificar a implementação dos algoriítmos de controle.

Um bom montante do trabalho se dá no processo de conversão entre os códigos gerados pelo Simulink (com o auxílio do RTW), *Constellation* framework e interfaces C/C++ para código legado em VxWorks. Em alguns casos, é necessário prover uma estrutura transparente de comunicação que permita o acesso de funções ou toolbox do MATLAB ao ambiente montado sobre o sistema operacional VxWorks, possibilitando a alteração de valores de alguns estados em tempo de execução diretamente daquele ambiente. Para isso foram utilizados alguns padrões de projetos como: *Interfaces*, *Adaptors, Facades, Fatories e Proxies*; possibilitando diferentes formas de acesso e flexibilidade ao framework construído (GAMMA, et al., 1995).

Os passos a seguir são necessários para gerar código compatível com a arquitetura de hardware in the loop proposta, baseado, é claro, inicialmente no desenvolvimento e análise do modelo conceitual, lógico e físico dos problemas associados à construção de um veiculo submarino autônomo.

- Preparar o modelo do controlador em Simulink para o UUV;
- Converter aquele modelo em um componente de software compatível com o modelo e arquitetura do *Constellation*;
- Preparar o ambiente target com sistema operacional de tempo real VxWorks. A comunicação pode ser feitas de inúmeras formas, porém, a mais usual e rápida se dá por meio da criação e compartilhamento de um sistema de arquivo de rede (*Network File System – NFS*), permitindo acesso direto a diretório e arquivos com os quais seja necessário trabalhar;

- Configurar o *Constellation* para gerar código correspondente para a máquina target (VxWorks) ou para ambiente de simulação (código em C/C++ sem considerar as restrições de tempo gerenciadas pelo S.O. de tempo real);
- Durante o processo de conexão, comunicação e execução do software de controle na máquina *target* (*UUV*) a engine do MatLab será chamada sendo possível ainda, caso necessário, acompanhar, gravar logs ou mesmo alterar valores de variáveis durante o processo, como mostra a Figura 16.
- Outro recurso ainda utilizado é a integração com a ferramenta Stethoscope® da WindRiver (SCOPETOOLS, 2005), que fornece gráficos de acompanhamento dos valores de variáveis que deseja-se monitorar.
- Uma importante característica do *Constellation* é a visualização gráfica dos estados criados pelo software de controle na máquina target em tempo de execução, isso favorece a tomada de decisão e a correção de erros mais rapidamente.
- O software WindView tem grande importância no ambiente, pois através dele é possível certificar que o algoriítmo de controle, quando em execução, não excede os tempos limites de amostragem, preservando, assim a integridade do contexto.

5.9 Conversor Simulink/RTW - Constellation

O módulo de conversão Simulink-RTW, foi desenvolvido como tentativa de solucionar o problema de conversão de um modelo especificado em MATLAB/Simulink em código ANSI C++ que interagisse com o framework Constellation num ambiente de tempo real.

Em muitos casos, ao se especificar um algoriítmo de controle, por exemplo, ocorrem situações que são inerentes e/ou específicas de um determinado problema, inviabilizando ou mesmo impossibilitando uma solução genérica de software para a conversão automática de códigos numa determinada linguagem de programação.

Nestes casos, onde a especialização é inevitável, utiliza-se um recurso oferecido por algumas linguagens de programação e que surgiu com o advento da programação orientada a objetos, que é a herança ou especialização de uma estrutura de dados com base em outras, estruturas de dados, porém virtuais (também conhecidas conceituamente como interfaces) (GAMMA, et al., 1995).

Uma arquitetura de software baseado em interfaces possui um elevado grau de reutilização, sendo posível alterar o comportamento de um sistema possuindo apenas um núcleo reduzido com de estruturas de dados bem definidas. Por isso mesmo a maior parte dos frameworks orientados a objetos desenvolvidos são orientados a interfaces, podendo ser utilizados nas mais variadas aplicações.

Esta abordagem - reimplementação de interfaces específicas dos ambientes MATLAB/Simulink e Constellation, permite com que comportamento e/ou ações específicas sejam inseridas a partir do modelo (utilização de um controlador PID, ou PD, ou remapeamento de uma saída ora para uma porta serial, ora para uma arquivo, por exemplo) e geradas em código ANSI C++ de forma amigável (sem a necessidade de efetuar alterações no código gerado).

O ambiente MATLAB/Simulink, por concepção, já possui embutido em seu ambiente estruturas "virtuais" semelhantes, sendo possível expandi-lo ou executar algumas de suas funcionalidades através de uma aplicação externa, bastando para isso reimplementá-las de acordo com as especificações da tecnologia COM/DCOM (Component Object Model) sob a qual ele foi construído. Na prática isso significa criar uma aplicação específica que utiliza recursos da engine do MATLAB. Desse modo, torna-se viável a interação com o framework Constellation, executado num ambiente de tempo real, com possibilidade para especificar prioridades de acordo com os componentes do modelo, avaliar contadores e temporizadores para computação absoluta de tempo, e utilizar estruturas como semáforos, monitores e buffers duplos com intenção de garantir a integridade de dados, determinismo e performance.

Esse software, porém, contém algumas limitações, como:

- Falhas de interpretação quando da inserção de códigos não ANSI C++;
- Utilização de templates ou extensões que se encontram no ramo da metaprogramação em linguagem C++;
- Não trabalhar com alocação dinâmica - todos os vetores, por exemplo, possuem alocação estática;

- Caso seja necessária a alocação dinâmica, esta deveria ser realizada manualmente (diretamente no código em C++ que correm o risco de serem apagadas no caso de uma subsequente conversão).

O conversor possui as seguintes características:

- Suporte à conversão de blocos do tipo S-Functions do MATLAB/Simulink, sendo permitido alterar, para cada bloco convertido, taxas de periodicidade diferentes;
- Gerar código para a captura de eventos, como interrupções de hardware, por exemplo;
- Disparar subsistemas como tarefas independentes, favorecendo testes com tarefas concorrentes;
- Permite a atribuição de prioridades para os diferentes subsistemas.
- Converte apenas código em ANSI C++;
- Necessita do MATLAB/Simulink instalado, devido à utilização da arquitetura COM/DCOM deste.

Parte da Implementação e comentários adicionais sobre o conversor encontram-se no Apêndice C.

6. RESULTADOS

Neste capítulo são apresentados os resultados obtidos com a aplicação do ambiente de simulação com hardware in the loop proposto sobre o modelo do Hornet, passando pela conversão do modelo em Matlab/Simulink em código executável.

O modelo dinâmico do UUV e seu algoritmo de controle, publicados em (BRAUNL, et al., 2004) foram reproduzidos em MATLAB/Simulink com sucesso. Simulações feitas em MATLAB/Simulink reproduziram com fidelidade o comportamento do UUV descrito na literatura.

6.1 Modelo Dinâmico do UUV adaptado para execução no ambiente Vx Works

Foi necessário realizar adaptações no modelo dinâmico original do UUV, como pode ser observado no Apêndice A, para eliminar loops algébricos do modelo em Simulink fornecido pelos autores do Hornet que impediam a sua conversão para código executável em Vx Works. Esses loops são causados por realimentação direta, em que uma saída é direcionada à entrada do mesmo bloco, ou quando um bloco é realimentado por um outro bloco que também tem realimentação.

6.2 Controlador do UUV adaptado para execução no computador embarcado

Após a geração de código, é possível visualizar, através do Apêndice B, o código C++ correspondente a parte de um bloco de controle selecionado do modelo do UUV, com cálculos e interações de entrada e saída.

6.3 Simulações com o controlador embarcado

Os gráficos apresentados a seguir procuram mostrar que o sistema de controle embarcado no PC104 consegue gerar os sinais de atuação em conformidade com os resultados encontrados na literatura. Ou seja, o controlador

implementado em MATLAB/Simulink foi portado com sucesso para o PC104. Os gráficos foram gerados da seguinte forma:

1. Amostras dos sinais representando os sensores foram gravados em arquivos, que foram transferidos para o sistema de arquivos do PC104;
2. O código do controlador (embarcado no PC104) leu as amostras dos sinais e calculou os sinais de atuação dos propulsores, segundo o algoritmo de controle implementado.
3. Os sinais de saída que seriam enviados aos propulsores no UUV foram gravados em arquivos no PC104.
4. Por fim, esses arquivos de atuação foram alimentados no simulador dinâmico do UUV, reproduzindo os gráficos de deslocamento e velocidade (curvas indicadas com o rótulo “Reproduzido”), que foram plotados juntamente com as curvas geradas pelo simulador original.

6.3.1 Simulação 1: Descolamento em direção a um ponto específico

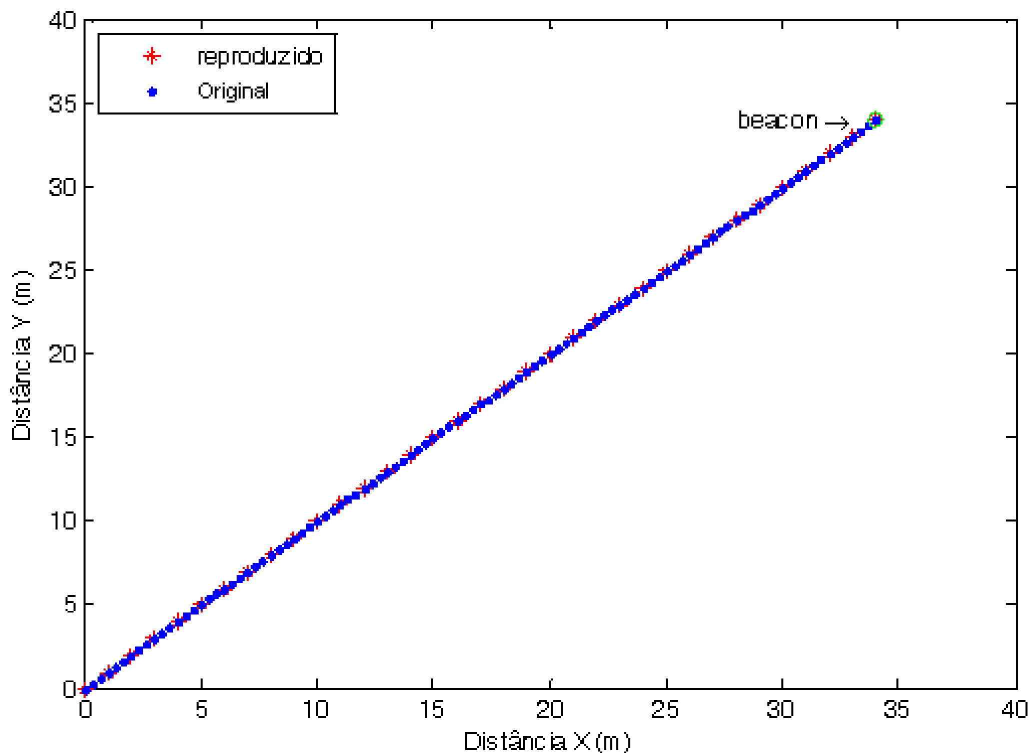

Figura 17. Visualização do caminho percorrido pelo UUV.

Do ponto de partida (0,0), o UUV percorre o mapa por ele conhecido até encontrar o ponto indicado (target), Figura 17 , que encontra-se, naquele momento, nas coordenadas (35,35) do mapa. A distância percorrida pelo UUV é de aproximadamente 49 m. Note que o resultado é compatível com o resultado publicado pelos autores do Hornet em (BRAUNL, et al., 2004),e o mesmo pode-se dizer para os resultados apresentados a seguir.

6.3.2 Simulação 2: Controle de Profundidade

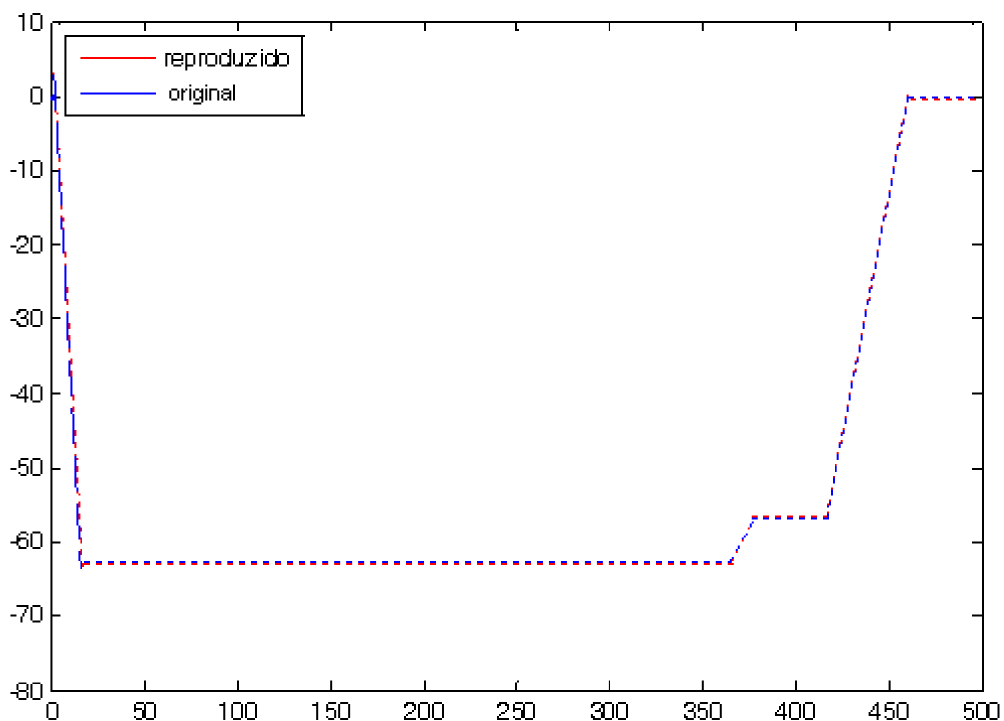

Figura 18. Percurso em profundidade empreendido pelo UUV.

Na simulação apresentada pela Figura 18, o veículo navega até a localização de seu alvo. Durante esta missão, o veículo foi comandado para descer em duas diferentes profundidades, uma a 64 metros e depois sobe a 55 metros, e a uma distância de 30 metros do alvo, o veículo sobe a 1 metro da superfície. O controlador PID de profundidade comportou-se bem como um controlador PD com $k_p = 10$, $k_i = 0$ e $k_d = 5$.

6.3.3 Simulação 3: Controle de Direção

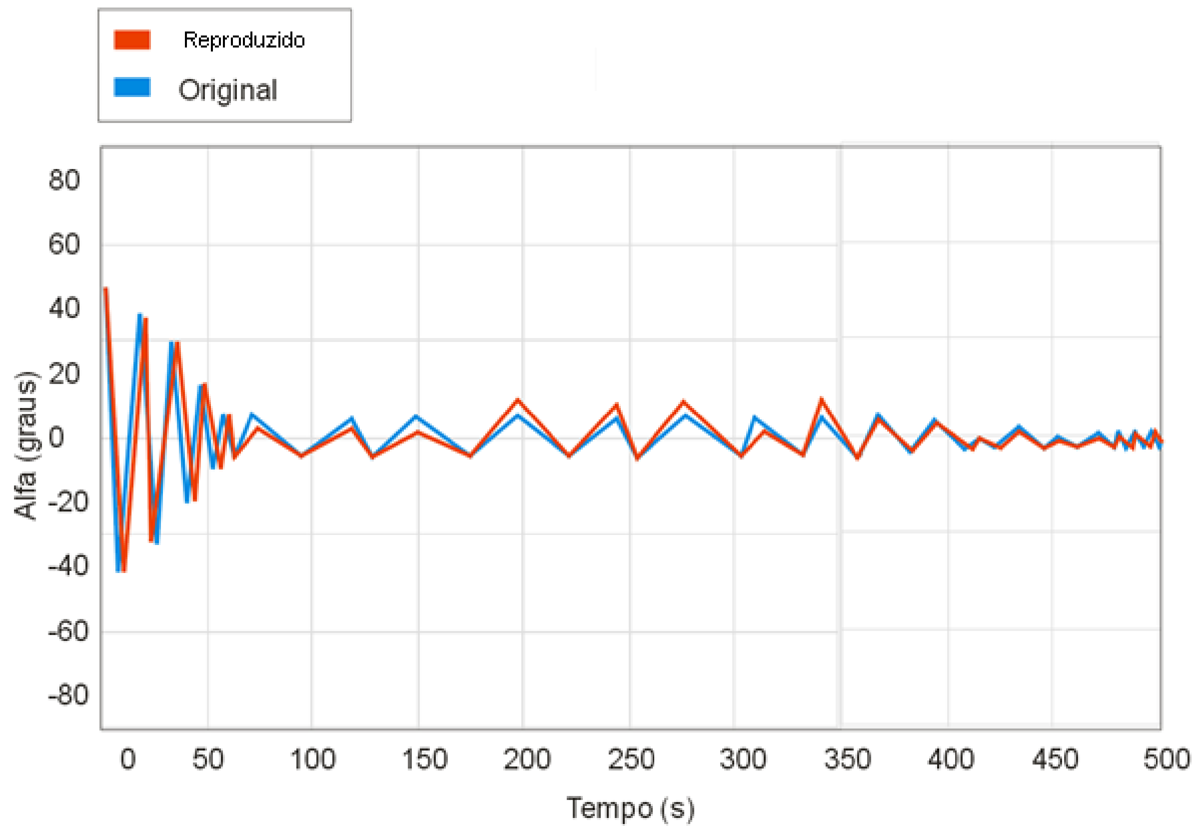

Figura 19. Performance do controlador de direção em modo deslizante.

Apesar do controlador híbrido de direção e velocidade estar habilitado para fazer com que o veículo alcance o alvo, ele apresentou deficiências, fazendo com que o ângulo de *yaw* do veículo oscilasse significativamente em até +/- 45 graus nos instantes iniciais da simulação, e posteriormente limitando as oscilações para ângulos abaixo de 5 graus, como mostra a Figura 19.

6.3.4 Simulação 4: Controle de Velocidade

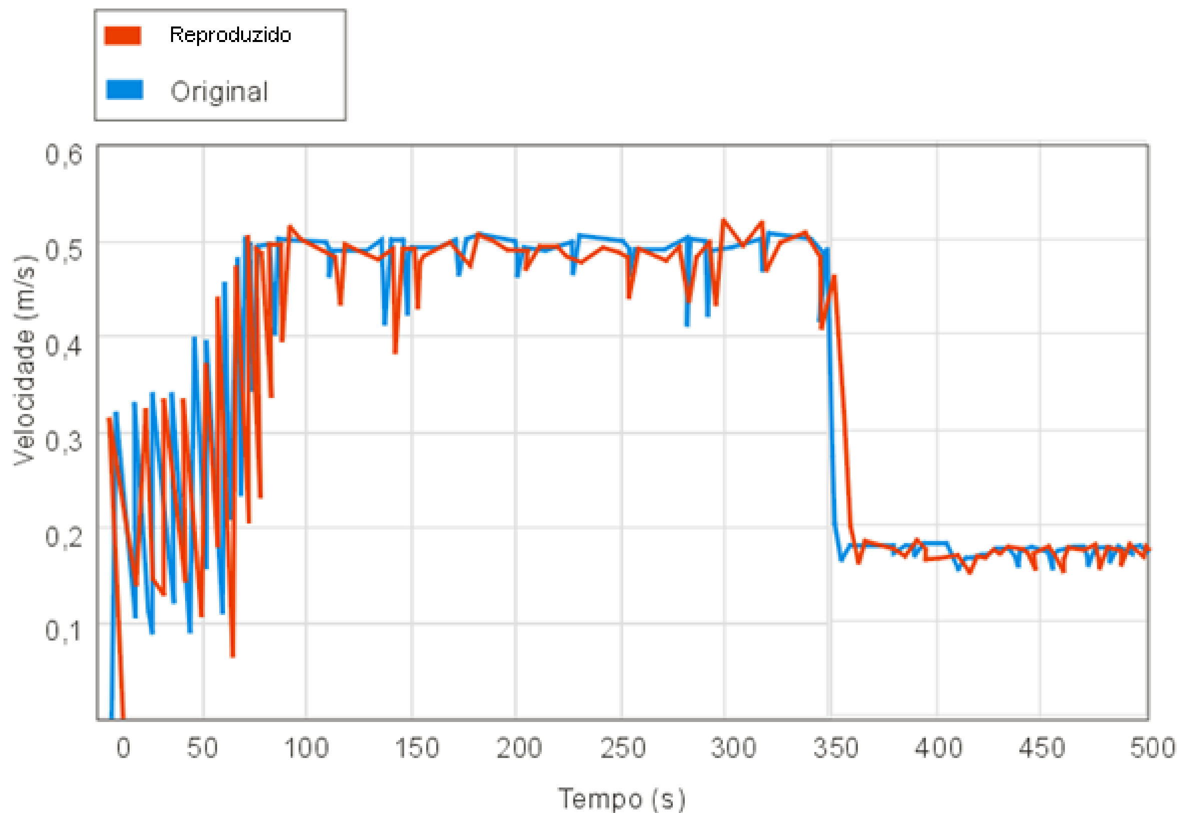

Figura 20. Performance do Controlador de velocidade PID.

O controlador de velocidade PID, integrado ao controlador de deslocamento, apresentou dificuldades em manter a velocidade, como mostra a Figura 20, porque foi constantemente chaveado em favor das correções de erros. Apesar disso, o UUV permaneceu seguindo em frente. Uma vez mais o comportamento apresentado pelo controlador embarcado apresenta comportamento similar aos resultados originalmente publicados para o mesmo modelo de UUV.

6.4 Integridade do contexto em ambiente de tempo real

Pela Figura 21 é possível observar que o ciclo de controle implementado, encapsulado pela tarefa CSSysClockData e indicado na circunferência em vermelho, não excedeu o seu período de tempo permitido de execução, na realidade, considerando que o período é de aproximadamente 15 s (indicado pelas barras vermelhas) e que a execução do módulo de controle só leva 2 s para executar suas rotinas, significa dizer que a implementação do algoriítmo de controle consome menos de 15% do tempo disponível. Dessa forma o sistema de hardware utilizado possui capacidade de processamento suficiente para atender o algoritmo de controle implementado.

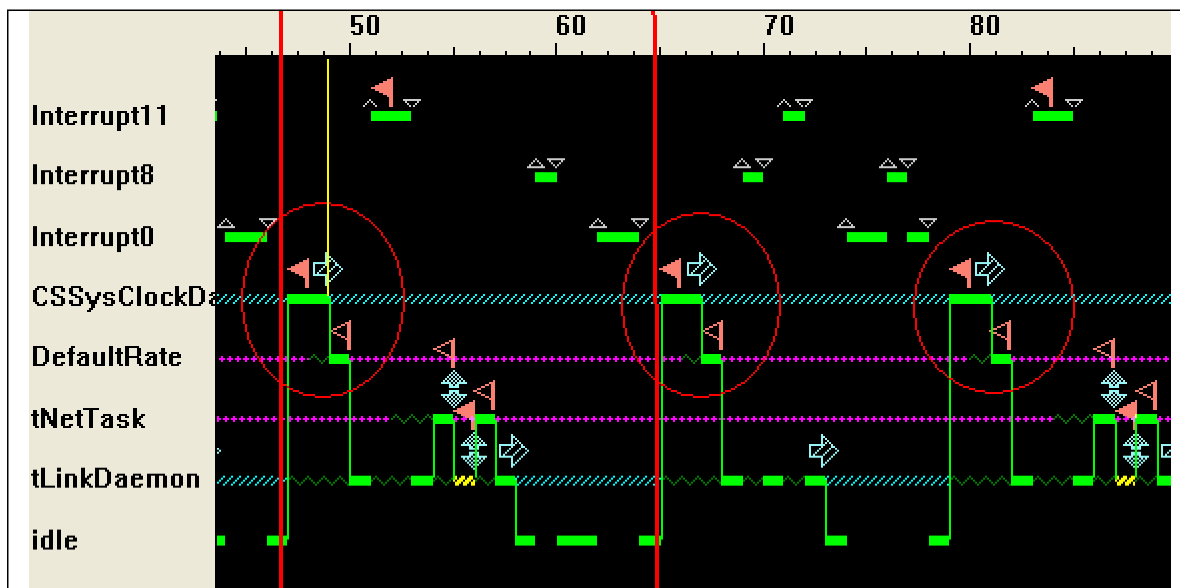

Figura 21. Visualização de tempo de execução e troca de contexto entre tarefas.

Este tipo de análise permite avaliar o relacionamento existente entre o hardware a ser utilizado no UUV e o algoritmo de controle implementado, buscando uma forma de compatibilizá-los de acordo com o resultado obtido.

7. CONCLUSÕES

O Modelo dinâmico e controlador do UUV Hornet foi reproduzido com sucesso em ambiente MATLAB/Simulink. O resultados das simulações são compatíveis com o da literatura disponível, mesmo após a realização de alguns ajustes, como a adição de alguns atrasos para evitar a formação de *loops* algébricos - uma limitação do próprio ambiente MATLAB/Simulink que impede a conversão de código em C++.

O desenvolvimento de um conversor foi necessário para aumentar a velocidade de geração do código de controle a partir do modelo em Simulink. Com o conversor, evita-se lidar com complexidades desnecessárias (como depurando a engine do MATLAB/Simulink, por exemplo), De fato sua construção foi bastante útil e essencial para as atividades que consistiam na conversão de código e atualização do software do computador embarcado.

A simulação com hardware in the loop funcionou de acordo com o esperado, comportando-se adequadamente nas etapas de leitura de um modelo de controle de um UUV, conversão daquele modelo em código executável e validação do algoritmo implementado em ambiente real. O controlador embarcado mostrou comportamento muito similar ao disponível na literatura e atendeu às restrições de tempo e processamento impostas pelo hardware utilizado.

Algumas melhorias ainda podem ser feitas em relação à geração automática de código em ANSI C++. Por exemplo, possibilitar a utilização de alocação dinâmica de memória; possibilitar a utilização de templates C++, pelo menos a STL (Standard Temple Library); além de seguir uma linha de padronização de código comum entre os desenvolvedores C++ que favoreça a leitura código gerado.

Pode-se verificar a integridade de contexto com o software de controle rodando no PC104 por meio de ferramenta do ambiente Vx Works. Trata-se de um funcionalidade bastante útil, porque através dela o engenheiro de controle pode avaliar se o algoritmo em desenvolvimento está compatível com as especificações do hardware utilizado como computador embarcado; análise que, dependendo do resultado, pode levar a uma melhoria do algoritmo (em termos de performance) ou mesmo apontar a necessidade de se trocar o hardware por outro com melhor desempenho .

Com a utilização da estrutura montada para a realização das simulações com *hardware in the loop*, parte das complexidades inerentes ao se lidar com sistemas operacionais de tempo real (sincronização, escalonamento, prioridades, etc), são tratadas de forma quase transparente, de forma a ajudar o projetista ou engenheiro de controle a ter um foco maior sobre o problema.

Problemas encontrados durante o envio/recepção de comandos e atualização remota de bibliotecas sugerem o desenvolvimento de um modelo de carga computacional com enfoque na comunicação com processos e eventos externos de forma que seja possível avaliar melhor o comportamento dos algoritmos de controle implementados na ocorrência dos mesmos.

REFERÊNCIAS BIBLIOGRÁFICAS

(ADAKAWA, 1995) **ADAKAWA K.** “Development of AUV: Aqua explorer 1000”. Design and Control In Underwater Robotic Vehicles. Albuquerque : TSI, 1995.

(ADAM, 1985) **ADAM J.A.** “Probing beneath the sea”. IEEE Spectrum, 1985. Vols. pp: 55–64.

(ADAM, 1991) **ADAM J.D.** “Using a micro-sub for in-vessel visual inspection”. Nuclear Europe Worldscan, 1991. Vols. pp: 5–6.

(ALLES, et al., 1994) **ALLES S., SWICK C. e HOFFMAN M. et al** "A Real Time Hardware in the Loop Simulation for Traction Assist". International Journal of Vehicle Design, 1994. Vols. 15, pp. 597-625.

(ALMEIDA, 1999) **ALMEIDA P. E. M., SIMÕES, M. G.** "Projeto de um sistema robótico inteligente para instalação de equipamentos em poços petrolíferos em águas profundas". São Paulo : IV SBAI – Simpósio Brasileiro de Automação Inteligente, 1999.

(ANAKWA, et al., 2002) **ANAKWA W. K. [et al.]** "Environments for rapid implementation of control algorithms and hardware in the loop simulation". In IEEE 28th Annual Conference of the Industrial Electronics Society, 2002.

(ASHLEY, 1993) **ASHLEY S.** “Voyage to the bottom of the sea”. Mechanical Engineering, 1993. Vols. pp: 52–57.

(BELLINGHAM, 1993) **BELLINGHAM J.G., CHRYSSOSTOMIDIS, C.** “Economic ocean survey capability with AUVs”. Sea Technology, 1993. Vols. pp: 12–18.

(BELLINGHAM, 1996) **BELLINGHAM J.G., WILLCOX, J.C.** “Optimizing AUV oceanographic surveys”. IEEE Sym of AUVT, 1996. Vols. pp: 391–398.

(BLIDBERG, 1991) **BLIDBERG D.R.** “Autonomous underwater vehicles: A tool for the ocean”. Unmanned Systems, 1991. Vols. pp: 10–15.

(BOVET, et al., 2002) **BOVET D. P. e CESATI M.** "Understanding The Linux Kernel". O'Reilly, 2002. Vol. 2.

(BRAUN, et al., 1983) **BRAUN, M., COLEMAN, C. e DREW, D. “**Differential Equation Models”. Springer-Verlag, NY, 1983.

(BRAUNL, et al., 2004) **BRAUNL T. [et al.]** "The Autonomous Underwater Vehicle Initiative – Project Mako". Singapore : IEEE Conference on Robotics, Automation, and Mechatronics (IEEE-RAM), 2004. Vols. pp: 446-451.

(BRUTZMAN, 1996) **BRUTZMAN D. et all** "NPS Phoenix AUV software integration and in-water testing". Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology, 1996. Vols. pp. 99-108.

(BRUTZMAN, et al., 1992) **BRUTZMAN D. P., KANAYAMA Y. e ZIDA M. J** “Integrated Simulation for Rapid Development of Autonomous Underwater Vehicles”. IEEE Symposium On Autonomous Underwater Vehicle

Technology., 1992.

(BUZDUNGAN, et al., 1999) **BUZDUNGAN, L., BALLING, O., LEE, P., BALLING, C., FREEMANN, J. e HUCK, F.** “Multirate Integration for Real-Time Simulation of Wheel Loader Hydraulics”. In Proceedings of DETC’99 ASME Design Engineering Technical Conferences, September, Las Vegas, Nevada, USA, 1999.

(CALVO, 2002) **CALVO I. J.** “High order starting iterates for implicit Runge–Kutta methods”. Journal Numeric Analyses, 2002. Vols. 22, pp. 153-166.

(CONSTELLATION, 2005) **CONSTELLATION** "Real Time Innovations - Constellation, Real-Time Software Framework ". Real-Time Innovations, Inc, 2005.

(CREUZE, 2002) **CREUZE V.** "Navigation referencée terrain pour véhicule autonome sous-marin". PhD thesis, Université de Montpellier II, 2002.

(CRISTI, et al., 1990) **CRISTI R., PAPOULIAS-FOTIS A. e FEALEY J.** "Adaptive sliding mode control of autonomous underwater vehicles in the dive plane". IEEE Journal of Oceanic Engineering, 1990.

(DEBITETTO, 1995) **DEBITETTO A.** "Fuzzy logic for depth control of unmanned undersea vehicles". IEEE Journal of Oceanic Engineering, 1995.

(DECARLO, et al., 1996) **DECARLO R.A., HAK S.H. e DRAUKUNOV S.V.** “Variable Structure, Sliding-Mode Controller Design.”. Florida : In The Control Handbook, CRC Press , 1996.

(DSPACE, 2007) **DSPACE** "dSpace – http://www.dspaceinc.com/ww/en/inc/home/applicationfields/other_fields.cfm”. - 2007.

(DUNO, et al., 1994) **DUNO S. E., SMITH S. M. e BETZER P.** “Design of Autonomous Underwater Vehicles for Coastal Oceanography, Underwater Robotic Vehicles. Design and Control”. TSI Press, 1994. Vols. pp. 229-326.

(FOSSEN, 2002) **FOSSEN T.I** "Marine Control Systems: Guidance, Navigation and Control of Ships, Rigs and Underwater Vehicles". Trondheim : Marine Cybemetrics AS, 2002.

(FREEBSD, 1999) **FREEBSD** "The FreeBSD Handbook". The FreeBSD Documentation Project, 1999.

(GAMMA, et al., 1995) **GAMMA, E., HELM, R., JOHNSON, R. e VLISSIDES, J.** "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995

(HEALEY, et al., 1993) **HEALEY J. e LIENARD D.** "Multivariable sliding mode control for autonomous diving and steering of unmanned underwater vehicles". Jalving : IEEE Journal of Oceanic Engineering, 1993.

(HEATH, 1997) **HEATH, M**. “Scientific Computing: An Introductory Survey”. McGraw-Hill, 1997.

(JALVING, 1994) **JALVING B.** "The NDREA-AUV flight control system". IEEE Journal of Oceanic Engineering, 1994.

(JANSSON, et al., 1994) **JANSSON, A. e PALMBERG, J. O**. “Load simulation, a flexible tool

for assessing the performance of hydraulic valves”. In Proceedings of the Fourth Triennial International Symposium on Fluid Control, Fluid Measurement, and Visualisation, Toulouse, France, 1994.

(KAWANO, et al., 2002) **KAWANO H. e URA T.** "Fast reinforcement learning algorithm for planning of non-holonomic autonomous underwater vehicle in disturbance". International Conference on Intelligent Robots and Systems, 2002.

(KEY, 1987) **KEY C.** “Cooperative Planning in the Pilot’s Associate”. Proc. DARPA Knowledge-Based Planning Workshop., 1987.

KIM J. e SRINIVAN M. A. "Computationally efficient technique for real time surgical simulation with force feedback". Proc. 10th Sym. On Haptic Interfaces For Virtual Environment & Teleoperator Systems, 2002.

(KOK, 1984) **KOK K., LAW, T., BARTILSON, B., RENNER, G.F., ROSEN, K.** “Application of robotic systems to nuclear power plant maintenance tasks”. In Proceedings of the 1984 National Topical Meeting on Robotics and Remote Handling in Hostile Environments, 1984. Vols. pp: 161–168.

(LAMBERT, 1991) **LAMBERT, J.** “Computational Methods in Ordinary Differential Equations”. John Wiley & Sons, 1991.

(LEPAGE, et al., 2000) **LEPAGE Y. G. e HOLAPPA K. W.** "Simulation and control of an autonomous underwater vehicle equipped with a vectored thruster". MTS/IEEE OCEANS, 2000.

(LRT, 2001) **LRT** "National Instruments - LabView Real time User Manual". National Instruments, 2001.

(MACLAY, 1997) **MACLAY D.** “Simulation Gets Into the Loop”. IEEE Review, 1997.

(MARTINS-ENCARNAÇÃO, 2002) **MARTINS-ENCARNAÇÃO P.** "Nonlinear path following control system for ocean vehicles". PhD thesis, Universidade Técnica de Lisboa, 2002.

(MATHWORKS, 2007) **MATHWORKS** "The MathWorks - http://www.mathworks.com/company/user_stories/". 2007.

(MATLAB, 2000) **MATLAB** "The Math Works Inc. MATLAB User's Guide". 2000.

(MATRIxX, 2003) **MATRIxX** "National Instruments - MatrixX Getting Started". National Instruments Inc., 2003.

(NATIONAL, 2007) **NATIONAL** "National Instruments - http://www.ni.com/realtime”. 2007.

(OGATA, 1997) **OGATA K.** “Modern Control Engineering”. New Jersey : Prentice Hall, 1997. Vol. Third Edition.

(OLGAC, et al., 1991) **OLGAC N., PLATIN B.E. e CHANG J. M.** “Sliding Mode Control of Remotely Operated Vehicles for Horizontal Plane Motions.”. IEEE Proceedings, 1991. Vol. 138.

(PAN-MOOK, et al., 1999) **PAN-MOOK L., SEOK-WON H. e YONG-KON L.** "Discrete-time quasi-sliding mode control of an autonomous underwater vehicle". IEEE Journal of Oceanic Engineering, 1999.

(PIC, 2001) **PIC** "Microchip - PIC16F87X Data Sheet". Microchip, 2001.

(PRASETIAWAN, et al., 1999) **PRASETIAWAN E. A. [et al.]** "Modeling and control design of a powertrain simulation testbed for earthmoving vehicles". Journal of Fluid Power Systems and Technology, 1999

(QNX, 1996) **QNX** "QNX Operating System – System Architecture". Ontario, Canada : QNX Software Systems, 1996.

(ROBERSON, et al., 1997) **ROBERSON J. A. e CROWE C. T.** "Engineering Fluid Mechanics". New York : John Wiley & Sons, 1997. Vol. Sixth Edition.

(ROBINSON, 1986) **ROBINSON R.C.** "National defense applications of autonomous underwater vehicles". IEEE J. Oceanic Engineering, 1986. Vol. pp: 11.

(RODRIGUES, et al., 1996) **RODRIGUES L., TAVARES P. e PRADO M.** "Sliding mode control of an AUV in the diving and steering plane". MTS/IEEE OCEANS, 1996.

(RTW, 2000) **RTW** "The Math Works Inc. Real-Time Workshop User's Guide". 2000.

(SANTOS, 1995) **SANTOS S. A.** "Contribution à la Conception des sous-marins autonomes : Architecture des actionneurs, architecture des capteurs d'altitude et commandes référencées capteurs". PhD thesis, Ecole Nationale Supérieure des Mines de Paris., 1995.

(SCHIELA, et al., 2002) **SCHIELA, A. e BONERMANN, F.** "Sparsing in Real Time Simulation". Accepted for publication in ZAMM Z. Math. Mech, 2002.

(SCOPETOOLS, 2005) **SCOPETOOLS** "Windriver's ScopeTools Suite". Windriver Corp., 2005.

(SEAL, 2005) **SEAL D.** "ARM Architecture Reference Manual". Addison-Wesley, 2005. Vol. 2nd Edition.

(SENTA, et al., 2002) **SENTA Y e OKAMURA E.** "HIL simulation system for hdd servo firmware". IEEE Transactions on Magnetics, 2002. pp: 2204-2207.

(SHAMPINE, 1994) **SHAMPINE, L**. "Numerical solution of ordinary differential equations". Chapman & Hall, 1994.

(SIMULINK, 2000) **SIMULINK** "The Math Works Inc. Simulink User's Guide". 2000.

(SMITH, 1995) **SMITH S., DUNN, S., BETZER, P., HOPKINS, T.** "Design of AUVs for coastal oceanography". In Underwater Robotic Vehicles: Design and Control. Albuquerque : TSI, 1995.

(STAMP, 2004) **STAMP BASIC** "BASIC Stamp Syntax and Reference Manual". Parallax Inc., 2004. Vol. 2.1.

(TORNADO, 2005) **TORNADO** "Windriver - Tornado User's Guide". Windriver, 2005. Vol. 2.2.1.

(TUCKER, 1986) **TUCKER J.B.** "Submersibles reach newdepths". High Technology, 1986. Vols. pp: 17–24.

(VXWORKS, 2005) **VXWORKS** "Windriver - VxWorks Programmer's Guide". Windriver, 2005. Vol. 5.5.

(WANG, 1993) **WANG H.H., MARKS, R.L., ROCK, S.M.** “Task-Based Control Architecture for an Untethered,Unmaned Submersible”. 8th International Symposium on Unmaned Untethered Submersible Technology, UNH, 1993. Vols. pp: 1-12.

(XPC, 2002) **XPC** "Mathworks - xPC Target Quick Reference Guide for v2.5". Mathworks Inc., 2002.

(YORGER, 1991) **YORGER D.N., BRADLEY, A.M., WALDEN, B.B.** “The autonomous benthic explorer”. Unmanned Systems, 1991. Vols. pp: 17–23.

(YUH, 2000) **YUH J.** “Design and Control of Autonomous Underwater Robots: A Survey”. Honolulu, Hawaii : Autonomous Systems Laboratory, University of Hawaii, 2000.

(YURODA, et al., 1996) **YURODA Y., ARAMAKI K. e URA T.** “AUV Test Using Real/Virtual Synthetic World”. IEEE Symp. On Autonomous Underwater Vehicle Technology, 1996.

(ZHANG, et al., 2003) **ZHANG R., CARTER D. E. e ALLEYNE A. G.** "Multivariable control of an earthmoving vehicle powertrain experimentally validated in an emulated working cycle". Washington : In Proc ASME International Mechanical Engineering Congress & Exposition, 2003.

(ZHEN, et al., 2004) **ZHEN L., KYTE M. e JOHNSON B.** "Hardware in the loop real-time simulation interface software design". In Proc of The 7th International IEEE Conference on Intelligent Transportation Systems, 2004. pp: 1012-1017.

Apêndice A

Adaptação do modelo originalmente publicado de forma a evitar a existência de loops algébricos.

Apêndice B

Parte do código gerado a partir do modelo dinâmico do UUV implementado em MATLAB/Simulink e preparado para ser executado no sistema operacional VxWorks.

O trecho de código a seguir é interessante por apresentar algumas funções essenciais para o correto funcionamento do algoritimo de controle (*uuv_model_Outputs_wrapper, uuv_model_Derivatives_wrapper*) e esclarecer que estas funções são chamadas periodicamente, numa sequência lógica, pelo ambiente de execução sobre o VxWors.

```
/*
*
*   This file is a wrapper S-function produced automatically.
*   Changes made outside these fields will be lost the next
*   time the block is used to load, edit, and resave this file.
*   This file will be overwritten by the S-function Builder block.
*   If you want to edit this file by hand, you must change it only
*   in the area defined as:
*
*        %%%-SFUNWIZ_wrapper_XXXXX_Changes_BEGIN
*            Your Changes go here
*        %%%-SFUNWIZ_wrapper_XXXXXX_Changes_END
*
*   For better compatibility with the Real-Time Workshop, the
*   "wrapper" S-function technique is used.  This is discussed
*   in the Real-Time Workshop User's Manual in the Chapter titled,
*   "Wrapper S-functions".
*
*   Created: Wed Mar  7 18:43:24 2007
*/

/*
* Include Files
*
*/
#include "simstruc.h"
#include "aStarLibrary.h"
#include "uuv_modelMdl.h"
#include "states.h"

static double stepX = 0;
static double stepY = 0;

#include <math.h>
#define u_width 1
#define y_width 1

#ifndef NOUTPUTS
#define NOUTPUTS 8
#endif

#ifndef NINPUTS
```

```
#define NINPUTS 7
#endif

/*
* Create external references here.
*
*/

/*
 * Output functions
 *
 */
void uuv_model_Outputs_wrapper(const real_T *x0,
                          real_T *sys,
                          const real_T *xC)
{
      int_T i;

      for(i=0; i < 8; i++)
      {
            sys[i] = dStates[i];
      }

}

/*
  *  Derivatives function
  *
  */
void uuv_model_Derivatives_wrapper(const real_T *x0,
                          const real_T *sys,
                          real_T *dx,
                          real_T *xC)
{
        real_T Fdx = 0;
        real_T Fdy = 0;
        real_T Fdz = 0;
        real_T eml_em_dv0[2];
        real_T eml_em_dv1[2];
        real_T eml_BGxform[4];
        real_T eml_input[2];
        real_T eml_geoXY[2];
        int32_T eml_em_i0;
        int32_T eml_em_i1;
        ExternalInputs_uuv_modelMdl *rtwU = NULL;

        /*Define vehicle in-water weight*/
        const real_T W = 0; /*assume neutrally buoyant*/

/*Define Vehicle mass (slugs) and moment of inertia about    verticalaxis*/
        const real_T m = 6.2; /*approx 200 lbs.*/

        /*Define horizontal thruster separation distance (ft)*/
const real_T d = 0.91667; /*approx. 11" between centers of thrusters*/

        real_T Iz = 0.5 * m * (1.969 * 1.969); /*%Iz = 0.5*m*A^2;*/

        /* Definition of inputs*/
        real_T T1 = x0[0];    /*Thrust force from the port-side thruster*/
real_T T2 = x0[1];      /*Thrust force from the starboard-side
thruster*/
```

```
      real_T T3 = x0[2];      /*Thrust force from the vertical thruster*/
real_T Fxext = x0[3]; /*External disturbance forces (water currents, etc) -
X-axis*/
real_T Fyext = x0[4]; /*External disturbance forces (water currents, etc) -
Y-axis*/
real_T Fzext = x0[5]; /*External disturbance forces (water currents, etc) -
Z-axis*/
real_T Mzext = x0[6]; /*Momento of inertia about the (vertical) Z-axis*/

      /* Definition of states*/
real_T Xx = dStates[0];/*(surge) - linear motion with respect to the
longitudinal axis*/
real_T Yy = dStates[1];/*(sway) - linear motion with respect to the
transverse axis*/
real_T Zz = dStates[2];/*(heave) - linear motion with respect to the
vertical axis*/
real_T psi = dStates[3];/*(yaw) - rotational motion about the Z-axis -
radians*/
        real_T xdot = dStates[4];
        real_T ydot = dStates[5];
        real_T zdot = dStates[6];
        real_T psidot = dStates[7];

        /*Restrict thrust force to actual motor limits (assume 10 lbf),
in case the controllers give us some outrageous control signal to
meet*/

        if(T1 > 10)
        {
              T1 = 10;
        }
        else
        {
              if(T1 < -10)
              {
                    T1 = -10;
              }
        }
        if(T2 > 10)
        {
              T2 = 10;
        }
        else
        {
              if(T2 < -10)
              {
                    T2 = -10;
              }
        }
        if(T3 > 10)
        {
              T3 = 10;
        }
        else
        {
              if(T3 < -10)
              {
                    T3 = -10;
              }
        }
```

```
/*Calculate hydrodynamic drag forces
from the equation Fd = Cd*Ap*rho*(Vo^2/2)
ASSUME: AUV is 28" tall, 21" dia. cylinder 24/19
rho (seawater) = 1.99
Cd = 1.0
Ap = 4.083 sq.ft.*/
Fdx = 1.0 * 4.083 * 1.99 * ((xdot * xdot)/2);
if(xdot > 0)
{
      Fdx = -Fdx;
}

/*Cd = 1.0
Ap = 4.083 sq.ft.*/
Fdy = 1.0 * 4.083 * 1.99 * ((ydot * ydot)/2);
if(ydot > 0)
{
      Fdy = -Fdy;
}

/*Cd = 0.9 for cylinder moving parallel to flow with L/D = 1
Ap = 2.405 sq.ft.*/
Fdz = 0.87 * 2.405 * 1.99 * ((zdot * zdot)/2);
if(zdot > 0)
{
      Fdz = -Fdz;
}

/*Calculate state derivatives*/

dx[0] = xdot + (Yy * psidot);
dx[1] = ydot - (Xx * psidot);
dx[2] = xC[6]; /*Zdot*/
dx[3] = xC[7]; /*Psidot*/
dx[4] = ((T1 + T2 + Fdx + Fxext)/m) + (ydot * psidot); /*Acceleration
about X-axis*/
dx[5] = ((Fdy + Fyext)/m) - (xdot * psidot); /*Acceleration about Y-axis*/
dx[6] = (T3 + Fdz + Fzext - W)/m; /*Acceleration about Z-axis*/
dx[7] = (((T2-T1)*(d/2)) + Mzext)/Iz; /*Sum of moments about Z-axis*/

rtwU = (ExternalInputs_uuv_modelMdl*)  rtmGetU(myUUVModel);

/* Define coordinate transformation matrix */
eml_em_dv0[0] = cos(0);
eml_em_dv0[1] = -sin(0);
eml_em_dv1[0] = sin(0);
eml_em_dv1[1] = cos(0);

for(eml_em_i0 = 0; eml_em_i0 < 2; eml_em_i0++)
  {
  eml_BGxform[eml_em_i0 << 1] = eml_em_dv0[eml_em_i0];
  eml_BGxform[1 + (eml_em_i0 << 1)] = eml_em_dv1[eml_em_i0];
}

  /* Convert body-fixed coords. to geographical coords. */
eml_input[0] = rtwU->BeaconX;
eml_input[1] = rtwU->BeaconY;
for(eml_em_i1 = 0; eml_em_i1 < 2; eml_em_i1++)
  {
  eml_geoXY[eml_em_i1] = 0.0;
  for(eml_em_i0 = 0; eml_em_i0 < 2; eml_em_i0++)
```

```
            {
          eml_geoXY[eml_em_i1] += eml_BGxform[eml_em_i1 + (eml_em_i0 << 1)]
* eml_input[eml_em_i0];
        }
      }


      if( FindPath(1, Xx, Yy, eml_geoXY[0], eml_geoXY[1]) == found)
      {
            stepX = ReadPathX(1,1);
            stepY = ReadPathY(1,1);
      }
      dStates[0] = stepX;
      dStates[1] = stepY;
      dStates[2] = Zz;
      dStates[3] = psi;
      dStates[4] = dx[0];
      dStates[5] = dx[1];
      dStates[6] = dx[2];
      dStates[7] = dx[3];

}
```

Apêndice C

O código a seguir em linguagem de programação C++ corresponde à biblioteca de acesso à engine do MATLAB/Simulink. Através dela é possível "automatizar" o uso do MATLAB/Simulink por meio de uma aplicação externa, acessando quando necessário todas as suas variáveis (matrizes), objetos e blocos de controle.

A biblioteca é compilada para o ambiente Windows, necessitando do MATALB instalado. O compilador utilizado foi o Visual Studio 2003/2005, da Microsoft . É necessário mapear os diretórios do MATALB no ambiente de desenvolvimento do Visual Studio.

A linha de comando final para o processo de construlção da biblioteca é a seguinte:

```
Cl /OUT:"C:\TEMP\teste\Debug\engineLib.lib" /INCREMENTAL /NOLOGO /MANIFEST /MANIFESTFILE:"Debug\\engineLib.lib.intermediate.manifest" /DEBUG /PDB:"c:\temp\teste\debug\\engineLib.lib" /SUBSYSTEM:CONSOLE /MACHINE:X86 /ERRORREPORT:PROMPT kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

O código a seguir envolve utilização das seguintes tecnologias de desenvolvimento de software: Arquitetura COM/DCOM, Windows API e utilização de templates C++.

```
/**
* Construtor default
*/

CMatlabEngine::CMatlabEngine(bool bDedicated)
: m_bInitialized(false)
{
#ifndef _UNICODE
	m_pBuffer=new WCHAR[1024];
	m_uBufferSize=1024;
#endif

	m_bstrWorkspace = ::SysAllocString( L"base" );

	m_dpNoArgs.cArgs=0;
	m_dpNoArgs.cNamedArgs=0;
	m_dpNoArgs.rgdispidNamedArgs=NULL;
	m_dpNoArgs.rgvarg=NULL;

	// Get the Class Identifier for Matlab Application Object,
	// which is a globally unique identifier (GUID)
	OLECHAR FAR* szFunction;

	CLSID clsid;
	if (bDedicated)
		m_hr=::CLSIDFromProgID(OLESTR("Matlab.Application.Single"),
&clsid);
	else
		m_hr=::CLSIDFromProgID(OLESTR("Matlab.Application"), &clsid);
	if (FAILED(m_hr))
		return;


	// Create an instance of the Matlab application and obtain the
pointer
	// to the application's IUnknown interface

	IUnknown* pUnk;
	m_hr = ::CoCreateInstance(clsid, NULL,
CLSCTX_SERVER,IID_IUnknown,(void**) &pUnk);
	if (FAILED(m_hr))
		return;

	// Query IUnknown to retrieve a pointer to the application IDispatch
	// interface
	m_hr = pUnk ->QueryInterface(IID_IDispatch, (void**)&m_pMtlbDispApp);
	if (FAILED(m_hr))
		return;

	// Get the Dispatch Identifiers
	szFunction = OLESTR("GetFullMatrix");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_GetFullMatrix);
	if (FAILED(m_hr))
		return;


	szFunction = OLESTR("PutFullMatrix");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_PutFullMatrix);
```

```
	if (FAILED(m_hr))
		return;


	szFunction = OLESTR("Execute");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_Execute);
	if (FAILED(m_hr))
		return;


	szFunction = OLESTR("MinimizeCommandWindow");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_MinimizeCommandWindow);
	if (FAILED(m_hr))
		return;

	szFunction = OLESTR("MaximizeCommandWindow");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_MaximizeCommandWindow);
	if (FAILED(m_hr))
		return;

#ifdef MATLAB_VERSION_6
	szFunction = OLESTR("GetCharArray");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_GetCharArray);
	if (FAILED(m_hr))
		return;

	szFunction = OLESTR("PutCharArray");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_PutCharArray);
	if (FAILED(m_hr))
		return;

	szFunction = OLESTR("Visible");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_Visible);
	if (FAILED(m_hr))
		return;
#endif

	szFunction = OLESTR("Quit");
	m_hr = m_pMtlbDispApp ->GetIDsOfNames(IID_NULL, &szFunction, 1,
LOCALE_USER_DEFAULT, &m_dispid_Quit);
	if (FAILED(m_hr))
		return;

	m_bInitialized = true;
}

CMatlabEngine::~CMatlabEngine()
{
	if (m_pMtlbDispApp)
		m_pMtlbDispApp->Release();

	::SysFreeString(m_bstrWorkspace);

#ifndef _UNICODE
	if (m_pBuffer)
```

```
            delete[] m_pBuffer;
#endif
}

void CMatlabEngine::SetWorkspace(LPCTSTR szWorkspace)
{
      ProcessString( szWorkspace, m_bstrWorkspace);
}


void CMatlabEngine::ErrHandler(HRESULT hr, EXCEPINFO excep, UINT uArgErr)
{
  if (hr == DISP_E_EXCEPTION)
  {
    TCHAR errDesc[512];
    TCHAR errMsg[512];
    ::_tprintf(errMsg, TEXT("Run-time error %d:\n\n %s"),
            excep.scode & 0x0000FFFF,  //Lower 16-bits of SCODE
            errDesc);                  //Text error description
    ::MessageBox(NULL, errMsg, TEXT("Automation Server Error"),
                MB_SETFOREGROUND | MB_OK);
  }
  else
  {
    LPVOID lpMsgBuf;
    ::FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                    FORMAT_MESSAGE_FROM_SYSTEM |
                    FORMAT_MESSAGE_IGNORE_INSERTS, NULL, hr,
                    MAKELANGID(LANG_NEUTRAL,
                    SUBLANG_DEFAULT),(LPTSTR) &lpMsgBuf,
                    0, NULL);
    if ((hr == DISP_E_TYPEMISMATCH ) ||
        (hr == DISP_E_PARAMNOTFOUND))
    {
      TCHAR extMess[512];
      ::_tprintf(extMess, TEXT("%s Position of incorrect argument is
%u.\n"), (LPCTSTR) lpMsgBuf, uArgErr);
      ::MessageBox(NULL, extMess, TEXT("COM Error"), MB_OK |
MB_SETFOREGROUND);
    }
    else
    {
      ::MessageBox(NULL, (LPCTSTR)lpMsgBuf, TEXT("COM Error"),MB_OK |
MB_SETFOREGROUND);
    }
    ::LocalFree(lpMsgBuf);
  }
}

/**
* Start the Matlab's engine execution
*/

HRESULT CMatlabEngine::Execute(LPCTSTR szCode)
{
      VARIANT vArgsTypeText[1];
      DISPPARAMS dpTypeText;
      BSTR bstrName=NULL;

      if (!m_bInitialized || !m_pMtlbDispApp)
            return m_hr=S_FALSE;
```

```
	ProcessString(szCode, bstrName);

	vArgsTypeText [0].vt = VT_BSTR;
	vArgsTypeText [0].bstrVal = bstrName;

	dpTypeText.cArgs = 1;
	dpTypeText.cNamedArgs = 0;
	dpTypeText.rgvarg = vArgsTypeText;

	m_hr = m_pMtlbDispApp ->Invoke(m_dispid_Execute, IID_NULL, 
LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dpTypeText, &m_vResult, &m_excep, 
&m_uArgErr);

	if (FAILED(m_hr))
		goto Exit;

// Cleaning memory...
Exit:
	::SysFreeString(bstrName);

	return m_hr;
}

/**
* Presents or not the Matlab's windows
*/

#ifdef MATLAB_VERSION_6
HRESULT CMatlabEngine::Show(bool bShow)
{
	VARIANT vArgsTypeLong[1];

	if (!m_bInitialized || !m_pMtlbDispApp)
		return m_hr=S_FALSE;

	::VariantInit(vArgsTypeLong);

	vArgsTypeLong[0].vt = VT_I4;
	vArgsTypeLong[0].lVal = (bShow) ? 1 : 0; // Visible = 1; Invisible = 
0

	DISPID dispidNamed = DISPID_PROPERTYPUT;

	DISPPARAMS dpVisible;
	dpVisible.cArgs = 1;
	dpVisible.cNamedArgs = 1;
	dpVisible.rgvarg = vArgsTypeLong;
	dpVisible.rgdispidNamedArgs = &dispidNamed;

	return m_hr = m_pMtlbDispApp ->Invoke(m_dispid_Visible, 
IID_NULL,LOCALE_USER_DEFAULT, DISPATCH_PROPERTYPUT, &dpVisible, NULL, 
&m_excep, &m_uArgErr);
}
```

```
/**
* Verify if the Matlab's Window is visible
*/


bool CMatlabEngine::IsVisible()
{
	VARIANT vArgsTypeLong[1];
	if (!m_bInitialized || !m_pMtlbDispApp)
	{
		m_hr = S_FALSE;
		return false;
	}

	::VariantInit(vArgsTypeLong);

	vArgsTypeLong[0].vt = VT_I4;
	vArgsTypeLong[0].lVal = 0; // Visible = 1; Invisible = 0

	DISPID dispidNamed = DISPID_PROPERTYPUT;

	DISPPARAMS dpVisible;
	dpVisible.cArgs = 1;
	dpVisible.cNamedArgs = 1;
	dpVisible.rgvarg = vArgsTypeLong;
	dpVisible.rgdispidNamedArgs = &dispidNamed;


	m_hr = m_pMtlbDispApp ->Invoke(m_dispid_Visible, IID_NULL,
		LOCALE_USER_DEFAULT,
		DISPATCH_PROPERTYGET, &m_dpNoArgs, &m_vResult, &m_excep,
&m_uArgErr);

	return m_vResult.lVal != 0;
}
#endif

/**
* Get Matlab's error Messages
*/

void CMatlabEngine::GetLastErrorMessage() const
{
	if(FAILED(m_hr))
		ErrHandler(m_hr, m_excep, m_uArgErr);
}

/**
* Resize the Matlab's Windows
*/

HRESULT CMatlabEngine::MaximiseWidow()
{
	if (!m_bInitialized || !m_pMtlbDispApp)
		return m_hr=S_FALSE;

	return m_hr = m_pMtlbDispApp->Invoke(m_dispid_MaximizeCommandWindow,
		IID_NULL,
		LOCALE_USER_DEFAULT, DISPATCH_METHOD, &m_dpNoArgs, NULL,
		&m_excep, &m_uArgErr);
}
```

```
/**
* Resize the Matlab's Windows
*/

HRESULT CMatlabEngine::MinimizeWindow()
{
      if (!m_bInitialized || !m_pMtlbDispApp)
            return m_hr=S_FALSE;

      return m_hr = m_pMtlbDispApp ->Invoke(m_dispid_MinimizeCommandWindow,
            IID_NULL,
            LOCALE_USER_DEFAULT, DISPATCH_METHOD, &m_dpNoArgs, NULL,
            &m_excep, &m_uArgErr);
}

/**
* Add Matrix Values into Matlab's engine
*/

HRESULT CMatlabEngine::PutMatrix( LPCTSTR szName, const
std::vector<double>& vRealArray, const std::vector<double>& vImgArray, UINT
nRows, UINT nCols)
{
      BSTR bstrName=NULL;

      if (!m_bInitialized || !m_pMtlbDispApp || (vRealArray.size() !=
vImgArray.size()) || (nRows*nCols > vRealArray.size()) )
            return m_hr=S_FALSE;

      ProcessString(szName, bstrName);

      SAFEARRAYBOUND realPartDims[2];
      realPartDims[0].lLbound = 0;      // Lower bound of the first
dimension
      realPartDims[0].cElements = nRows;
      realPartDims[1].lLbound = 0;      // Lower bound of the second
dimension
      realPartDims[1].cElements = nCols;


      SAFEARRAY *realPart = ::SafeArrayCreate(VT_R8, 2, realPartDims);
      SAFEARRAY *imgPart = ::SafeArrayCreate(VT_R8, 2, realPartDims);

      long lIndex[2];
      UINT k;
      double val;

      for (int i = 0; i < nRows; i++)
      {
            lIndex[0] = i;

            for (int j = 0; j < nCols; j++)
            {
                  lIndex[1] = j;

                  k=i*nCols+j;
                  val=vRealArray[k];
                  m_hr = ::SafeArrayPutElement(realPart, lIndex, &val);
                  if (FAILED(m_hr))
                        goto Exit;
```

```
                val=vImgArray[k];
                m_hr = ::SafeArrayPutElement(imgPart, lIndex, &val);
                if (FAILED(m_hr))
                      goto Exit;
          }
    }


    VARIANT vArgPutFullMatrix[4];
    for (i = 0; i < 4; ++i)
          ::VariantInit(&vArgPutFullMatrix[i]);


    V_VT(&vArgPutFullMatrix[0]) = VT_ARRAY | VT_R8;
    V_ARRAY(&vArgPutFullMatrix[0]) = imgPart; // do set to NULL when not
complex
    V_VT(&vArgPutFullMatrix[1]) = VT_ARRAY | VT_R8;
    V_ARRAY(&vArgPutFullMatrix[1]) = realPart;
    V_VT(&vArgPutFullMatrix[2]) = VT_BSTR;
    V_BSTR(&vArgPutFullMatrix[2]) = m_bstrWorkspace;
    V_VT(&vArgPutFullMatrix[3]) = VT_BSTR;
    V_BSTR(&vArgPutFullMatrix[3])= bstrName;


    DISPPARAMS dpPutFullMatrix;
    dpPutFullMatrix.cArgs = 4;
    dpPutFullMatrix.cNamedArgs = 0;
    dpPutFullMatrix.rgvarg = vArgPutFullMatrix;


    m_hr = m_pMtlbDispApp ->Invoke(m_dispid_PutFullMatrix, IID_NULL,
  LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dpPutFullMatrix, NULL, &m_excep,
&m_uArgErr);

    if (FAILED(m_hr))
          goto Exit;

Exit:
    ::SysFreeString(bstrName);
    m_hr = ::SafeArrayDestroy(realPart);
    m_hr = ::SafeArrayDestroy(imgPart);

    return m_hr;
}

/**
* Overrided Put Matrix Method
*/

HRESULT CMatlabEngine::PutMatrix( LPCTSTR szName, const
std::vector<double>& vArray, UINT nRows, UINT nCols)
{
    BSTR bstrName=NULL;

    if (!m_bInitialized || !m_pMtlbDispApp || (nRows*nCols >
vArray.size()) )
          return m_hr=S_FALSE;

    ProcessString(szName, bstrName);

    SAFEARRAYBOUND realPartDims[2];
```

```
	realPartDims[0].lLbound = 0;      // Lower bound of the first 
dimension
	realPartDims[0].cElements = nRows;
	realPartDims[1].lLbound = 0;      // Lower bound of the second 
dimension
	realPartDims[1].cElements = nCols;


	SAFEARRAY *realPart = ::SafeArrayCreate(VT_R8, 2, realPartDims);

	long lIndex[2];
	double val;

	for (int i = 0; i < nRows; i++)
	{
		lIndex[0] = i;

		for (int j = 0; j < nCols; j++)
		{
			lIndex[1] = j;

			val=vArray[i*nCols+j];
			m_hr = ::SafeArrayPutElement(realPart, lIndex, &val);
			if (FAILED(m_hr))
				goto Exit;
		}
	}


	VARIANT vArgPutFullMatrix[4];
	for (i = 0; i < 4; ++i)
		::VariantInit(&vArgPutFullMatrix[i]);


	V_VT(&vArgPutFullMatrix[0]) = VT_ARRAY | VT_R8;
	V_ARRAY(&vArgPutFullMatrix[0]) = NULL; // do set to NULL when not 
complex
	V_VT(&vArgPutFullMatrix[1]) = VT_ARRAY | VT_R8;
	V_ARRAY(&vArgPutFullMatrix[1])= realPart;
	V_VT(&vArgPutFullMatrix[2]) = VT_BSTR;
	V_BSTR(&vArgPutFullMatrix[2]) = m_bstrWorkspace;
	V_VT(&vArgPutFullMatrix[3]) = VT_BSTR;
	V_BSTR(&vArgPutFullMatrix[3]) = bstrName;


	DISPPARAMS dpPutFullMatrix;
	dpPutFullMatrix.cArgs = 4;
	dpPutFullMatrix.cNamedArgs = 0;
	dpPutFullMatrix.rgvarg = vArgPutFullMatrix;

	m_hr = m_pMtlbDispApp ->Invoke(m_dispid_PutFullMatrix, IID_NULL,
		LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dpPutFullMatrix, NULL, 
&m_excep, &m_uArgErr);

	if (FAILED(m_hr))
		goto Exit;

Exit:
	::SysFreeString(bstrName);
	m_hr = ::SafeArrayDestroy(realPart);
```

```
	return m_hr;
}

/**
* Get a matrix from Matlab's Engine
*/
HRESULT CMatlabEngine::GetMatrix( LPCTSTR szName,  UINT& nRows, UINT&
nCols, std::vector<double>& vRealArray, std::vector<double>* pImgArray)
{
	if (!m_bInitialized || !m_pMtlbDispApp)
		return m_hr=S_FALSE;

	static TCHAR tzBuffer[512];
	std::vector<double> vSize;
	BSTR bstrName=NULL;

	ProcessString(szName, bstrName);

	// computing size of matrix
	_stprintf(tzBuffer, _T("%sSize=size(%s);"), szName, szName);
	m_hr = Execute(tzBuffer);
	if (FAILED(m_hr))
		goto Exit;

	_stprintf(tzBuffer, _T("%sSize"), szName);
	m_hr = GetMatrixKnownSize( tzBuffer, 1,2, vSize);
	if (FAILED(m_hr))
		goto Exit;

	nRows = vSize[0];
	nCols = vSize[1];

	// getting matrix...
	if (pImgArray)
		m_hr = GetMatrixKnownSize( szName, nRows,nCols, vRealArray,
*pImgArray);
	else
		m_hr = GetMatrixKnownSize( szName, nRows,nCols, vRealArray);

	if (FAILED(m_hr))
		goto Exit;


Exit:
	::SysFreeString(bstrName);

	return m_hr;
}

/**
* Get Matrix size
*/

HRESULT CMatlabEngine::GetMatrixKnownSize( LPCTSTR szName,  UINT nRows,
UINT nCols, std::vector<double>& vArray)
{
	if (!m_bInitialized || !m_pMtlbDispApp)
		return m_hr=S_FALSE;

	BSTR bstrName=NULL;
	UINT i,j;
```

```
	SAFEARRAY *realPart=NULL;
	SAFEARRAY *imgPart=NULL;
	double*			pDummy = NULL ; // for access to the data
	VARIANT vArgGetFullMatrix[4];


	// Preparing name
	ProcessString(szName, bstrName);

	SAFEARRAYBOUND realPartDims[2];
	realPartDims[0].lLbound = 1;	// Lower bound of the first
dimension
	realPartDims[0].cElements = nRows;
	realPartDims[1].lLbound = 1;	// Lower bound of the second
dimension
	realPartDims[1].cElements = nCols;
	realPart=::SafeArrayCreate(VT_R8, 2, realPartDims);

	SAFEARRAYBOUND imgPartDims[1] = { {0,0}};
	imgPart=::SafeArrayCreate(VT_R8, 1, imgPartDims);


	//loading data...
	for (i = 0; i < 4; ++i)
		::VariantInit(&vArgGetFullMatrix[i]);

	V_VT(&vArgGetFullMatrix[0]) = VT_ARRAY | VT_R8 | VT_BYREF;
	vArgGetFullMatrix[0].pparray = &imgPart; // do set to NULL when not
complex
	V_VT(&vArgGetFullMatrix[1]) = VT_ARRAY | VT_R8 | VT_BYREF;
	vArgGetFullMatrix[1].pparray = &realPart;
	V_VT(&vArgGetFullMatrix[2]) = VT_BSTR;
	V_BSTR(&vArgGetFullMatrix[2]) = m_bstrWorkspace;
	V_VT(&vArgGetFullMatrix[3]) = VT_BSTR;
	V_BSTR(&vArgGetFullMatrix[3]) = bstrName;

	DISPPARAMS dpGetFullMatrix;
	dpGetFullMatrix.cArgs = 4;
	dpGetFullMatrix.cNamedArgs = 0;
	dpGetFullMatrix.rgvarg = vArgGetFullMatrix;

	m_hr = m_pMtlbDispApp ->Invoke(m_dispid_GetFullMatrix, IID_NULL,
		LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dpGetFullMatrix, NULL,
&m_excep, &m_uArgErr);

	if (FAILED(m_hr))
		goto Exit;


	if (::SafeArrayGetDim(realPart) != 2)
	{
		m_hr = S_FALSE;
		goto Exit;
	}

	vArray.resize(nRows*nCols);

	SafeArrayAccessData(realPart, (void HUGEP **)(&pDummy)) ;  // dummy
now points to the data

	// copy each element across into the matrix to return
```

```
	for (i = 0; i < nRows ; ++i)
	{
		for (j = 0; j < nCols ; ++j)
		{
			vArray[i*nCols+j]=pDummy[i*nCols+j] ;
		}
	}
	pDummy = NULL ; // no longer valid

Exit:
	::SysFreeString(bstrName);
	::SafeArrayUnaccessData(realPart) ;// release the safe array data 
pointer
	::SafeArrayDestroy(realPart);
	::SafeArrayDestroy(imgPart);

	return m_hr;
}

/**
* Overrided GetMatrix Size
*/

HRESULT CMatlabEngine::GetMatrixKnownSize( LPCTSTR szName,  UINT nRows, 
UINT nCols, std::vector<double>& vRealArray, std::vector<double>& 
vImgArray)
{
	if (!m_bInitialized || !m_pMtlbDispApp)
		return m_hr=S_FALSE;

	BSTR bstrName=NULL;
	UINT i,j;
	SAFEARRAY *realPart=NULL;
	SAFEARRAY *imgPart=NULL;
	double*			pRealDummy = NULL ;	
	// for access to the data
	double*			pImgDummy = NULL ;	
	// for access to the data
	VARIANT vArgGetFullMatrix[4];


	// Preparing name
	ProcessString(szName, bstrName);

	SAFEARRAYBOUND realPartDims[2];
	realPartDims[0].lLbound = 1;      // Lower bound of the first 
dimension
	realPartDims[0].cElements = nRows;
	realPartDims[1].lLbound = 1;      // Lower bound of the second 
dimension
	realPartDims[1].cElements = nCols;
	realPart=::SafeArrayCreate(VT_R8, 2, realPartDims);

	SAFEARRAYBOUND imgPartDims[2];
	imgPartDims[0].lLbound = 1;      // Lower bound of the first dimension
	imgPartDims[0].cElements = nRows;
	imgPartDims[1].lLbound = 1;      // Lower bound of the second 
dimension
	imgPartDims[1].cElements = nCols;
	imgPart=::SafeArrayCreate(VT_R8, 2, imgPartDims);
```

```
	//loading data...
	for (i = 0; i < 4; ++i)
		::VariantInit(&vArgGetFullMatrix[i]);

	V_VT(&vArgGetFullMatrix[0]) = VT_ARRAY | VT_R8 | VT_BYREF;
	vArgGetFullMatrix[0].pparray = &imgPart; // do set to NULL when not
complex
	V_VT(&vArgGetFullMatrix[1]) = VT_ARRAY | VT_R8 | VT_BYREF;
	vArgGetFullMatrix[1].pparray = &realPart;
	V_VT(&vArgGetFullMatrix[2]) = VT_BSTR;
	V_BSTR(&vArgGetFullMatrix[2]) = m_bstrWorkspace;
	V_VT(&vArgGetFullMatrix[3]) = VT_BSTR;
	V_BSTR(&vArgGetFullMatrix[3]) = bstrName;

	DISPPARAMS dpGetFullMatrix;
	dpGetFullMatrix.cArgs = 4;
	dpGetFullMatrix.cNamedArgs = 0;
	dpGetFullMatrix.rgvarg = vArgGetFullMatrix;

	m_hr = m_pMtlbDispApp ->Invoke(m_dispid_GetFullMatrix, IID_NULL,
		LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dpGetFullMatrix, NULL,
&m_excep, &m_uArgErr);

	if (FAILED(m_hr))
		goto Exit;


	if (::SafeArrayGetDim(realPart) != 2)
	{
		m_hr = S_FALSE;
		goto Exit;
	}

	vRealArray.resize(nRows*nCols);
	vImgArray.resize(nRows*nCols);

	SafeArrayAccessData(realPart, (void HUGEP **)(&pRealDummy)) ; //
dummy now points to the data
	SafeArrayAccessData(imgPart, (void HUGEP **)(&pImgDummy)) ; // dummy
now points to the data

	// copy each element across into the matrix to return
	for (i = 0; i < nRows ; ++i)
	{
		for (j = 0; j < nCols ; ++j)
		{
			vRealArray[i*nCols+j]=pRealDummy[i*nCols+j] ;
			vImgArray[i*nCols+j]=pImgDummy[i*nCols+j] ;
		}
	}
	pRealDummy = NULL ;
	// no longer valid
	pImgDummy = NULL ;
	// no longer valid

Exit:
	::SysFreeString(bstrName);
	::SafeArrayUnaccessData(realPart) ; // release the safe array data
pointer
```

```
	::SafeArrayUnaccessData(imgPart) ; // release the safe array data
pointer
	::SafeArrayDestroy(realPart);
	::SafeArrayDestroy(imgPart);

	return m_hr;
}

/**
* Ends Matlab's engine
*/
HRESULT CMatlabEngine::Quit()
{
	if (!m_bInitialized || !m_pMtlbDispApp)
	{

		m_bInitialized = false;
		return S_FALSE;
	}

	return m_hr = m_pMtlbDispApp ->Invoke(m_dispid_Quit, IID_NULL,
		LOCALE_USER_DEFAULT,
		DISPATCH_METHOD, &m_dpNoArgs, NULL, &m_excep, &m_uArgErr);
}

void CMatlabEngine::ProcessString( LPCTSTR szName, BSTR& bstrName)
{
	::SysFreeString(bstrName);

#ifndef _UNICODE
	int nChar;
	AllocateBuffer(_tcslen(szName));
	nChar=MultiByteToWideChar(CP_ACP,MB_PRECOMPOSED,szName,-
1,m_pBuffer,m_uBufferSize);
	bstrName = ::SysAllocString(m_pBuffer);
#else
	bstrName = ::SysAllocString(OLESTR(szName));
#endif
}

#ifdef MATLAB_VERSION_6
HRESULT CMatlabEngine::PutString(LPCTSTR szName, LPCTSTR szString)
{
	BSTR bstrName=NULL;
	BSTR bstrString=NULL;
	UINT i;

	if (!m_bInitialized || !m_pMtlbDispApp)
		return S_FALSE;

	ProcessString(szName, bstrName);
	ProcessString(szString, bstrString);


	VARIANT vArgPutString[3];
	for (i = 0; i < 3; ++i)
		::VariantInit(&vArgPutString[i]);


	V_VT(&vArgPutString[0]) = VT_BSTR;
	V_BSTR(&vArgPutString[0]) = bstrString;
```

```
	V_VT(&vArgPutString[1]) = VT_BSTR;
	V_BSTR(&vArgPutString[1]) = m_bstrWorkspace;
	V_VT(&vArgPutString[2]) = VT_BSTR;
	V_BSTR(&vArgPutString[2]) = bstrName;


	DISPPARAMS dpPutString;
	dpPutString.cArgs = 3;
	dpPutString.cNamedArgs = 0;
	dpPutString.rgvarg = vArgPutString;

	m_hr = m_pMtlbDispApp ->Invoke(m_dispid_PutCharArray, IID_NULL,
		LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dpPutString, NULL,
&m_excep, &m_uArgErr);

	if (FAILED(m_hr))
		goto Exit;

Exit:
	::SysFreeString(bstrName);
	::SysFreeString(bstrString);

	return m_hr;
}

/**
* Get a string from Matlab's engine
*/
HRESULT CMatlabEngine::GetString(LPCTSTR szName, LPTSTR& szString)
{
	static TCHAR tzBuffer[512];
	BSTR bstrName=NULL;
	UINT i;
	std::vector<double> vSize;

	if (!m_bInitialized || !m_pMtlbDispApp)
		return S_FALSE;

	ProcessString(szName, bstrName);


	// preparing argumetns
	VARIANT vArgGetString[2];
	for (i = 0; i < 1; ++i)
		::VariantInit(&vArgGetString[i]);

	V_VT(&vArgGetString[0]) = VT_BSTR;
	V_BSTR(&vArgGetString[0]) = m_bstrWorkspace;
	V_VT(&vArgGetString[1]) = VT_BSTR;
	V_BSTR(&vArgGetString[1]) = bstrName;


	DISPPARAMS dpGetString;
	dpGetString.cArgs = 2;
	dpGetString.cNamedArgs = 0;
	dpGetString.rgvarg = vArgGetString;

	m_hr = m_pMtlbDispApp ->Invoke(m_dispid_GetCharArray, IID_NULL,
		LOCALE_USER_DEFAULT, DISPATCH_METHOD, &dpGetString, &m_vResult,
&m_excep, &m_uArgErr);
```

```
	if (FAILED(m_hr))
		goto Exit;

	// process back to szString...
	{
		_bstr_t s(m_vResult);
		szString=new TCHAR[s.length()+1];
		_tcscpy( szString, (LPCTSTR)s);
	}

Exit:
	::SysFreeString(bstrName);

	return m_hr;

}

#endif
```