UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DESENVOLVIMENTO DE UMA PLATAFORMA DE CONTROLE CONFIGURÁVEL E DE CÓDIGO PORTÁVEL PARA TRANSMISSORES DIGITAIS EM PROCESSOS INDUSTRIAIS

GILSON FONSECA PERES FILHO

Uberlândia
2015

GILSON FONSECA PERES FILHO

DESENVOLVIMENTO DE UMA PLATAFORMA DE CONTROLE CONFIGURÁVEL E DE CÓDIGO PORTÁVEL PARA TRANSMISSORES DIGITAIS EM PROCESSOS INDUSTRIAIS

Dissertação de mestrado submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, como parte dos requisitos necessários para a obtenção do título de mestre em Ciências.

Área de concentração: Controle e automação

Orientador: Prof. Fábio Vincenzi Romualdo da Silva, Dr.

Uberlândia
2015

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

P437d 2015

Peres Filho, Gilson Fonseca, 1977-

Desenvolvimento de uma plataforma de controle configurável e de código portável para transmissores digitais em processos industriais / Gilson Fonseca Peres Filho. - 2015.

77 f. : il.

Orientador: Fábio Vincenzi Romualdo da Silva.

Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.

Inclui bibliografia.

1. Processos de fabricação - Automação - Teses. 2. Automação industrial - Teses. I. Silva, Fábio Vincenzi Romualdo da, 1974- II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 621.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DESENVOLVIMENTO DE UMA PLATAFORMA DE CONTROLE CONFIGURÁVEL E DE CÓDIGO PORTÁVEL PARA TRANSMISSORES DIGITAIS EM PROCESSOS INDUSTRIAIS

GILSON FONSECA PERES FILHO

Dissertação de mestrado submetida à Universidade Federal de Uberlândia, perante a banca de examinadores abaixo, como parte dos requisitos necessários para a obtenção do título de mestre em Ciências.

Banca Examinadora:
Fábio Vincenzi Romualdo da Silva, Dr. – Orientador UFU
Márcio José da Cunha, Dr. – UFU
Marcelo Barros de Almeida, Dr. – UFU
Kleber Lopes Fontoura, Dr. – CEFET/ARAXÁ

Uberlândia
2015

Gilson Fonseca Peres Filho

DESENVOLVIMENTO DE UMA PLATAFORMA DE CONTROLE CONFIGURÁVEL E DE CÓDIGO PORTÁVEL PARA TRANSMISSORES DIGITAIS EM PROCESSOS INDUSTRIAIS

Dissertação de mestrado submetida à Universidade Federal de Uberlândia, como parte dos requisitos necessários para a obtenção do título de mestre em Ciências.

Prof. Fábio Vincenzi Romualdo da Silva, Dr.
Orientador

Prof. Edgard Afonso Lamounier Júnior, Dr.
Coordenador da Pós-Graduação

Uberlândia
2015

DEDICATÓRIA

A minha esposa Ismênia,
a meus pais Gilson e Maria Luisa,
aos meus filhos Marcos e Iasmin ,
aos amigos que contribuiram para este.

AGRADECIMENTOS

À minha amada esposa Ismênia pelo carinho, apoio e paciência. Obrigado por acreditar, compreender e sonhar comigo.

Aos meus pais Gilson e Maria Luisa pelo incentivo aos meus estudos e esforços.

Ao professor Fábio Vincenzi Romualdo da Silva pela disponibilidade, paciência, ajuda, dedicação e ter sido o grande incentivador deste trabalho.

Aos colegas e professores do mestrado que contribuiram de uma forma ou de outra para o meu crescimento acadêmico.

Aos meus amigos professores e engenheiros da área que contribuiram com o sonho de termos um sistema aberto e configurável.

“Felicidade é ter algo o que fazer, ter algo que amar e algo que esperar.”

Aristóteles

RESUMO

PERES FILHO, Gilson. **DESENVOLVIMENTO DE UMA PLATAFORMA DE CONTROLE CONFIGURÁVEL E DE CÓDIGO PORTÁVEL PARA TRANSMISSORES DIGITAIS EM PROCESSOS INDUSTRIAIS**. 2015. 77 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Engenharia Elétrica, Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia. Uberlândia, 2015.

Este trabalho propõe uma plataforma de código livre e aberto com o objetivo de padronizar o processo de configuração e programação de equipamentos industriais. A plataforma é vinculada a uma máquina virtual projetada para sistemas embarcados com baixo custo computacional. Esta plataforma permitirá a fabricantes e pesquisadores criarem interfaces de configuração e programação padronizadas, com a vantagem adicional da implementação de algoritmos de controle personalizados que podem contribuir para otimização de processos produtivos com economia de insumos e energia. Os custos com a implementação de padrões abertos por cada fabricante de equipamentos será reduzido. Os fabricantes poderão se concentrar nas características específicas de seus equipamentos ao invés do padrão e oferecer equipamentos com hardware mais barato e flexível. Finalmente, a formação de pessoal qualificado para manutenção e expansão de processos industriais será facilitada.

Palavras-chave: Padronização. Código livre e aberto. Algoritmos personalizados. Programação . Equipamentos industriais . Redução de custos. Formação técnica. Pesquisa. Desenvolvimento.

ABSTRACT

PERES FILHO, Gilson. **DEVELOPMENT OF AN OPEN SOURCE CONFIGURABLE CONTROL PLATFORM WITH PORTABLE CODE FOR DIGITAL TRANSMITTERS IN INDUSTRIAL PROCESSES**. 2015. 77 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Engenharia Elétrica, Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia. Uberlândia, 2015.

This work proposes a platform with open and free code to standardize the configuration and programming of industrial devices. The platform is bounded to a virtual machine designed to embedded systems with low computational costs. The platform will allow manufactures and researchers build standard configuration and programming interfaces, with the advantage of implementing customized algorithms that may contribute to optimize productive processes reducing energy and supplies. The costs to implement open standards by each manufacture will be minimized. The manufactures could concentrate efforts on specific device behavior instead of standard, offering lower cost and more flexible hardware devices. Finally, technical training for maintenance and expansion of industrial processes will be easier.

Key-words: Standardization. Open and free source code. Custom algorithms. Industrial devices programming. Cost reduction. Specialized training. Research. Development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

CAN	*Control Area Network*
CLP	Controlador lógico programável, PLC em inglês
CRC	*Cyclic Redundancy Check*
FAT	*File Allocation Table* (tabela de alocação de arquivos)
GC	*Garbage Collector*, o coletor de lixo para liberação automática de recursos sem referência (recursos não mais usados)
HDLC	*High-level Data Link Control*, protocolo de comunicação de enlace (segunda camada do modelo OSI)
IP	*Internet Protocol*
JVM	*Java virtual machine*
KiB (**kibi**byte)	**ki**lo**bi**nary: 2^{10} bytes = 1KiB; IEC 60027-2
MiB (**mebi**byte)	**me**ga**bi**nary: 2^{20} bytes = 1MiB, IEC 60027-2
MMU	*Memory management unit,* entidade do processador em que todas as referências à memória passam por ela. Permitindo proteção de acesso, virtualização, isolamento entre processos, arbitragem de barramento, swap em disco, etc.
OSI	*Open Systems Interconnection,* modelo de referência para camadas de protocolo
PC	*Personal computer,* o computador pessoal
PCI	*Protocol Control Information*¸ informação de controle de protocolo. São os dados de controle colocados por uma determinada camada de rede durante o encapsulamento
PDU	*Protocol Data Unit* (PCI+SDU)
PWM	*Pulse Width Modulatiom,* modulação por largura de pulso
RTSJ	*Real-Time Specification for Java,* especificação de tempo real para Java
SDU	*Service data unit,* unidade de serviço de dados. São os dados a serem encapsulados por uma determinada camada de protocolo
SFC	S*equencial Function Chart*
SNMP	*Simple Network Management Protocol,* protocolo para gerenciamento de dispositivos em redes IP
UART	*Universal Asynchronous Receiver/Transmitter*
UDP	*User Datagram Protocol,* protocolo não confiável para envio de pacotes em redes IP

SUMÁRIO

1 INTRODUÇÃO

Em menos de cem anos, os sistemas de automação e controle passaram de sistemas mecânicos e pneumáticos para sistemas elétricos de controle analógicos, posteriormente para sistemas digitais e, finalmente, para as modernas redes de automação e controle digitais com seus dispositivos programáveis.

Atualmente, há um grande número de padrões abertos de tecnologias para automação como Profibus, Fieldbus Foundation (FF), DeviceNet, EtherNet/IP, dentre outros (SAUTER, 2010; SHAHRAEINI, JAVIDI e GHAZIZADEH, 2011; GAJ, JASPERNEITE e FELSER, 2013). Vale salientar que, no ramo de automação industrial, entende-se por "padrões abertos", aqueles que podem ser comprados e implementados por qualquer fabricante de dispositivos industriais.

A aceitação de padrões abertos fez com que palavras como interoperabilidade, intercambialidade e unificado se tornassem comuns entre engenheiros e técnicos de instrumentação. A interoperabilidade entre equipamentos consiste na comunicação entre equipamentos de fabricantes diferentes, desde que utilizem o mesmo padrão. Os dispositivos tornaram-se intercambiáveis porque um dispositivo de um determinado fabricante pode ser substituído por outro de fabricante diferente. E o termo unificado refere-se à possibilidade de se realizar comunicação entre redes com protocolos diferentes por meio de *gateways* (PANTON, TORRISI e BRANDÃO, 2007).

A maioria dos fabricantes aproveitaram as benesses deste novo mundo e começaram a desenvolver equipamentos de acordo com estes padrões. No entanto, para o fabricante, existem alguns custos inerentes aos padrões abertos que são desconhecidos aos usuários. A forma como esses padrões foram projetados é uma fraqueza e um desafio ao mesmo tempo. Isto acontece porque as especificações de alguns padrões são muito caras, extensas, difíceis de serem implementadas e propensas à ambiguidade. Desse modo, existe um grupo de pessoas em cada fundação que é responsável por desenvolver e dar manutenção em seus respectivos padrões. Estes grupos gastam muito tempo escrevendo, revisando, melhorando e frequentemente arbitrando a interpretação do próprio texto para resolverem contestações e problemas. Por outro lado, os grupos de desenvolvimento em cada fábrica, interpretam e testam inúmeras vezes os equipamentos desenvolvidos. Mas antes que um equipamento seja produzido e comercializado, é necessário, para a maioria dos padrões, que seja testado e validado pela respectiva fundação ou organização.

Apesar da padronização de tecnologias, existem particularidades referentes à configuração e programação dos dispositivos industriais. Não existe, portanto, portabilidade de programação e configuração entre os diversos dispositivos industriais construídos por fabricantes diferentes. Esta situação dificulta a formação técnica especializada no sentido de realizar manutenção e expansão de processos industriais e contribui para que o usuário torne-se refém da tecnologia adquirida.

Há um segundo tópico a ser analisado que é a complexidade dos sistemas atuais. Talvez a arquitetura de controle de processos distribuído mais avançada seja a da Fieldbus FOUNDATION TM (FF). Ela promete trazer controle ao campo, prover informações de qualidade dos sinais e um sistema poderoso de alarme e gerenciamento, modos de operação e determinismo de rede.

A FF criou o modelo de bloco funcional para os processos contínuos e padronizou a camada de aplicação. Para tornar a vida dos usuários mais fácil, muitos blocos foram padronizados. Como AI, AO, DI, DO, MDI, MDO, etc (FIELDBUS FOUNDATION, 2008).

Ultimamente, até os transdutores, como os posicionadores de válvula, por exemplo, entraram no processo de padronização. Isto foi muito bom do ponto de vista do usuário, mas colocou mais pressão nas ferramentas de teste como ITS (*Interoperability Test Systems*), que são usadas no ITK (*Interoperability Test Kit*) (VREELAND e MITSCHKE, 1999). Esta ferramenta realiza vários testes, porém, não é capaz de testar tudo, devido ao grande número de combinações. Em cada nova versão do ITK, os fabricantes são surpreendidos por um novo teste que não era feito anteriormente ou que foi demandado devido a novas características na especificação. Geralmente, um ou dois meses são necessários para resolver todos os problemas no firmware do equipamento ou no software do ITS.

Ademais, embora os blocos funcionais flexíveis no padrão FF existam para integrar a lógica Ladder ou qualquer outra linguagem proprietária do fabricante, permitindo alguma customização de algoritmos, eles são limitados em capacidade e velocidade para transmissores mais simples. Sendo mais capazes em controladores mais poderosos como *linking devices* e CLPs.

A partir do exposto, verifica-se o grande esforço realizado por parte das organizações e fabricantes no sentido de aperfeiçoarem os equipamentos industriais. Neste sentido, este trabalho propõe uma nova abordagem de hardware e software com o propósito de obterem-se os seguintes benefícios: 1) tornar a programação e configuração de dispositivos industriais portáveis; 2) reduzir o esforço de interpretação, implementação e testes de padrões por parte

dos fabricantes; 3) diminuir o custo de hardware dos equipamentos; 4) possibilitar a implementação de controles diversos.

Para atingir estas metas, a Plataforma 3C (Controlador de Código Comum) proposta pode ser definida como controladores de processos com seus próprios sensores, atuadores e qualquer outra interface física necessária. Por controladores de processo entende-se uma entidade de processamento (3C) que permitiria executar códigos arbitrários.

Há inúmeros softwares de sucesso de código fonte aberto e desenvolvimento compartilhado rodando atualmente, com crescente contribuição de vários programadores em todo mundo. Os softwares abertos mais conhecidos atualmente são: Linux (*kernel* e coletânea, isto é, coletânea de outros softwares que compõem o sistema como utilitários e compiladores GNU), browser Firefox, servidor Apache e a suíte Open Office (editor, planilha, apresentação e base de dados).

Assim, uma solução de código aberto reduziria muito os problemas de interoperabilidade e interpretação equivocada de normas, permitindo que o esforço de desenvolvimento se concentre na parte específica do equipamento ao invés do padrão (MONTPELIER OPEN SOURCE, 2015; BROWNING, 2010; KANTER, *et al.*, 2012).

Outro apelo ao software aberto seria a contribuição de entidades educacionais. Para a maioria delas é praticamente impossível contribuir com os padrões que são fechados e arcar com os custos de anuidade requeridas dos membros.

Todos os recursos específicos requeridos por uma dada aplicação devem ser carregados dinamicamente no equipamento. Um conjunto padrão de recursos, além do ambiente de execução deve ser padronizado, como funções matemáticas, acesso de E/S, temporizadores e interfaces de protocolo.

Quaisquer recursos, como alarmes, qualidade de sinal, registro de histórico, conjunto de variáveis (*views*), ligações lógicas entre equipamentos (*links*), etc, fica a cargo da ferramenta de configuração. Ela é responsável por gerar um código comum a partir de qualquer linguagem de programação suportada como Ladder, diagrama de blocos, texto estruturado, SFC (*sequencial function chart*) dentre outras linguagens. Tal ferramenta permite ainda que o usuário edite o código gerado antes de aplicá-lo. Deste modo, um novo mercado para ferramentas de configuração intercambiáveis e capacidades avançadas também poderá ser criado. Vale salientar, no entanto, que esta proposta não tem a pretensão de forçar a substituição da tecnologia já presente nos dispositivos industriais. Ela visa contribuir para o desenvolvimento da área em questão criando uma nova abordagem, mais simples, barata e flexível.

Esse documento está dividido da seguinte forma: no Capítulo 2 é apresentada uma comparação entre a Plataforma 3C, o sistema de referência existente e outros sistemas. No Capítulo 3 é apresentada a implementação bem como as demais ferramentas desenvolvidas. No Capítulo 4 são mostrados os testes de desempenho e uma planta controlada por este sistema. Enfim, o Capítulo 5 traz as conclusões e sugestões para trabalhos futuros.

2 COMPARAÇÃO ENTRE A PLATAFORMA 3C, O SISTEMA DE REFERÊNCIA FOUNDATION FIELDBUS™ E OUTROS SISTEMAS

2.1 Introdução

Este capítulo aborda as características que se deseja na Plataforma 3C, como: flexibilidade, padronização, interoperabilidade, inovação, viabilidade técnica e viabilidade financeira. Faz, ainda, alguns paralelos com outros sistemas e o sistema Foundation Fieldbus™, que será adotado como referência por ser atualmente uma das arquiteturas de controle de processos distribuídos mais avançadas.

A seguir, são apresentadas e comparadas com o sistema de referência e outros sistemas as principais características almejadas na plataforma proposta.

2.2 Flexibilidade e personalização

Flexibilidade e personalização é a capacidade do usuário do sistema poder criar seus próprios algoritmos de controle. No sistema de referência esta característica é limitada pela capacidade de combinação de blocos funcionais ou blocos flexíveis (lógica Ladder). Em outros sistemas, em geral, é obtido através de lógica Ladder ou texto estruturado e blocos de funções previamente definidos. Normalmente o processamento de lógica é abundante em CLPs, mas limitado ou inexistente para aplicações que necessitam de processamento matemático, estruturas de dados e recursão.

A maioria dos dispositivos industriais do sistema de referência utiliza blocos com algoritmos fixos. Estes algoritmos fixos, na sua maioria, são os clássicos como PID, média, lead-lag, raiz, temporizadores, totalizadores e contadores. No entanto, várias soluções interessantes podem ser obtidas com outras estratégias como lógica Fuzzy, redes neurais artificiais, controle multivariável e até mesmo algoritmos de controle mais específicos para um dado problema, como os gerados por programação genética (PILAT e OPPACHER,

2000). As ferramentas para estas estratégias são escassas em sistemas industriais, disponíveis apenas em centros de pesquisa devido ao seu custo, caráter acadêmico ou com robustez insuficiente para a indústria. Neste sentido, dar flexibilidade a um sistema com algoritmos fixos, é uma tarefa difícil, quando não impossível, porque só é permitida alguma combinação entre os mesmos. Na Plataforma 3C proposta, ao invés de definir-se um conjunto de blocos com algoritmos fixos, permite-se a definição dos algoritmos destes blocos. Ou seja, fragmentam-se os blocos em funções mais simples (operações matemáticas, funções lógicas, estruturas de controle de execução, etc.) que possam gerar qualquer algoritmo.

Assim, a sintaxe da estratégia de controle, na interface com o usuário, tanto pode se dar diretamente por *scripts* do usuário quanto por *scripts* gerados por uma ferramenta de configuração avançada, no formato de entrada mais adequado como linguagem Ladder, ligação de blocos funcionais, máquinas de estado, etc.

Esta abordagem de fragmentar os blocos em primitivas mais simples aproxima-se de um microprocessador que possui instruções simples que possibilitam a montagem de códigos mais complexos. Porém, gerar código nativo para cada plataforma inviabiliza a portabilidade de programação e configuração entre os diversos dispositivos. Deste modo, uma máquina virtual (um software) que simula um processador (real ou fictício) resolveria a questão da portabilidade, satisfaz a questão da flexibilidade e possibilita a execução de código seguro (GOLDBERG, *et al.*, 1996; PREVELAKIS e SPINELLIS, 2001).

2.3 Padronização

A padronização permite o reuso de algoritmos e configurações entre diferentes dispositivos. No sistema de referência podem-se aproveitar algumas configurações entre equipamentos que executem blocos funcionais iguais ou equivalentes (em geral apenas do mesmo fabricante). No entanto, o reuso de configurações não é suportado entre ferramentas de configuração de fabricantes diferentes. Assim, um programa **L** escrito em linguagem Ladder utilizado em um equipamento de um fabricante **A** não pode ser copiado e utilizado em um equipamento de um fabricante **B**, sem que o usuário entre novamente com **L** na ferramenta de configuração de **B** e o readéque, pois blocos como temporizadores, contadores, movimentação, dentre outros, não são padronizados. Além disso, há vários casos de incompatibilidade entre modelos diferentes do mesmo fabricante.

Não existe, portanto, portabilidade de programação e configuração entre os diversos dispositivos industriais construídos por fabricantes diferentes. Esta situação dificulta a formação técnica especializada no sentido de realizar manutenção e expansão de processos industriais.

A padronização seria muito bem vinda aos usuários do sistema que poderiam reutilizar programas e configurações entre diferentes fabricantes. Mesmo sem o uso de uma máquina virtual, proposta anteriormente, isto é perfeitamente possível através do uso de linguagens intermediárias padronizadas, que gerem o mesmo resultado através de algum tradutor (compilador ou interpretador). As linguagens interpretadas como Python, PERL e Ruby ou as compiladas como Java, C e PASCAL poderiam ser usadas como intermediárias se fosse padronizada a biblioteca (API) de acesso ao sistema.

Na Plataforma 3C o código "*script*" de um dispositivo é portável para outro dispositivo, inclusive entre fabricantes distintos. Ressalva-se que para a substituição de um dispositivo por outro é necessário que as interfaces de entrada e saída usadas sejam compatíveis.

2.4 Especificação, implementação e interoperabilidade

"Havendo olhos suficientes, todos os erros são óbvios."

Eric Steven Raymond

Para atingir as características de interoperabilidade entre os equipamentos, o sistema de referência criou uma extensa especificação. São aproximadamente 1700 páginas na compilação das versões de seus volumes no ano de 2009, relacionadas à especificação H1 (excluindo-se a versão HSE – *High Speed Ethernet*), que definem as camadas superiores (SM, FAS, FMS, FBAP) dos equipamentos que usam o meio físico e *Data Link Layer* definidos pela IEC 61158-2. Esta última, sozinha, possui 392 páginas além daquelas 1700.

Uma especificação formal muito extensa é difícil de ser completamente validada e de não apresentar inconsistências. Por isso, a abordagem proposta possui uma implementação escrita na linguagem C que, embora não isenta de erros, pode ser largamente "portada" e testada em diversos ambientes computacionais adequados. Um caso de ausência de especificação formal é a Máquina Virtual Lua que não possui especificação do seu conjunto de instruções, mas é implementada em código C juntamente com o seu compilador (JUNG e BROWN, 2007).

Na Plataforma 3C, a padronização da linguagem permite o reaproveitamento de configurações e a interoperabilidade permite a comunicação entre dispositivos de diferentes fabricantes de forma a operarem em conjunto.

Como já foi exposto, é difícil e caro tanto criar quanto implementar uma especificação formal muito extensa (HANNA, 2007). Implementá-la por diferentes equipes, em diferentes empresas só aumenta o custo e o esforço desnecessariamente. A alternativa mais imediata é o desenvolvimento de um código aberto, portável e comum.

Para a defesa desta linha pode-se usar o seguinte raciocínio: a maioria das metodologias de definição de software consiste em um documento com os requisitos gerais, sem entrar em muitos detalhes (salvo em alguns casos de uso mais específicos). Desta forma, softwares desenvolvidos por equipes diferentes, a partir dos mesmos requisitos, podem chegar a resultados similares, contudo, cada qual com suas particularidades, fruto do detalhamento inferido por cada equipe. Todavia, se estas duas equipes se unirem no desenvolvimento daquele software, além do ganho de escala, haverá um único software resultante.

Extrapolando este raciocínio para o que temos hoje com os softwares livres e desenvolvidos em conjunto pela comunidade como: a coleção GNU, servidor Apache, Firefox e outros, observa-se que existe relativa harmonia na implementação e, portanto, garante-se a interoperabilidade (resguardados os devidos cuidados de portabilidade para as diversas arquiteturas de processadores).

Por isso, uma das metas futuras de desenvolvimento da Plataforma 3C é que ela seja de código comum e livre, pois, como ocorre com muitos softwares livres, consegue-se interoperabilidade e qualidade através da contribuição contínua de desenvolvedores ao redor do mundo que fazem parte da comunidade relacionada à área.

2.5 Simplicidade e custos

"Everything should be made as simple as possible, but not simpler."
Albert Einstein

Dispositivos de controle complexos com muitos recursos pré-definidos podem ser caros dada a gama de recursos de hardware requerida como processamento, memória de programa (ROM ou FLASH), armazenamento (FLASH, EEPROM ou FRAM) e trabalho (RAM). Um sistema canônico que permita o reaproveitamento desses recursos conforme a necessidade específica da aplicação pode trazer significativa redução de custos de hardware e eficiência.

Neste sentido, a Plataforma 3C apresenta um número reduzido de funções e recursos pré-definidos. Esta estratégia resulta em redução do custo total do hardware (sistema mínimo de processamento e memória) e contribui para que o hardware esteja de acordo com as limitações de consumo e armazenamento de energia estabelecida por normas como as de áreas classificadas (NBR IEC 60079-14, 2006) ou disponibilidade de bateria e outras fontes limitadas.

Além disso, a Plataforma 3C proposta possuirá um sistema mínimo de processamento e memória capaz de executar algoritmos do usuário, reduzindo, desta forma, recursos pré-definidos de hardware e software necessários na implementação de uma especificação extensa.

2.6 Especificação da Plataforma 3C

Com base na reflexão anterior, a solução adotada é especificada a seguir:

- Código aberto e portável;
- Máquina virtual;
- Linguagem comum;
- Hardware eficiente e com baixo custo.

2.6.1 Máquina virtual

O uso de uma máquina virtual (MV) *(VM,* do inglês *Virtual Machine)* adequada é um dos principais requisitos para que a plataforma proposta possua flexibilidade de algoritmos e portabilidade entre diferentes plataformas. Neste sentido, uma opção seria desenvolver uma máquina virtual adequada ao problema ou fazer uso de uma máquina virtual existente.

Implementar uma máquina virtual pode levar a uma solução ótima, mas requer trabalho árduo e de longa duração (BOLZ e RIGO, 2007). Considerando que o foco principal deste trabalho não consiste no desenvolvimento de uma MV, mas propor uma nova abordagem, o uso de uma máquina virtual já existente, no momento, é a opção mais adequada.

2.6.1.1 Seleção da máquina virtual

Existem diversas máquinas virtuais disponíveis, no entanto, pelos requisitos anteriores ela deve apresentar as seguintes características:

- Leve: executar eficientemente em hardware simples e de baixo custo;
- Aberta: ter seu código fonte aberto permitindo adaptações e melhorias;
- Amigável: possuir um compilador ou interpretador com linguagem de alto nível e de fácil aprendizado para os usuários (definir uma linguagem ou gerar um compilador para a máquina extrapola o escopo deste trabalho);

Com base nos requisitos anteriores foi compilada a lista de MVs da Tabela 1.

Tabela 1 - Máquinas virtuais mais simples encontradas

Nome	Arquitetura	ROM	RAM	Outros
Kaffe	**JVM**	100KiB para processadores de 32 bits (interpreter, GC, native code)	Depende da aplicação, ideal > 1MiB	Requer, indiretamente, MMU (Kaffe FAQ, 2007)
JamVM	**JVM**	de 80KiB a 110KiB (interpreter, GC, native code)	Não informado	Requer subsistema unix (Linux, FreeBSD,etc)

Nome	Arquitetura	ROM	RAM	Outros
Mika VM	**JVM**	Menos de 2MiB (interpreter, GC, native code, class libraries)	Não informado	RTSJ
Mysaifu JVM	**JVM**	Não informado	Não informado	Requer Windows Mobile OS
Java In The Small	**JVM**	>150KiB (interpretador, GC e código nativo de suport)	>120KiB para aplicação bem simples	Aberto, mas não gratuito (MARQUET, *et al.*, 2004). Parece descontinuado.
Squawk	**JVM – subconjunto modificado**	Alvo 160KiB	8KiB + 32KiB (NVM in EEPROM)	Há um tradutor intermediário. (SHAYLOR, SIMON e BUSH, 2003)
Squeak vm	**Específico: adequado a Small Talk**	200KiB (estimado, sem GUI)	Não informado	Aparentemente, requer OS (Windows, Linux e subsistema)
IVM	**Forth Machine**	1KiB	Só depende das pilhas (trabalho e retorno) do algoritmo.	Até a memória pode vir de uma E/S. (IV Machine or Forth Machine, 2012)
PyMite VM	**Python 2.6 (subconjunto)**	55KiB	8KiB (depende do script)	Não inclui compilador (PyMite VM, 2014)
Micro Python	**Python 3.0 (subconjunto)**	60KiB	Depende do script, 4KiB para "HelloWorld". Placa de teste tem 128KiB	Requer geração de código nativo para Cortex-M3 (Micro Python, 2014)
Lua	**Lua VM**	40KiB (sem compilador e biblioteca C padrão)	Depende do script. Recomendável > 25KiB	(A Linguagem de Programação Lua, 2014)

De acordo com as características desejadas, uma das máquinas virtuais mais comumente encontradas e que poderia ser utilizada é a JVM *(Java Virtual Machine)*. Com

relação à linguagem de alto nível Java, ela possui a vantagem de ser sempre suportada pelo compilador padrão javac, entretanto a linguagem Java apresenta uma forma muito complexa de definição para usuários com conhecimentos mais básicos em programação. Por isso, uma linguagem semelhante a um texto estruturado com menor apelo a "*tipagem rígida*" teria sua difusão facilitada. Linguagens como: Eiffel, Clojure, Groovy, Scala, Ruby e Python também possuem compiladores específicos que geram código para a JVM. No entanto, os requisitos de memória da JVM são relativamente altos, quando não requerem características específicas de hardware como MMU (*Memory Management Unit*) ou de sistema operacional como Windows, Windows Mobile, Unix, etc.

Em função da disponibilidade de linguagens de alto nível, pode-se desconsiderar a linguagem Forth (foi colocada na tabela apenas como referência para baixo requisito de recursos) e utilizar um subconjunto da linguagem Python 2.6 suportada pela máquina PyMite ou um subconjunto de Python 3.0 suportado por MicroPython – MicroPython é um trabalho promissor, mas o segundo nível de sua otimização está associado à geração de código nativo, o que torna a portabilidade mais complexa. Python possui uma sintaxe poderosa para trabalhar com listas, objetos e funções. No entanto, talvez ela seja relativamente complexa para usuários acostumados a linguagens mais simples.

Por fim, Lua, a última arquitetura apresentada na tabela, possui linguagem portável, eficiente como apresentado por Ierusalimschy *et al*. (2001) e "embarcável". Foi desenvolvida pensando-se na portabilidade para outras arquiteturas e fácil integração com código C. Todas as partes do subsistema Lua foram escritos em C (máquina virtual, compilador, bibliotecas de acesso, *garbage collector*, etc.). Diversas características da linguagem Lua foram motivadas pela indústria e por requisitos de usuários (IERUSALIMSCHY, FIGUEIREDO e CELES, 2005). Lua é bastante eficiente e econômica em uso de RAM e ROM, também foi amplamente usada como linguagem de extensão em alguns jogos: Angry Birds, World of Warcraft, Escape from Monkey Island e em softwares famosos como Adobe Photoshop Lightroom, Apache HTTP Server, Firefox e MediaWiki. Assim, devido ao conjunto das características apresentadas, a máquina Lua foi considerada a mais adequada para os objetivos deste trabalho.

2.6.1.2 Portabilidade

Um dos principais objetivos deste, trabalho é permitir que o algoritmo de controle do usuário desenvolvido para um dispositivo, possa ser utilizado em outros. Assim, a escolha de

uma linguagem padronizada e bem difundida, permitirá o reuso de código entre diferentes plataformas de fabricantes diferentes.

A linguagem Lua, que acompanha a Máquina Virtual Lua, é bastante simples e amigável tanto para o aprendizado quanto para seu uso na codificação de algoritmos de controle personalizados. Deste modo, Lua será a linguagem adotada na Plataforma 3C.

Na Tabela 2 são apresentados alguns exemplos de programas Lua.

Tabela 2 – Exemplos de programas Lua

O clássico "Hello World"	`print('Hello World!')`

Fatorial de 5	`function factorial(n)`
	`  local x = 1`
	`  for i = 2,n do`
	`  x = x * i`
	`  end`
	`  return x`
	`end`
	`print(factorial(5))`

Algoritmo PID série (SHAW, 2006)	`local function runPid(pid)`
	`  error = 0`
	`  pvD=0`
	`  outTmp=0`
	`  if pid.en==false then`
	`  pid.lpv = pid.pv`
	`  pid.fbk = 0`
	`  pid.out = pid.lout;`
	`  else`
	`  pvD = pid.pv + (pid.pv - pid.lpv)*pid.td*pid.rptmin;`
	`  pid.lpv = pid.pv`
	`  error = pid.sp - pvD`
	`  pid.lerror = error`
	`--  error = -error; -- inverter sinal para ação direta`
	`  outTmp = error * pid.k + pid.fbk;`
	`  if (outTmp > pid.max) then`
	`  outTmp = pid.max`
	`  else`
	`  if (outTmp < pid.min) then`
	`  outTmp = pid.min`
	`  end`
	`  end`
	`  pid.out = outTmp;`
	`  pid.lout = outTmp;`
	`  pid.fbk = pid.fbk + (outTmp-pid.fbk)*pid.resetRate/pid.rptmin;`
	`  end`
	`end`

2.7 Conclusão

Este capítulo abordou as principais características que se deseja na Plataforma 3C, como: flexibilidade, padronização, interoperabilidade, simplicidade, inovação no modelo de desenvolvimento, viabilidade técnica e redução de custo em relação ao hardware necessário através da minimização de recursos pré-definidos. Além disso, fez um paralelo das características que a Plataforma 3C deve possuir com aquelas que o sistema de referência e outros apresentam.

A partir das características desejadas, determinou-se que a máquina virtual Lua e sua linguagem de programação possibilitam a implementação dos recursos e características pretendidos na plataforma proposta.

3 IMPLEMENTAÇÃO

3.1 Introdução

O principal software desenvolvido neste trabalhado foi chamado **Controlador de Código Comum** (3C), ele cria todo o ambiente necessário à máquina virtual selecionada, dando suporte de E/S (entrada e saída, *I/O*), controle da máquina virtual – execução e carregamento, temporização, rede de comunicação e sistema operacional. Este conjunto mais as camadas de portabilidade criam a Plataforma 3C.

3.2 Arquitetura da plataforma proposta

A arquitetura da plataforma foi modelada em blocos, cada um disponibilizando um dado serviço. A Figura 1 ilustra os blocos implementados (ressaltando que o bloco LuaVM não foi desenvolvido neste trabalho e será detalhado posteriormente). Os programas escritos na linguagem Lua para a Plataforma 3C serão chamados "*scripts*" no restante do texto.

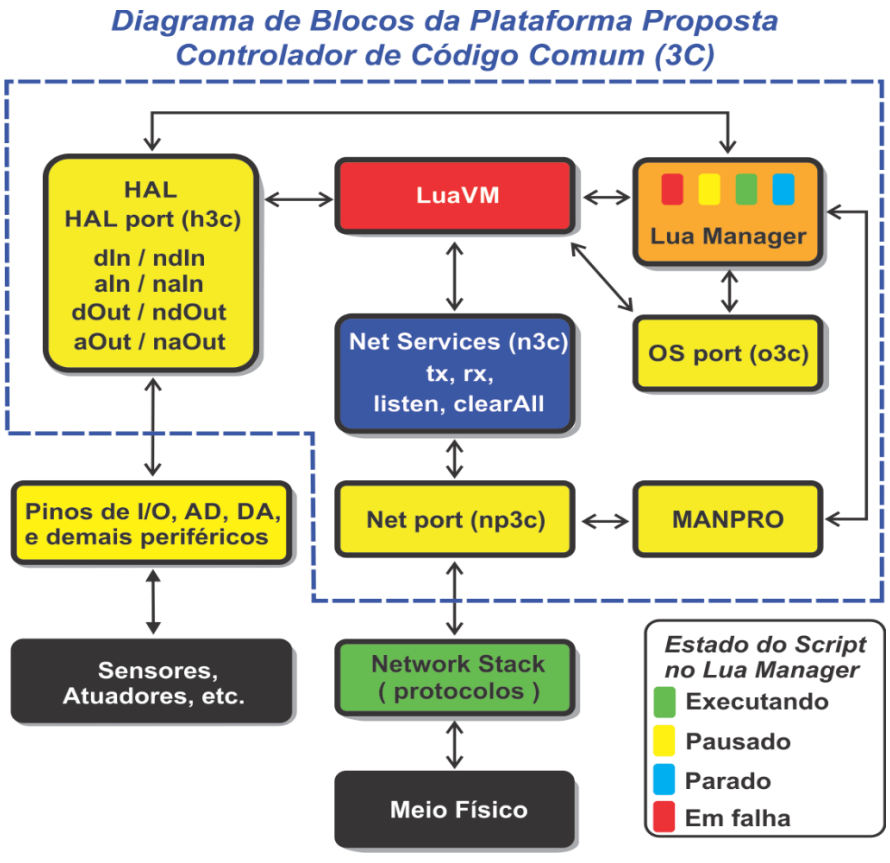


Figura 1 – Diagrama de blocos da Plataforma 3C

3.3 A máquina virtual Lua (LuaVM)

Lua (linguagem e máquina virtual) possui um desenvolvimento aberto coordenado pelo grupo da PUC-RJ. Durante o desenvolvimento deste trabalho a versão estável utilizada foi a 5.2 (liberada em 16/12/2011 e suportada pelo projeto eLua), sendo a última *release* a 5.2.3 de 7 de dezembro de 2013. A versão de desenvolvimento foi a 5.3, ainda em teste (LUA.ORG, 2014). Esta última trás algumas características bem interessantes para sistemas embarcados como a possibilidade do tipo "*number*" de Lua ser inteiro ou ponto flutuante de acordo com atribuição inicial – desde o princípio o tipo "*number*" foi um ponto flutuante[1] em Lua como apresentado em Ierusalimschy *et al*. (2001). Para sistemas embarcados com processadores sem FPU (*floating point unit),* o ponto flutuante é um tipo com processamento oneroso.

A máquina virtual Lua foi desenvolvida em linguagem ANSI C o que facilita sua portabilidade para diversas plataformas. O código da máquina é relativamente compacto, para sua versão Linux x86 ocupa menos de 150KiB de ROM com todas as bibliotecas Lua e compilador integrado, o núcleo sem as bibliotecas consome menos de 100KiB. Para a plataforma Cortex-M3 usada neste trabalho consome também menos de 100KiB de ROM. Várias funções da biblioteca padrão ANSI C são necessárias. Neste contexto embarcado destaca-se uma bastante crítica: **realloc()**. Trata-se de uma função "perigosa" para plataformas com poucos recursos por fazer alocação e realocação de blocos de memória, e desta forma propiciar, dependendo do algoritmo usado na implementação da mesma, a fragmentação de memória.

Uma característica interessante de Lua é a pré-compilação do código fonte Lua (o "*script")* para um código intermediário ("*byte code*") e só então a máquina virtual Lua inicia a sua execução, desta forma o código fonte não é interpretado diretamente. Este compilador, de apenas uma passagem, consome uma quantidade significativa de memória RAM, dependendo do *script,* que é agravado posteriormente pelo uso de memória das tabelas das bibliotecas de suporte de Lua em RAM (matemática, bit e dos módulos 3C desenvolvidos neste trabalho). Felizmente, para minimizar o consumo daquelas tabelas, um projeto conhecido como eLua (*embedded* Lua) (eLua, 2014) já se encontra relativamente maduro e possui vários *patches*

[1] Nos arquivos de configuração de Lua versão 5.2 é possível alterar a definição do tipo number para que seja um inteiro, mas isto remove completamente a capacidade de usar o ponto flutuante.

(alterações para um dado fim) adequados para reduzir os recursos de memória necessários. O objetivo daquele projeto é bem audacioso. Ele conta com um console ou *shell* Lua disponível via porta serial, sistema de arquivos FAT, drivers para cartões de memória, protocolo XMODEM, RPC *(remote procedure call), embedded text editor*, *embedded help*, *embedded HTTP WEB Server*, etc. Existem *ports* (adaptações necessárias para uma dada plataforma) para diversas plataformas. O projeto eLua é muito amplo, por isso, foram usados alguns daqueles *patches* para economia de recursos (redução de 25KiB para 2KiB de RAM só para a execução de "hello World", por exemplo) mas sem a pretensão daquele shell ou seus recursos mais avançados.

Algumas interfaces fundamentais para o que o *script* tenha referência de tempo e seja capaz de acessar o ambiente externo foram padronizadas: temporização, E/S (analógica e digital) e rede (comunicação). Estas interfaces foram implementadas da forma tradicional de Lua para extensão: módulos em linguagem C (*the C API*) (IERUSALIMSCHY, 2013). Visando a portabilidade para diferentes arquiteturas, o uso de diferentes protocolos e características específicas de cada equipamento, três outros módulos dedicados especificamente à portabilidade foram definidos: *HAL port* (h3c), *Net port* (np3c) e *OS port* (o3c). Notar que os serviços de temporização e E/S foram definidos diretamente nas camadas de portabilidade devido a sua simplicidade. Por outro lado a camada de rede requer um tratamento especial e foi dividida em duas partes: serviços comuns (portáveis, n3c) e serviços que requerem portabilidade (np3c). Algumas bibliotecas padrões de Lua implementadas através da C API foram suprimidas como acesso a sistemas de arquivo e manipulação de data e hora por não serem necessárias para esta aplicação.

3.4 Lua Manager

Este módulo foi implementado de modo a atuar na máquina virtual Lua e controlar sua execução. Ele permite atualizar e ler o *script* de controle, iniciar, pausar e parar a execução da máquina e, por consequência, o *script*. Há uma ligação deste módulo com a camada de rede através do protocolo MANPRO de forma a permitir sua comunicação sem a dependência da máquina Lua e seu *script*. O Lua Manager é controlado remotamente através de MIB (*Management Information Base*) semelhante à arquitetura do SNMP (*Simple Network Management Protocol*). Possui três objetos que podem ser lidos e/ou escritos. São eles: "*status*" (apenas leitura, pode estar em "Running", "Paused", "Stopped" e "Fault"), "*request*"

(pode-se escrever “Run”, “Stop” e “Pause”) e “*script*” (usado para escrever e ler o *script* do usuário).

O gerenciamento da máquina virtual Lua foi feito através de sua biblioteca de depuração. Foram usados *hooks* que são mecanismos que permitem registrar uma função para ser chamada em eventos específicos programados (IERUSALIMSCHY, 2013). O *hook* usado foi do tipo *count* que ocorre após um programado número de instruções.

Internamente, o Lua Manager cria uma tarefa exclusiva para a execução da máquina virtual através do módulo de portabilidade o3c (pilha da tarefa em RAM de 2KiB). Esta tarefa fica suspensa (aguardando um sinal, implementado como semáforo pelo o3c) até que um comando externo permita sua execução. Os principais passos desta tarefa são:

1 – Criar um objeto luaState;

2 – Inicializar variáveis de estado anterior e controle;

3 – Abrir bibliotecas e registrar o *hook* do tipo *count;*

4 – Aguardar o sinal de execução (semáforo);

5 – Compilar o último *script* carregado;

6 – Se houver sucesso na compilação, iniciar a execução;

7 – Se falhar a compilação, guardar o estado anterior como falha (*fault)*;

8 – Liberar o objeto luaState;

9 – Voltar ao passo 1.

Detalhe do código da tarefa:

```
bool fault = false;
for(;;)
{
  lua_State* L = luaL_newstate();   // alocando lua_State
  men_lmng3cState = (fault ? EL3S_FAULT: EL3S_STOPPED);
  men_lmng3cReq = EL3R_NONE;
  luaL_openlibs(L);
  lua_sethook(L, AbortRoutine, LUA_MASKCOUNT, 1000); // registro do hook
  O3C_WaitSignal();
  if (luaL_loadstring(L, (const char*)mau8_codeBuffer)==0) // compilando
  {
    men_lmng3cState = EL3S_RUNNING;
    lua_pcall(L, 0, LUA_MULTRET, 0); // executando
    fault = false;
  }
  else
  {
    fault = true;
  }
  lua_close(L);
}
```

Foi feito um registro do uso de memória RAM pela função realloc() e pela função de alocação de *heap* sbrk(), para o *script* do controlador PID (Apêndice A), usado no experimento de controle de nível do capítulo 4. Naquelas duas chamadas foi colocado um *break point* e anotado, manualmente, em uma planilha cada alocação bem como as liberações nas chamadas de free(). Internamente o algoritmo da função realloc() chama a função de sistema sbrk() (Sbrk, 2015). O registro da quantidade exata de RAM demandada pela máquina virtual pode ser feito na chamada de realloc(), no entanto, como o algoritmo de realloc() da biblioteca padrão (nesta implementação de desenvolvimento: newlib 2.0.0, *toolchain* Sourcery CodeBench Lite 2013.11-24) reserva blocos de memória no *heap* conforme: tamanho do bloco, controle, fragmentação e eficiência registrou-se também as alocações por sbrk() por ser o requisito real desta implementação. A Figura 2 apresenta o registro das alocações.

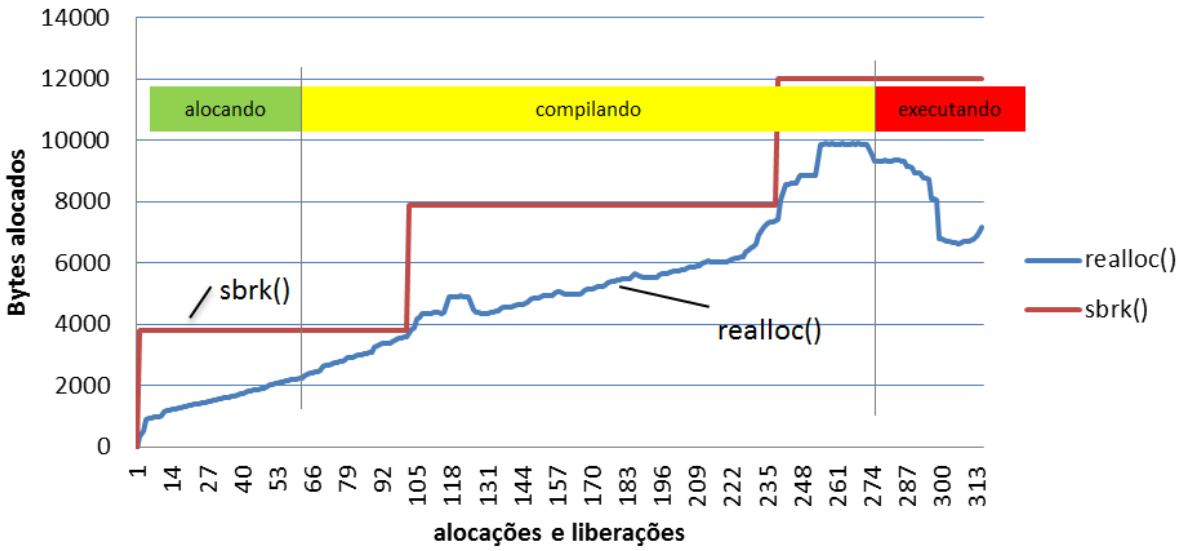


Figura 2 – Gráfico das alocações feitas pela máquina virtual

No detalhe do código da tarefa os comentários "//" fazem referência aos destaques no gráfico de alocações. A execução do "*byte code"* pela máquina virtual feita, em seguida, em "lua_pcall()" não demandou nenhuma alocação adicional por parte de sbrk() mas houve chamadas de realloc() no início da execução, que reutilizaram os blocos já reservados por sbrk() durante a compilação. No entanto, descobriu-se que se a função *print*, usada no *script* para mostrar os valores internos das variáveis do algoritmo PID *"print(pid.sp .. ' '.. pid.pv .. ' ' .. pid.out .. ' ' .. pid.k .. ' ' .. pid.resetRate .. '\r')"* tiver os valores dos parâmetros alterados isto demanda sucessivas novas alocações (chamadas de realloc() e free()). Para converter o *number* em *string* parece haver algum mecanismo de otimização que

só requer nova realocação quando o valor muda. Assim, é recomendável que o script não faça esse tipo de conversão em valores que mudem frequentemente para aumentar a velocidade e evitar a entrada do **GC**.

Nesta implementação, apenas a máquina virtual faz chamadas de alocação da função realloc() ou outras que terminam com a chamada de sbrk(). A função de alocação usada pelo sistema operacional é específica do mesmo e usa outro bloco de memória reservada, desta forma não interferiu nos dados obtidos.

Note que durante a execução da máquina, não havendo falhas, a tarefa fica executando a máquina virtual na chamada "lua_pcall()". A cada *count* instruções do *script* executadas o *hook* é chamado. Dentro desta chamada, com o contexto passado, é possível alterar o luaState. No retorno do *hook* a máquina virtual pode identificar alguma mudança e cancelar a execução retornando ao passo 8 da tarefa. Detalhe do código do *hook* usado:

```
static void AbortRoutine(lua_State *L, lua_Debug *ar)
{
  if (ar->event == LUA_HOOKCOUNT)
  {
    switch (men_lmng3cReq)
    {
      case EL3R_NONE:
      case EL3R_RUN:
        break;

      case EL3R_STOP:
        lua_pushstring(L, "Stop!");
        lua_error(L);
        men_lmng3cState = EL3S_STOPPED;
        break;

      case EL3R_PAUSE:
        men_lmng3cState = EL3S_PAUSED;

        do {

          O3C_WaitSignal(); // wait here till resume

        } while (men_lmng3cReq == EL3R_PAUSE);

        if (men_lmng3cReq == EL3R_STOP) // it may have changed
        {
          lua_pushstring(L, "Stop!");
          lua_error(L);
          men_lmng3cState = EL3S_STOPPED;
        }
        else
        {
          men_lmng3cState = EL3S_RUNNING;
        }
        break;
    }
    men_lmng3cReq = EL3R_NONE;
  }
}
```

As variáveis de controle usadas no *hook* são atualizadas por meio da MIB através de escritas remotas vindas do MANPRO. Dependendo do suporte do sistema operacional usado seria possível cancelar a tarefa de execução da máquina virtual, mas na maioria dos sistemas operacionais esta opção é crítica porque alguns objetos e recursos podem não ser liberados e ficarem bloqueados indefinidamente. Esta solução cooperativa alia ainda outra vantagem: a tarefa é criada uma única vez e reutilizada quantas vezes for necessário, eliminando problemas de fragmentação. O código desenvolvido neste trabalho seguiu uma recomendação para sistemas embarcados de se evitar fazer alocações e liberações dinâmicas de recursos: todas as alocações são feitas uma única vez na inicialização e reutilizadas na forma de "*pool*" (banco/conjunto) de recursos, exceção feita pela LuaVM – não desenvolvida neste trabalho.

3.5 HAL (*Hardware Abstraction Layer*)

Esta camada provê recursos para que o Lua Manager monitore a E/S, assuma o controle das saídas ou permita o acesso pela LuaVM. A princípio, por simplicidade, quatro variáveis e quatro funções foram implementadas (visíveis ao *script*):

- **h3c.naIn** – variável que contém o número de entradas analógicas disponíveis no hardware;
- **h3c.naOut** – variável que contém o número de saídas analógicas disponíveis no hardware;
- **h3c.ndIn** – variável que contém o número de entradas digitais disponíveis no hardware;
- **h3c.ndOut** – variável que contém o número de saídas digitais disponíveis no hardware;
- **h3c.aIn(i)** – lê uma entrada analógica. Sendo **i** [0...**h3c.naIn-1**] a entrada;
- **h3c.aOut(i, v)** – escreve em uma saída analógica. Sendo **i** [0...**h3c.naOut-1**] a saída e **v** o valor;
- **h3c.dIn(i)** – lê uma entrada digital. Sendo **i** [0...**h3c.ndIn-1**] a entrada;
- **h3c.dOut(i, v)** – escreve em uma saída digital. Sendo **i** [0...**h3c.ndOut-1**] a saída e **v** o valor.

Exemplo da implementação da função "*h3c.dIn*" (lendo os botões do *kit* de desenvolvimento Stellaris*)* em C acessível pelo *script:*

```
static int lh3c_dIn(lua_State *L) {
  lua_Integer val = 0;
  int nio = luaL_checkinteger(L, 1);

  switch(nio)
  {
    case 1:
      val = GPIOPinRead( GPIO_PORTF_BASE, GPIO_PIN_1 ); // SELECT
      break;
    case 2:
      val = GPIOPinRead( GPIO_PORTE_BASE, GPIO_PIN_0 ); // UP
      break;
    case 3:
      val = GPIOPinRead( GPIO_PORTE_BASE, GPIO_PIN_1 ); // DOWN
      break;
    case 4:
      val = GPIOPinRead( GPIO_PORTE_BASE, GPIO_PIN_3 ); // RIGHT
      break;
    case 5:
      val = GPIOPinRead( GPIO_PORTE_BASE, GPIO_PIN_2 ); // RIGHT
      break;
    case 15:
      val = HWREG(GPIO_PORTC_BASE + GPIO_O_DATA + (GPIO_PIN_5 << 2));
      break;
  }

  lua_pushinteger(L, val);

  return 1;
}
```

Trecho de código para registrar o módulo h3c de forma a ser visível pela máquina virtual Lua usando *patches* do projeto eLua:

```
const LUA_REG_TYPE lh3clib[] = {

  {LSTRKEY("dIn"), LFUNCVAL( lh3c_dIn)},
  {LSTRKEY("dOut"), LFUNCVAL( lh3c_dOut)},
  {LSTRKEY("aIn"), LFUNCVAL( lh3c_aIn)},
  {LSTRKEY("aOut"), LFUNCVAL( lh3c_aOut)},
  {LNILKEY, LNILVAL}
};

…
LREGISTER(L, "h3c", lh3clib);

lua_pushnumber(L, N_AnalogOut);
lua_setfield(L, -2, "naOut");
lua_pushnumber(L, N_AnalogIn);
lua_setfield(L, -2, "naIn");
lua_pushnumber(L, N_DigitalOut);
lua_setfield(L, -2, "ndOut");
lua_pushnumber(L, N_DigitalIn);
lua_setfield(L, -2, "ndIn");
…
```

Exemplo de *script* que permite gerar uma onda quadrada com a máxima frequência possível usando o módulo h3c:

```
while true do
      h3c.dOut(17, 1) -- coloca a saída 17 em nível alto
      h3c.dOut(17, 0) -- coloca a saída 17 em nível baixo
end
```

3.6 Net services (n3c)

Provê a LuaVM serviços de acesso a rede. A princípio, por simplicidade, quatro funções serão implementadas:

- **n3c.tx(id, data)** – enviar dados para a rede (id=identificador, data=dados a serem enviados). Os dados enviados são "bufferizados" e transmitidos quando possível, segundo o protocolo sobre o qual foi implementado;
- **n3c.listen(id)** – registra um id para ser recebido pelo Net Services. Uma vez registrado um id o sistema fará *buffering* de dados recebidos com aquele id;
- **n3c.clearAll()** – limpa todos os id's registrados;
- **r,id,data=n3c.rx()** – recebe qualquer dado com um dos id's registrados. Esta função retorna três parâmetros, sendo **r** true se um novo dado foi recebido, **id** o id da informação recebida, **data** os dados recebidos. Esta função não é bloqueante para não afetar a cadência de execução do algoritmo do usuário.

3.7 OS port (o3c)

Provê recursos do sistema operacional como temporização. Alguns recursos fundamentais de temporização foram implementados para uso do *script*:

- **o3c.dms(i)** – "pausa" a execução por **i** milissegundos (equivalentes a funções típicas como *delay* e *sleep*);
- **o3c.dus(i)** – "pausa" a execução por **i** microssegundos (equivalentes a funções típicas como *delay* e *sleep*);
- **o3c.ems()** – obtém o número de milissegundos do relógio de hardware ou sistema operacional (equivalente a funções típicas como GetTickCount, getTicks, etc).

Este módulo também é responsável pela portabilidade de alguns recursos internos usados pelo Lua Manager como criação de tarefa e acesso a semáforo, ambos não visíveis ao *script*.

3.8 Net port (np3c)

Camada que permite ao Net services e ao Lua Manager acesso à rede. É por meio desta camada que controla-se remotamente o Lua Manager para realizar a configuração e atualização do script que a LuaVM irá executar. Ela também encapsula os dados da camada Net Services para comunicação entre as aplicações do usuário (*scripts*) executando em diferentes equipamentos.

Esta abordagem permite que uma rede com distribuição de mensagens identificadas seja implementada sobre diversos protocolos e, ao mesmo tempo, propicia facilidade de uso por parte do *script*. Ela também permite priorizar as mensagens de comunicação com dados de controle (provindos dos *scripts*) e as mensagens de gerenciamento, com menor prioridade, endereçadas ao Lua Manager.

Para validação utilizou-se uma rede CAN. Nela foi atribuído o id ao id da mensagem CAN e o dado ao campo de dados da mensagem CAN que pode ter até oito bytes. Nesta camada a partir do ID da mensagem CAN fez-se o roteamento da mensagem para o Lua Manager ou para o Net services. Nesta implementação de validação foi criada uma tarefa que faz o roteamento das mensagens entre os diferentes “entes”. As mensagens vindas do controlador CAN chegam através de filas, à rotina de interrupção do controlador alimenta a fila e a tarefa do Net services remove da fila. Processo semelhante é criando no caminho da aplicação para a rede com uma fila de envio. Se a fila de envio está vazia, o controlador está disponível e o *frame* é colocado diretamente no controlador, se a fila de envio não estiver vazia acrescenta-se aquela a nova mensagem. Quando a interrupção de final de transmissão ocorrer ela remove da fila de envio a próxima mensagem a ser enviada.

3.9 Protocolo de comunicação com o Lua Manager

Um protocolo chamado MANPRO foi definido e implementado para que a interface de download e gerenciamento se comunique com o Lua Manager em cada dispositivo através da mesma rede de controle.

Comparado às camadas de referência do modelo ISO/OSI este protocolo corresponde as camadas 4, 5 e 7 do modelo OSI conforme Figura 3 (os serviços abrigados na camada 6, de apresentação, não foram necessários – criptografia, conversão de formatos e mapeamentos).

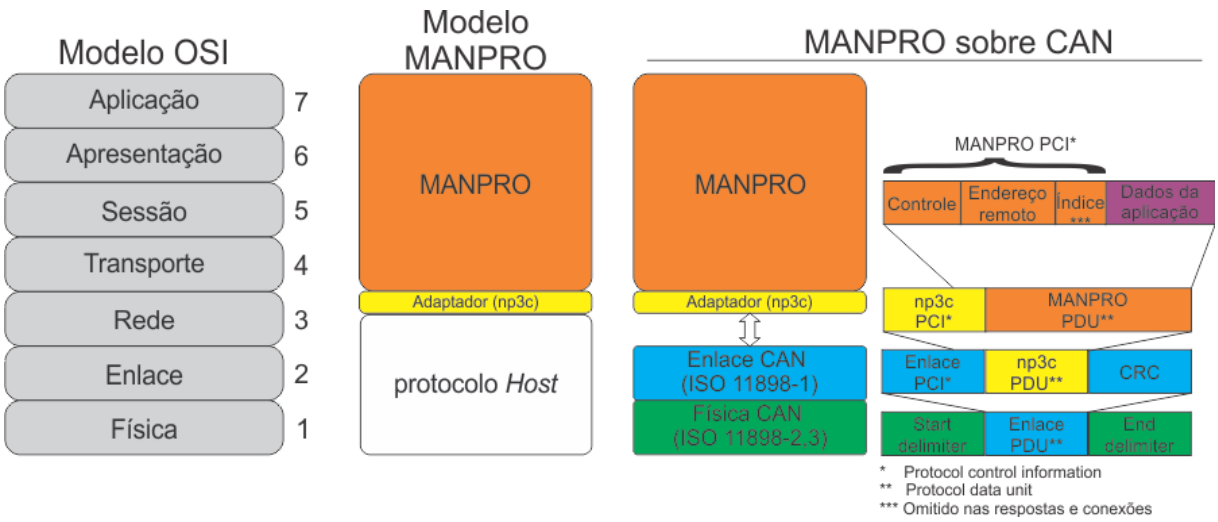


Figura 3 – Comparação do protocolo com o modelo de referência ISO/OSI.

Por se tratar de um protocolo simples com foco no mínimo consumo de recursos, cada requisição, dentro de uma sessão, precisa aguardar a resposta antes que seja enviada outra requisição (*max outstanding=0*) (TANENBAUM, 2002). Limitou-se ainda o número máximo de sessões simultâneas para 5 (configurável em código conforme disponibilidade de RAM) e o número máximo de sessões por estação remota em 1. A título de exemplo: pode haver cinco interfaces de gerenciamento remotas conectadas simultaneamente, mas cada estação remota pode abrir apenas uma sessão. As sessões podem ser desconectadas nas seguintes condições: pedido de desconexão, timeout, pedido de uma segunda conexão pela mesma estação remota e requisição remota repetida (mesmo ID de requisição) para índices diferentes.

Os serviços oferecidos pelo MANPRO são *connect*, *read*, *write* e *disconnect* permitindo a implementação de aplicações baseadas em MIB (*Management Information Base*) como as usadas no protocolo SNMP, no Foundation Fieldbus™ e no próprio Lua Manager. Objetos grandes como o *script* no Lua Manager podem ser fragmentados em várias partes conforme as limitações da camada de interface do protocolo "*host*" logo abaixo (para o protocolo CAN 8 bytes máximos de SDU).

A Figura 4 dá os detalhes da PDU do MANPRO.

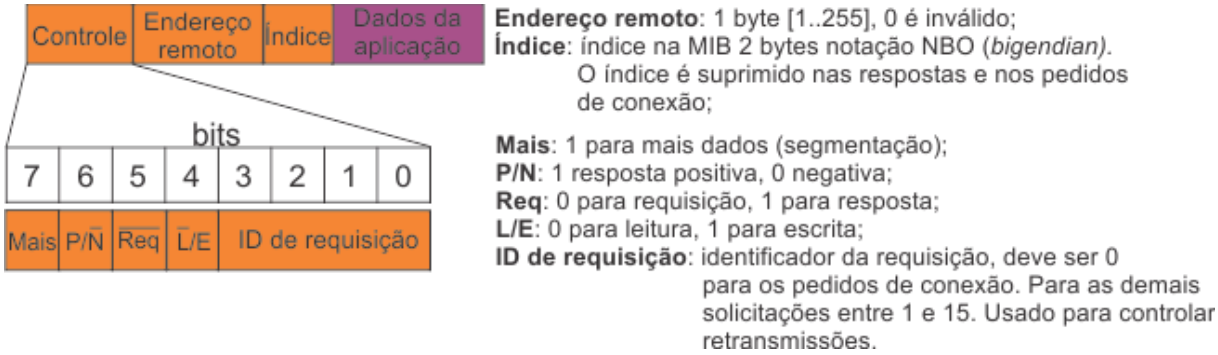


Figura 4 - Detalhe da PDU do MANPRO

Para melhor aproveitamento dos bits de controle, o serviço de conexão é implementado como uma mensagem de leitura com ID de requisição 0 (inválido para serviços *read* e *write*). Por sua vez, a desconexão usa uma mensagem de escrita com ID de requisição 0.

Por se tratar de um protocolo de camada superior de aplicação, nenhum mecanismo de verificação de integridade de dados foi incluído. Caso a camada de baixo não possua nenhum mecanismo de verificação deve-se acrescentar a verificação na camada de adaptação (np3c).

O kit de desenvolvimento usado para validação da Plataforma 3C possui uma interface CAN – CAN é largamente usado em redes industriais, sistemas automotivos e aviônica. Devido à disponibilidade e ao uso consagrado de CAN em diversos segmentos o MANPRO foi mapeado para este protocolo através da camada de adaptação (np3c).

Uma mensagem CAN contém bits de controle, identificador de 29bits (para CAN 2.0B), até 8 bytes de dados e 15 bits de CRC (*Cyclic Redundancy Check*). Reservou-se uma faixa de mensagens CAN iniciadas com identificador (ID) em hexadecimal 05 XX XX XX para o mapeamento do MANPRO sobre CAN. Os três bytes seguintes foram usados como: controle, endereço destino e endereço fonte respectivamente, pois a mensagem CAN não inclui na sua estrutura um mecanismo de endereçamento de estações. A Figura 5 apresenta o mapeamento de cada bit no identificador da mensagem:

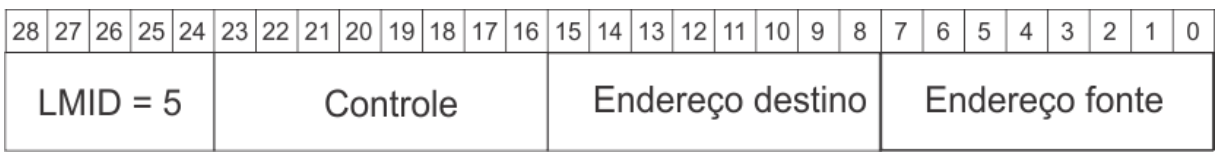
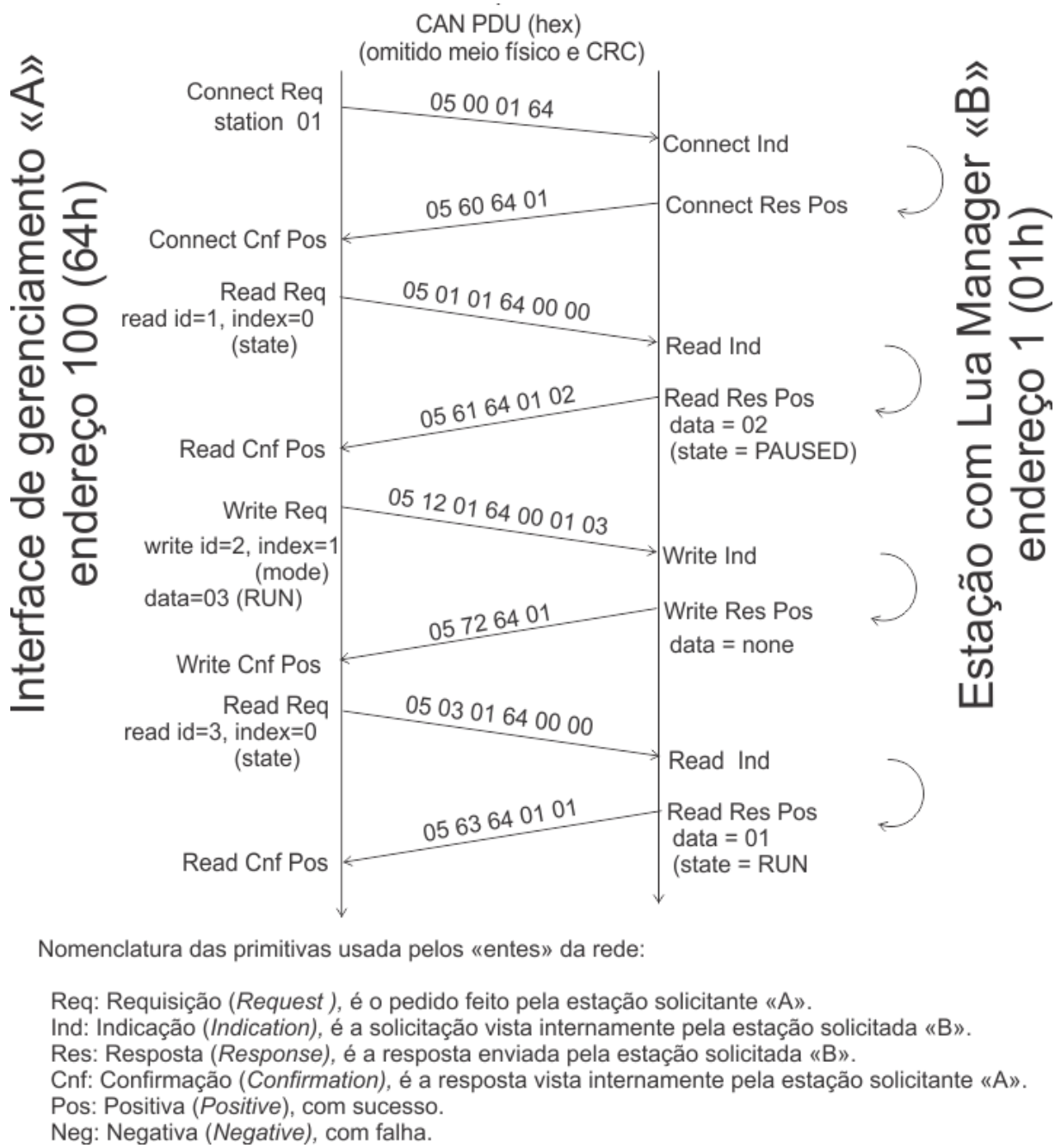


Figura 5 - Mapeamento do identificador da mensagem

Exemplo de comunicação entre a interface de gerenciamento e uma estação com a Plataforma 3C sobre CAN.



Nomenclatura das primitivas usada pelos «entes» da rede:

Req: Requisição (*Request)*, é o pedido feito pela estação solicitante «A».
Ind: Indicação (*Indication)*, é a solicitação vista internamente pela estação solicitada «B».
Res: Resposta (*Response)*, é a resposta enviada pela estação solicitada «B».
Cnf: Confirmação (*Confirmation)*, é a resposta vista internamente pela estação solicitante «A».
Pos: Positiva (*Positive*), com sucesso.
Neg: Negativa (*Negative)*, com falha.

Figura 6 – Exemplo de comunicação com o Lua Manager através do MANPRO

No exemplo da Figura 6 a estação de gerenciamento com endereço 100 (64h) inicia uma conexão com a estação contendo o Lua Manager no endereço 01h. Todos os bytes da PDU CAN estão indicados em hexadecimal. Estabelecida a conexão, a estação de gerenciamento faz uma leitura do parâmetro de estado (*state*, índice 0) da LuaVM e descobre

que ela se encontra no estado *PAUSED*, faz então uma escrita no parâmetro de requisição de modo (índice 1) para modo 3 (*run*). A resposta positiva da escrita é retornada para a estação de gerenciamento. Para confirmação da mudança, a estação de gerenciamento faz outra leitura para *state* e finalmente recebe a informação de que a LuaVM se encontra no estado 1 (*running).*

O MANPRO pode ser implementado em cima de outros protocolos através do mapeamento adequado (nem todos os protocolos e redes existentes serão adequados a este mapeamento). Por exemplo, o mapeamento em cima do UDP/IP (POSTEL, 1980) pode ser feito reservando-se uma porta UDP para o Lua Manager e o primeiro byte de dados (SDU) para o byte de controle, o endereço fonte pode ser mapeado diretamente como o byte menos significativo do endereço IP (limitando-se a uma rede classe C (POSTEL, 1981)). Para um mapeamento mais amplo, em outras classes de rede IP, faz-se o mapeamento de um endereço IPv4 de 32 bits ou IPv6 de 128 bits juntamente com a porta origem UDP em um identificador local de 8 bits (guardado em tabela) gerado pela camada de adaptação (np3c). Em um datagrama UDP/IP, então, ter-se-ia:

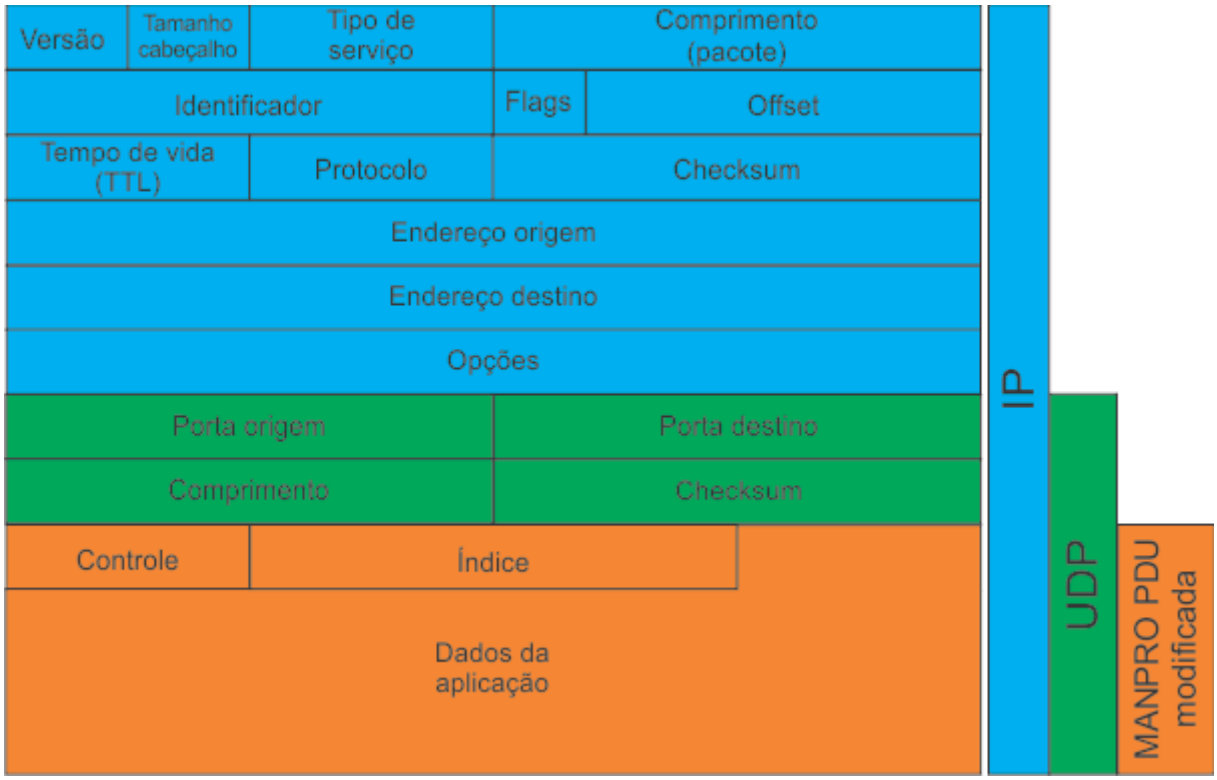


Figura 7 - Mapeamento de MANPRO sobre UDP/IP

Notar que foi suprimido o endereço remoto da PDU MANPRO modificada na Figura 7, feito pela camada de adaptação (np3c). O identificador local de 8 bits foi gerado para criar a PDU

MANPRO a partir do mapeamento de IP origem e porta origem UDP. No retorno, o identificador local de 8 bits é usado para encontrar o endereço IP origem e a porta origem UDP (que serão usados como destino), o identificador local de 8 bits é suprimido e a PDU MANPRO modificada enviada como dado do pacote UDP.

3.10 *Gateway* UART/CAN

Como o PC não possui nativamente uma interface CAN, foi desenvolvido um gateway UART/CAN para fazer a interface. A placa para teste de dispositivo CAN que acompanha o kit Stellaris possui: um microcontrolador LM3S2110, interface CAN, três botões (sendo um RESET) e demais pinos do microcontrolador disponíveis na placa (Figura 8). Destes pinos disponíveis foram usados dois pinos e atribuídas função de TX e RX assíncrono (UART). Em seguida interconectado a um PC através de adaptador USB para UART TTL.

Figura 8 - Placa que acompanha o kit Stellaris usada como "*CAN test device*"

O protocolo CAN trabalha com o conceito de mensagem claramente delimitada, um telegrama ou *frame* com bits de controle, identificador de mensagem e dados. Por outro lado a

UART delimita apenas os símbolos (normalmente 8 bits) através de *start* bit e *stop* bit. Usou-se um protocolo conhecido como ***Serial Line Internet Protocol (SLIP)*** para delimitar os *frames* na sequência de dados da UART através da reserva de alguns bytes (0xC0=*END*, 0xDB=*ESC*) para o gateway. O primeiro byte conhecido como *END* sinaliza o fim de um *frame* e só aparece neste caso. O segundo conhecido como *ESC* (*escape*) informa que este byte precisa ser removido e o seguinte substituído. Desta forma, se o dado do frame contiver o byte 0xC0, o codificador o substitui por *ESC* seguido de *ESC_END* (0xDC) e se o dado do *frame* contiver um *ESC* o mesmo é substituído por *ESC* seguido de 0xDD (*ESC_ESC*). Por sua vez o receptor faz o processo inverso.

Um exemplo de codificação de dois *frames* é mostrado na Tabela 3.

Tabela 3 - Codificação de *frame* com SLIP

Frame	*Frame* **codificado**
00 11 22 33 44	00 11 22 33 44 C0
00 11 **C0** 33 **DB**	00 11 **DB DC** 33 **DB DD** C0

O primeiro *frame* não possui bytes reservados, por isso apenas adiciona-se o delimitador *END* ao final. Já o segundo *frame* possui os dois bytes reservados, eles são devidamente codificados, segundo a convenção e finalmente o delimitador final é acrescentado É comum algumas implementações considerarem o delimitador *END* também no início do *frame* de forma que os dados fiquem completamente confinados e não seja necessário outro mecanismo de recuperação de sincronismo (temporizadores) ou validação de dados incompletos.

3.11 Interface de *download* e gerenciamento

A interface da Figura 9 foi desenvolvida para fazer testes de protocolo e o gerenciamento remoto das estações conectadas na rede.

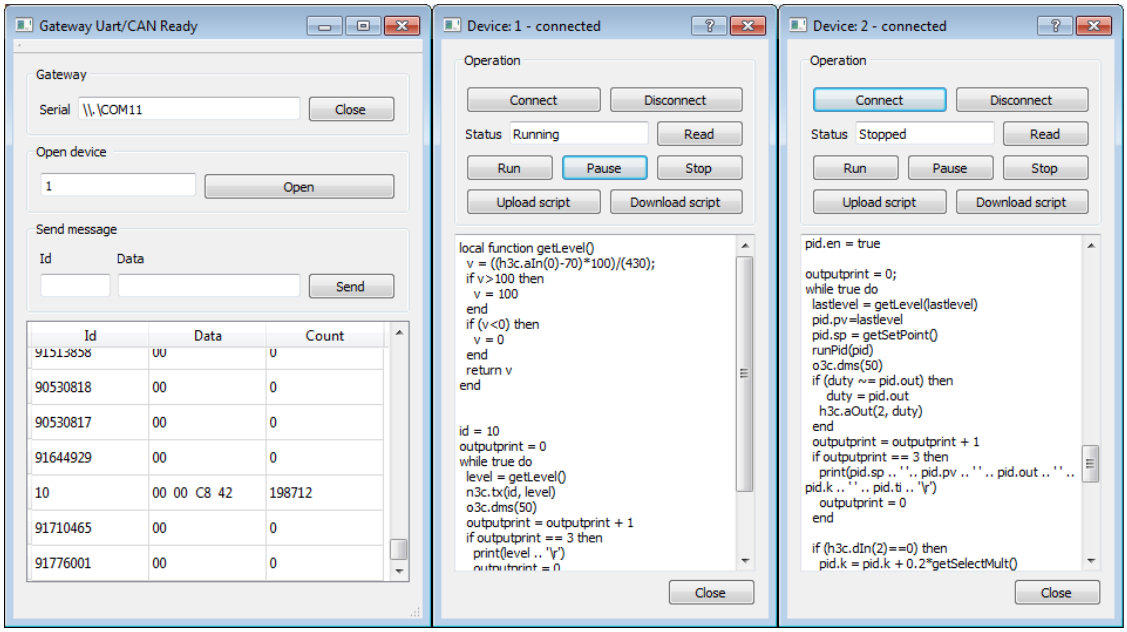


Figura 9 - *Gateway* UART/CAN desenvolvido

Resumidamente, a primeira janela (à esquerda) permite abrir a interface serial associada, visualizar todos os frames CAN que estão sendo recebidos pela interface CAN do *gateway* (mostra identificador da mensagem até 29 bits, os dados em hexadecimal e um contador que registra quantas vezes a mensagem com um dado Id apareceu). Permite ainda enviar qualquer mensagem CAN posta nos campos "Id" e "*Data*" seguida de click no botão "*Send*". Esta interface pode ser usada para inspecionar a rede CAN de um automóvel, identificando facilmente as mensagens enviadas na rede *(*monitor de rede ou *sniffer)* e até testar o envio modificado ou não de alguma mensagem.

O grupo "*Open device*" por sua vez permite abrir qualquer estação remota que tenha o Lua Manager implementado. No exemplo acima foram abertos dois dispositivos remotos (janelas da direita) com endereços 1 e 2 em seus títulos respectivamente. Esta interface já implementa a parte cliente do protocolo MANPRO, facilidades para o usuário conectar com o Lua Manager, requisitar que o script seja executado através do botão "*Run*", suspenso temporariamente através do botão "*Pause*" e encerrado por meio do botão "*Stop*". Permite ainda enviar um *script* do usuário colocado na caixa de texto ou recuperar o último *script* dentro do equipamento através do botão "*Upload script*".

3.12 Conclusão

Embora algumas características de Lua não sejam ideais para sistemas embarcados como o uso da função **realloc()** da biblioteca padrão C, o uso de um coletor de lixo *(garbage collector)* e alto uso de RAM (comparado a um algoritmo em código nativo), seu desempenho ainda é satisfatório para sistemas contínuos onde os tempos de mudança do processo são mais longos (da ordem de dezenas ou centenas de milisegundos). Nos experimentos realizados, a execução do controlador de nível com algoritmo PID executou, em média, em menos de 600µs para um clock de 50MHz. Para evitar a fragmentação Lua permite que o usuário defina sua própria função de alocação no lugar de **realloc().** Algoritmos de alocação como *buddy system*, *Fibonacci heap* e outros que criam blocos pré-definidos podem ser ajustados à necessidade e minimizar ou eliminar a fragmentação (SCOTT, 2009). Os consumos de memória RAM e ROM podem ser reduzidos removendo-se o compilador da máquina virtual. Neste caso, a ferramenta de programação compilaria o *script* e enviaria o "*byte code"* resultante para a execução. O ponto negativo é perder a capacidade de fazer *upload* de um *script*, útil em casos em que se tem o equipamento em campo funcionando e perdeu-se o *script*[2].

Existem alguns riscos inerentes ao uso de **GC** (*garbage collector)* em sistemas embarcados. É muito difícil, se não impossível, garantir que o **GC** não entre em momentos críticos ou que não seja necessário frequentemente para liberação de recursos. Klotzbuecher e Bruyninckx (2011) propoem algumas técnicas para reduzir o tempo de atuação do **GC** de Lua e tentar tornar sua execução um pouco mais determinada, atribuem ainda com grande importância o uso de dados experimentais com coleta de tempo de execução em vários cenários de uso. Ademais, Kalogirou (2014) propõe uma alteração no **GC** do Lua para ser executado de forma quase previsível. Em Lua o uso de variáveis locais em um loop principal com poucas chamadas de funções que tenham variáveis locais também contribui para a redução da necessidade do **GC**. Ressalta-se que para o *script* do Anexo A a única parte que demandou chamadas de realloc() foi a função *print*.

[2] Removido o compilador, pode haver memória não volátil suficiente (dependendo do hardware) para armazenamento do *script* original de forma compactada além do "*byte code"* para *backup* (não disponível para o equipamento, mas para a ferramenta de configuração).

A forte integração de Lua com a linguagem C, através do uso de módulos, permite fácil extensibilidade e portabilidade em diversas plataformas bem como o reaproveitamento de protocolos de rede e algoritmos de processamento digital de sinais existentes.

O protocolo MANPRO desenvolvido para comunicação com o Lua Manager pode ser estendido para a monitoração de pontos de supervisão internos do algoritmo definindo-se uma interface semelhante às listas de tags e parâmetros de blocos usadas em CLPs e transmissores digitais respectivamente. O mapeamento deste protocolo em CAN optou pelo uso do ID das mensagens CAN. Em uma rede com tráfego muito carregado este mapeamento pode não ser ideal, pois a arbitragem em caso de colisão é dependente primeiramente do conteúdo deste ID. No entanto, como as mensagens com dados de controle ficaram restritas a IDs de valores mais baixos, elas terão precedência na arbitragem em relação àquelas de gerenciamento (as mensagens do MANPRO).

O gateway UART/CAN associado à Interface de download e gerenciamento é uma ferramenta de depuração e testes para a Plataforma 3C bem como para outras redes que usem o meio físico CAN como as automotivas (GMLAN, SAE J1939, EnergyBus,etc) e industriais (CANOpen, DeviceNET, SafetyBUS). Uma melhoria futura da interface é acrescentar à sua interface gráfica um campo para entrar com o valor desejado de endereço na rede usado pelo Lua Manager “cliente” fixo em 100 (atualmente é necessário recompilar o código para alterá-lo).

4 RESULTADOS EXPERIMENTAIS

4.1 Introdução

Para validar a Plataforma 3C, foram realizados quatro conjuntos de experimentos. Os dois primeiros são bastante simples: “leitura, escrita e transmissão digital entre dois dispositivos” e “leitura digital com transmissão pela rede CAN e transmissão digital em outro dispositivo”. O terceiro é um controle de nível em uma planta didática com sistema centralizado e, em seguida, distribuído. Por fim, o conjunto de testes de desempenho com foco na máxima velocidade de execução.

Os experimentos foram realizados com um ou dois kits de desenvolvimento Luminary Micro - EKI-LM3S8962, equipados com microcontrolador Stellaris ARM® Cortex™-M3 (50MHz de clock por meio de PLL, 64KiB de SRAM e 256KiB de memória FLASH) e interface de rede CAN.

O requisito de memória total da Plataforma 3C na arquitetura ARM Cortex-M3, utilizando-se o compilador GCC 4.8.1 com otimização O2 é de 146KiB de FLASH e 35KiB de RAM para os scripts aqui testados. Estão inclusos inclusive FreeRTOS 7.5.2, funções matemáticas (sin, cos, log, sqrt, etc), de formatação de strings (sprintf, printf, etc) e de alocação (realloc, free) fornecidas pela newlib 2.0.0 (libc e libm), drivers para os periféricos CAN, UART, PWM e ADC do microcontrolador LM3S8962.

4.2 Exemplo de leitura, escrita e transmissão digital entre dois dispositivos

Cada resultado experimental apresentado a seguir é composto pelo diagrama de blocos da conexão entre as placas e sinais, do *script* utilizado em cada dispositivo e das respectivas formas de ondas obtidas por meio de um osciloscópio Agilent DSO-X 2002A.

A Figura 10 mostra o diagrama de blocos do Experimento 4.2. Neste diagrama, o Dispositivo 01 está executando o *script* A, conforme apresentado na Tabela 4 e o Dispositivo 02 está executando o *script* B, mostrado na Tabela 5.

O *script* A gera uma forma de onda de 500Hz no saída digital 17, que foi configurado na camada h3c para ser o pino PC17 do microcontrolador LM3S8962. O *script* B lê o sinal transmitido na entrada 1, pino PC15 do microcontrolador, e reproduz a leitura na saída digital 17, que corresponde ao pino PC17 do microcontrolador. Os resultados experimentais deste experimento são exibidos na Figura 11.

Vale ressaltar que, para cada equipamento, o fabricante deverá mapear as funções do HAL (h3c) aos periféricos disponíveis no seu equipamento e apresentar uma tabela no manual de forma que o usuário saiba qual a entrada ou saída lógica está associada a uma dada entrada ou saída física.

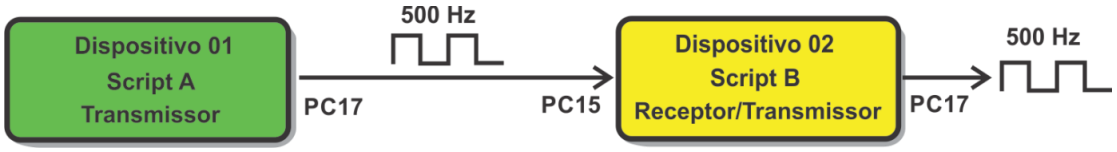


Figura 10 – Diagrama de blocos do experimento 4.2.

Tabela 4 – *script* A - Gera onda quadrada com frequência de 500Hz na primeira saída digital.

```
while true do
      h3c.dOut(17, 1) -- coloca a saída em nível alto
      o3c.dms(1)      -- aguarda 1ms
      h3c.dOut(17, 0) -- coloca a saída em nível baixo
      o3c.dms(1)      -- aguarda 1 ms
end
```

Tabela 5 – *script* B - Reproduz o sinal de uma entrada em uma saída.

```
while true do
      h3c.dOut(17, h3c.dIn(1)) -- lê o valor da entrada 1 e aplica na saída 17
end
```

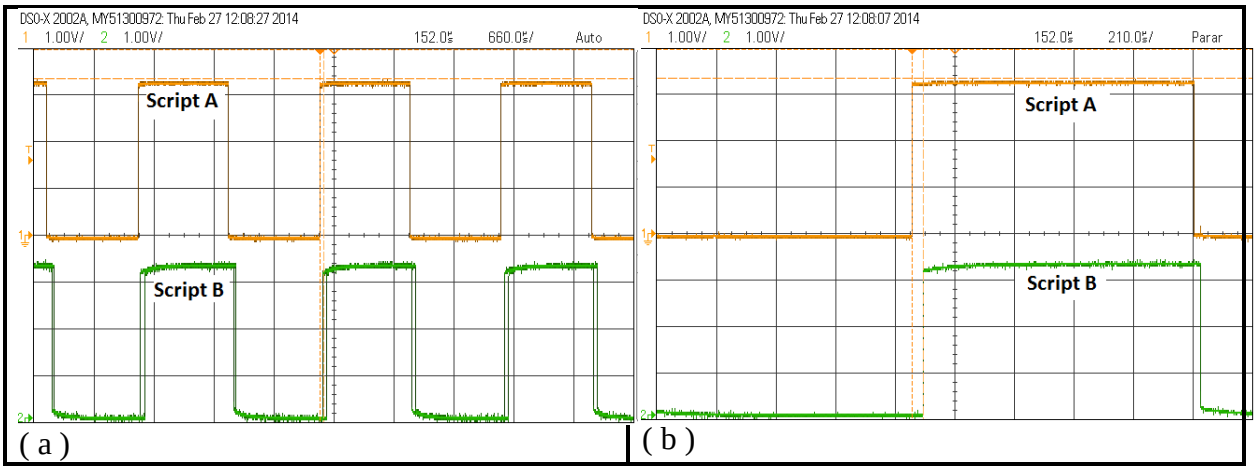


Figura 11 - (a) Forma de onda de saída do Dispositivo 01 -*script* A - 1V/div - 660µs/div e forma de onda de saída do Dispositivo 02 –*script* B – 1V/div - 660µs/div (b) atraso de propagação entre as formas de onda de 40µs – 1V/div - 210µs/div.

4.3 Exemplo de leitura digital, transmissão pela rede CAN e transmissão digital em outro dispositivo

A Figura 12 exibe o diagrama de blocos do Experimento 4.3. Neste diagrama, o Dispositivo 01 está executando o *script* C, conforme mostrado na Tabela 6 e o Dispositivo 02 está executando o *script* D, apresentado na Tabela 7.

Figura 12 – Diagrama de Blocos do Experimento 4.3.

Tabela 6 – *script* C: Lê sinal digital na entrada digital 1 e transmite o sinal pela rede CAN.

```
Id = 10
while true do
    n3c.tx(id, h3c.dIn(1)) -- lê a entrada 1 e publica na rede com id=10
end
```

Tabela 7 - *script* D: Recebe sinal pela rede CAN e reproduz o sinal na saída digital 17.

```
idl = 10
n3c.listen(idl) -- registra o id=10 para ser lido pelo stack de rede
while true do
  r,id,data = n3c.rx() -- lê o dado que estiver disponível (r informa se há algum)
  if r==true then
     h3c.dOut(17, data) --coloca na saída 17 o valor recebido
  end
end
```

O *script* C lê o valor imposto por uma forma de onda quadrada de 500Hz no pino PC15 do Dispositivo 01, que foi configurado para ser a entrada digital 1. Em seguida, o valor lido é publicado na rede CAN sob o id=10. Na sequência, o dado é recebido pela interface CAN do Dispositivo 02 e aplicado à saída digital 17 associada ao pino PC17 do mesmo, as formas de onda podem ser vistas na Figura 13.

No experimento 4.2, o atraso de propagação foi de 40µs. Por outro lado, no Experimento 4.3, quando o dado foi transmitido de um dispositivo para outro, por meio da rede CAN, o atraso de propagação sofreu um acréscimo de 780µs. Este atraso foi imposto pela rede CAN, pelas interfaces de software e pela velocidade de execução do script. Devido à rede CAN temos que uma mensagem de 4 bytes de dados como esta pode ter, no pior dos casos 158 bits, devido ao *bit stuffing* (bit extra adicionado no pacote de acordo com a convenção CAN, semelhante ao protocolo HDLC, para se evitar uma sequência muito longa de bits com o mesmo nível). Isto equivale a aproximadamente 191us para uma rede CAN com taxa de 500kbps sem levar em conta possíveis perdas de arbitragem. O restante do atraso se deve ao software.

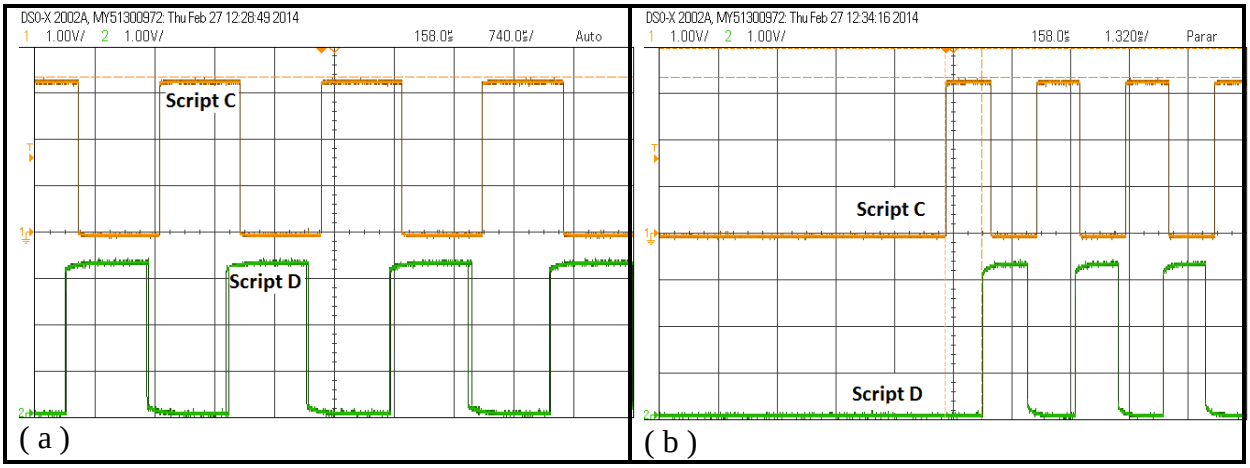


Figura 13 – (a) Forma de onda de saída do Dispositivo 01-*script* C-1V/div - 740µs/div e forma de onda de saída do Dispositivo 02–*script* D–1V/div – 1320ms/div (b) atraso de propagação entre as formas de onda de 820 µs.

4.4 Experimento de Controle de Nível em uma Planta

Para um experimento de controle de nível foi usada a planta apresentada na Figura 14 (diagrama na Figura 15), composta de dois tanques, tubos, conexões, válvula para descida por gravidade “V”, bomba de subida “B” e sensor de nível “S”. A válvula de descida manual é do tipo esférica para tubulação de 1/2 polegada, a bomba centrífuga tem vazão máxima de 10L/min, corrente nominal de 1A e tensão de 24Vcc, para medir o nível foi usado um sensor de pressão MPX5010DP com alimentação de 5Vcc, faixa de 0 a 10kPa e com saída linear de 0 a 5V (para o nível do tanque usado o máximo de saída é 3V).

A planta contém ainda válvula de entrada de água externa ao sistema e válvula para escoamento externo, ambas não usadas na aplicação. Foram colocados três volumes mortos (PETs de refrigerante) para redução de volume do tanque superior de forma a reduzir o volume do reservatório e, consequentemente, acelerar a velocidade de mudança de nível.

O objetivo deste experimento é controlar o nível do tanque superior, sendo, a PV *(process variable)* o nível daquele tanque, SP *(setpoint)* o nível desejado, configurável pelo usuário através de potenciômetro ligado em uma entrada analógica da placa executando a Plataforma 3C.

4.4.1 Hardware do controlador

O hardware do controlador digital é composto dos seguintes itens (Figura 16):

- Placa de avaliação Stellaris® LM3S8962 (microcontrolador LM3S8962 Cortex-M3 64KiB RAM, 256KiB FLASH com entradas analógicas e saídas PWM);
- Filtro RC, Figura 17 para acompanhamento dos sinais no osciloscópio com redução de ruídos;
- Potenciômetro analógico para ajuste do nível (s*etpoint);*
- Driver isolado de potência para o motor da bomba: entrada e saída PWM conforme Figura 18.

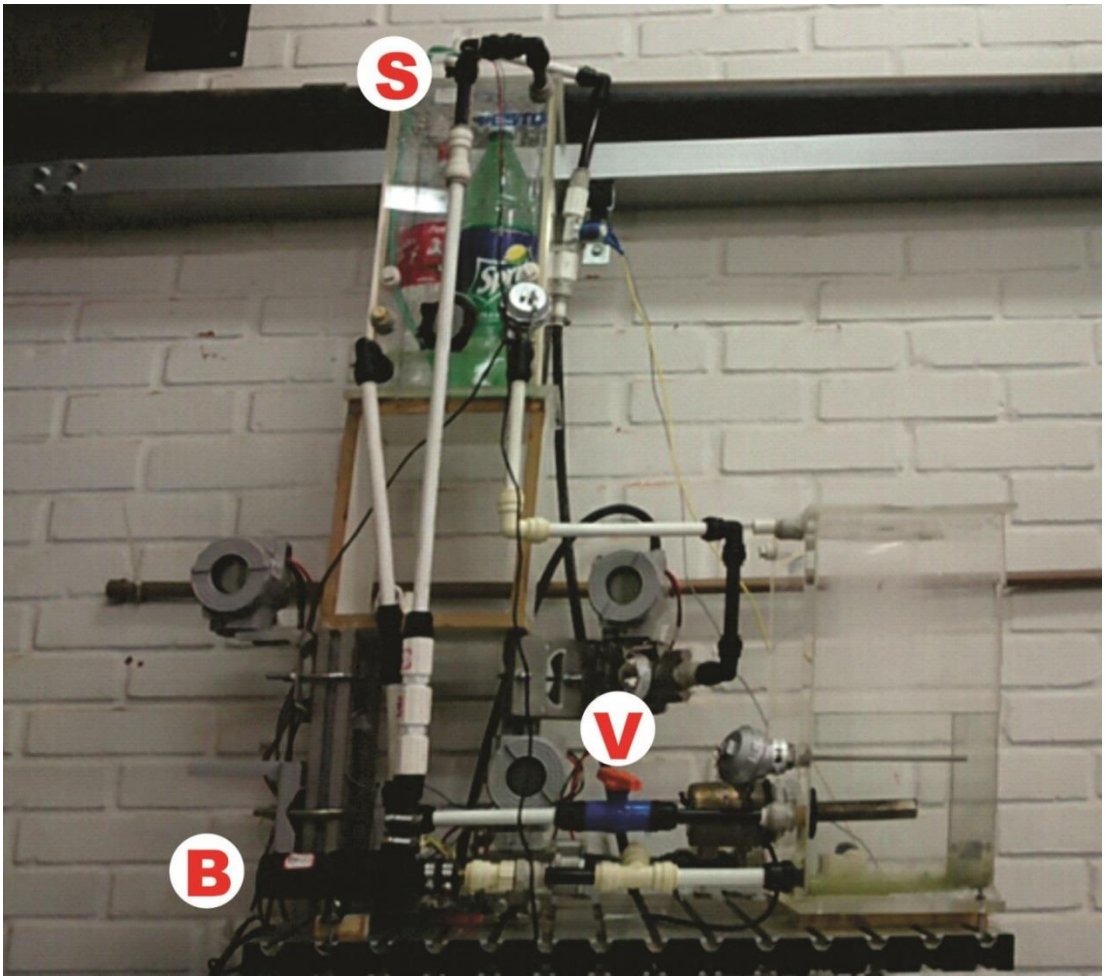


Figura 14 – Planta usada para controle de nível (B: bomba, S: sensor de nível, V: válvula de descida).

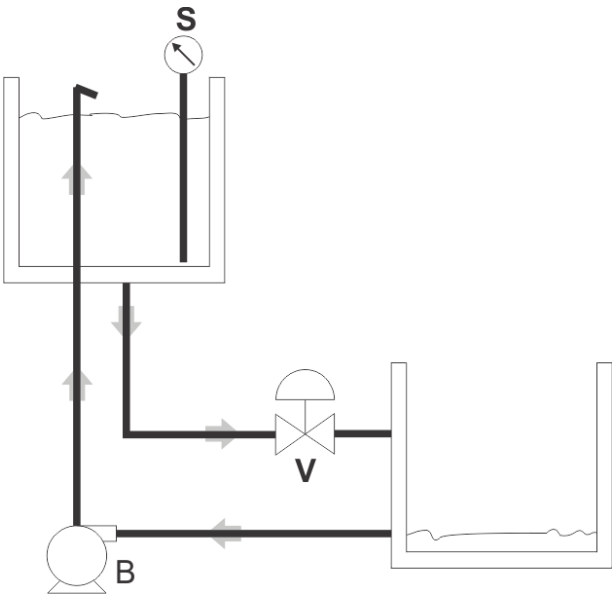


Figura 15 – Diagrama da planta de nível (B: bomba, S: sensor de nível, V: válvula de descida).

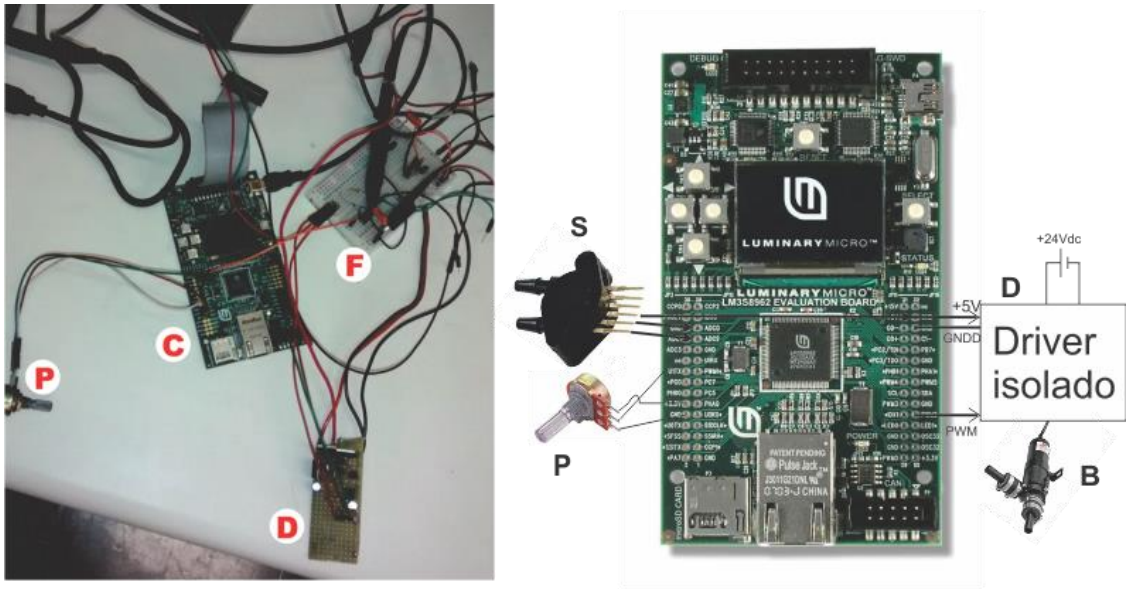


Figura 16 – Montagem do controlador (*C:* controlador, *D*: driver, *P*: potenciômetro; *F*: filtro) à esquerda. À direita detalhes da ligação.na placa do controlador C.

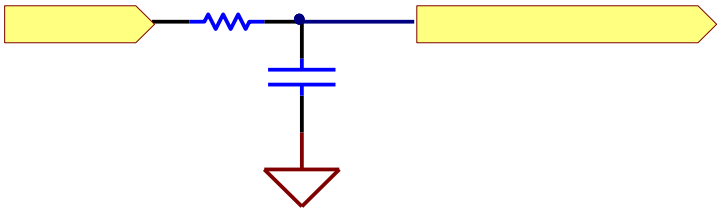


Figura 17 – Filtro para leitura do sensor pelo osciloscópio (sinal do sensor entra no filtro e no controlador C).

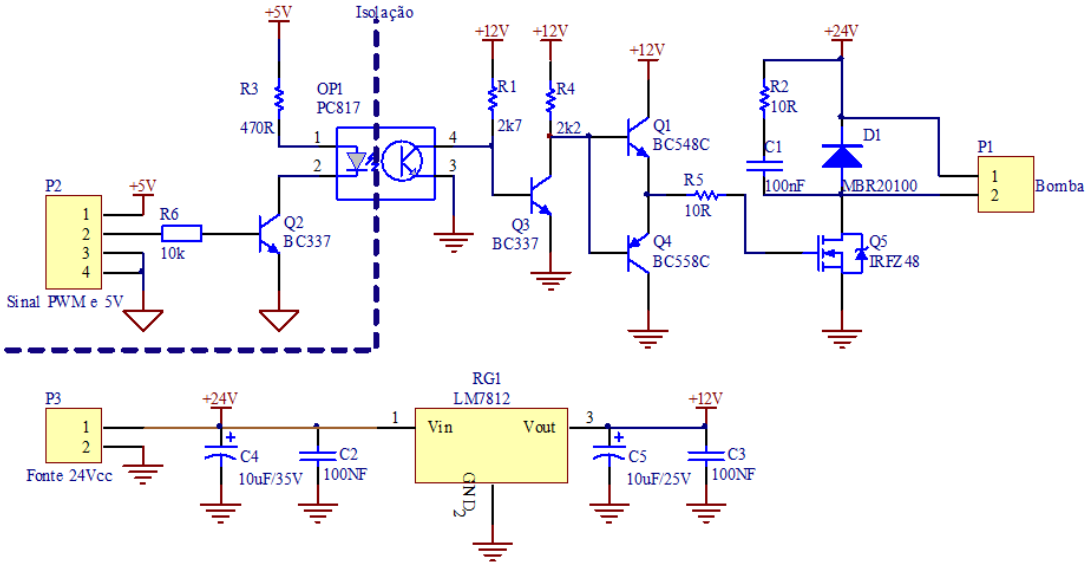


Figura 18 – Driver “D” desenvolvido para acionamento da bomba.

4.4.2 Estratégia de Controle Adotada

Para controle de nível são normalmente usados controladores do tipo PI. Não é o foco deste trabalho fazer uma discussão sobre o controle mais adequado para a planta utilizada, mas demonstrar as capacidades da Plataforma 3C em uma aplicação usual.

Desta forma foi implementado um controlador PID em um *script* (Apêndices A e B). O *script* foi então descarregado na placa executando a Plataforma 3C.

O diagrama de blocos do sistema é indicado na Figura 19:

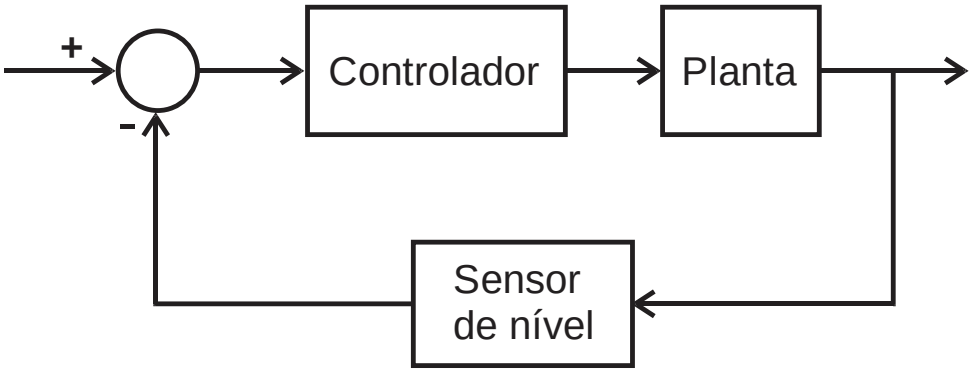


Figura 19 – Diagrama de blocos

O controlador implementado no *script* é a forma série do controlador PID, mais comum nos controladores industriais e de ajuste experimental mais fácil.

O ajuste do controlador integral e derivativo foram realizados de acordo com os artifícios apresentados em (SHAW, 2006). Neste trabalho, o derivativo é calculado a partir da saída do processo e não do erro, isto serve para evitar que mudanças no *setpoint* causem distúrbios na planta. O valor do integrador é calculado usando-se o sinal de realimentação da saída do controlador, com atraso, e aplicando-se o ganho integral sob esta diferença.

O diagrama de blocos do controlador com a planta no domínio Z fica um pouco diferente do usual e é mostrado na Figura 20.

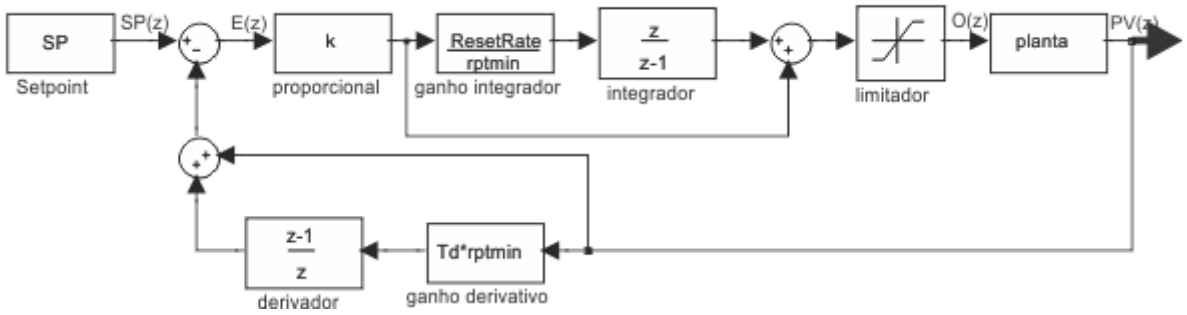


Figura 20 - Diagrama de blocos do controlador

Após alguns testes experimentais, os valores de ajuste do controlador PI foram:

- proporcional *k=4.84;*
- integral *ResetRate =4.18;*
- derivativo *Td=0* (não usado, controle PI).

OBS: O método Ziegler Nichols de malha fechada não foi utilizado porque a oscilação do sistema não é estável.

4.4.3 Ensaios com sistema centralizado

Foram realizados alguns ensaios de usos típicos para monitorar o comportamento da planta. Os valores para geração dos gráficos foram obtidos via saída do próprio *script "print(pid.sp .. ' '.. pid.pv .. ' ' .. pid.out .. '\r')"* a cada 150ms para a porta \serial virtual conectada na interface USB do computador. Todos os valores foram colocados em porcentagem e uma única placa de avaliação Stellaris® LM3S8962 rodando o *script* do Apêndice A realizou todo o controle.

4.4.3.1 Primeiro ensaio

No primeiro ensaio (Figura 21) aguardou-se a estabilização do sistema com a válvula de descida "V" completamente aberta e nível em 23%.

Em seguida alterou-se o *SP (setpoint)* para 50%. A Figura 21 mostra o comportamento do sistema com um pequeno "*overshoot*" para a PV (*process variable,* aqui o nível medido) e da saído do atuador (tensão aplicada na bomba) durante o ensaio. Nota-se que o sinal de controle do atuador estabiliza-se em um patamar superior devido ao aumento da coluna d'água e consequentemente maior taxa de reposição devido a maior vazão de saída.

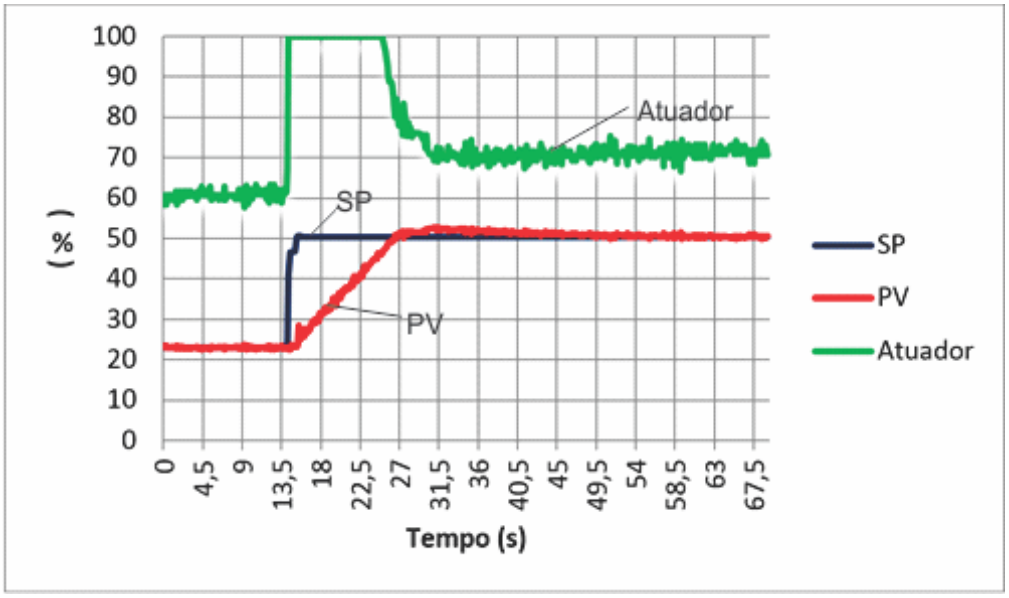


Figura 21 – Mudança de SP de 23 para 50. Válvula de descida "V" completamente aberta.

4.4.3.2 Segundo ensaio

Este ensaio foi realizado com as mesmas condições iniciais do anterior, porém a válvula de descida "V" não estava completamente aberta. Da mesma forma a estabilização do sinal de saída ficou acima do valor inicial, mas apenas ligeiramente. Neste caso, com a válvula um pouco mais fechada houve menor influência da vazão de saída em função do novo patamar. O comportamento pode ser observado na Figura 22.

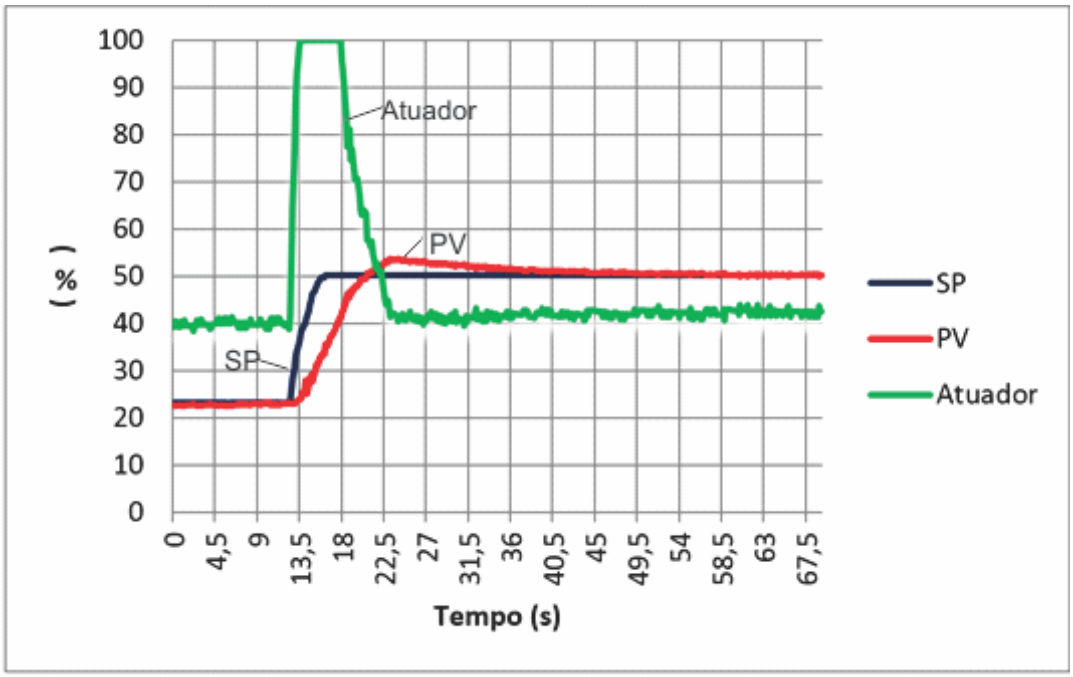


Figura 22 – Mudança de SP de 23 para 50. Válvula de descida "V" parcialmente fechada

4.4.3.3 Terceiro ensaio

Condições iniciais idênticas às condições finais do estágio anterior (ou seja SP=50%), e SP ajustado para 95%. Novamente, o sinal do atuador estabilizou-se em um valor um pouco acima devido a maior pressão exercida pelo novo nível (Figura 23).

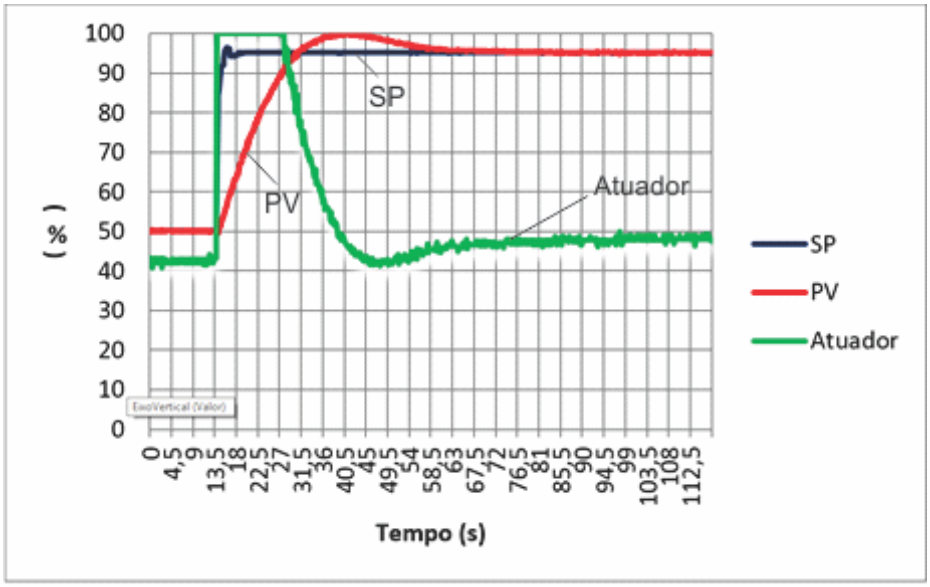


Figura 23 – Mudança de SP de 50 para 95. Válvula de descida "V" parcialmente fechada.

4.4.3.4 Quarto ensaio

Com o nível estabilizado em 95% abriu-se completamente a válvula de descida "V" em *t=22s* de forma a registrar o comportamento do sistema com o aumento da vazão de saída e não do aumento do SP. Na Figura 24, nota-se o comportamento da abertura e, ainda, por volta do instante t=121s com o sistema já estabilizado no novo patamar a válvula foi ligeiramente fechada causando pequena redução no sinal do atuador e quase imperceptível variação do nível (PV).

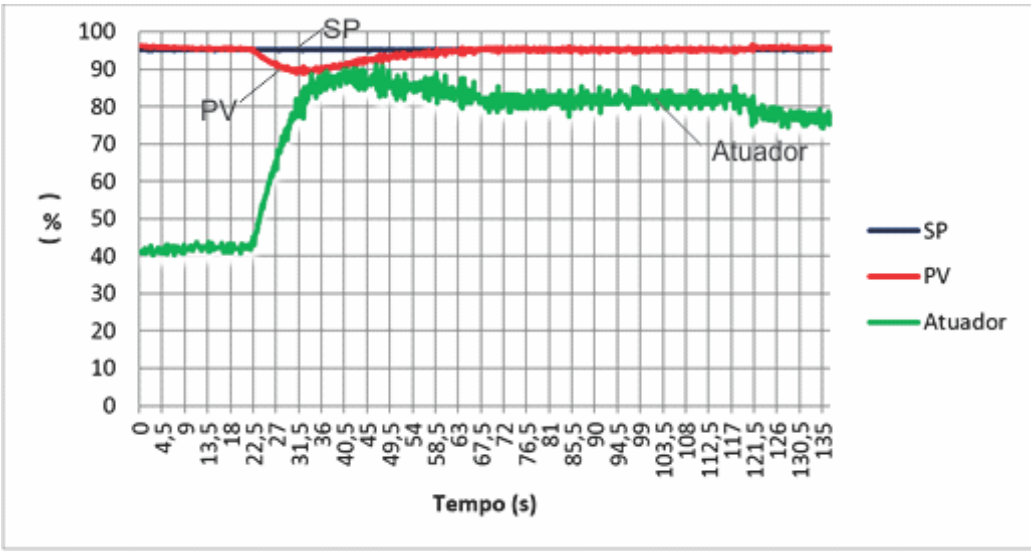


Figura 24 – Variação da vazão de saída através da válvula de descida “V”.

4.4.4 Ensaios com sistema distribuído

Para os ensaios com sistema distribuído foram usadas duas placas de avaliação Stellaris® LM3S8962 iguais rodando a Plataforma 3C, mas cada qual com um *script* diferente indicados no Apêndice B. O objetivo é provar as capacidades de comunicação e de distribuição de controle entre equipamentos através de uma rede digital. As placas foram interconectadas através de uma rede CAN. As funções de comunicação implementadas no módulo n3c (network 3C) foram usadas para enviar o valor do nível da placa “PA”, conectada ao sensor de nível “S”, para a placa “PB”, conectada ao atuador e ao potenciômetro de ajuste de nível.

A placa “PB” recebe o valor de nível da placa “PA”, usa uma de suas entradas analógicas para fazer a leitura do potenciômetro de ajuste de nível, executa o script de controle com algoritmo PID e aplica o sinal de controle em uma saída PWM. A Figura 25 mostra a interconexão entre as placas usando um cabo *flat* de 10 vias (embora apenas duas vias sejam necessárias).

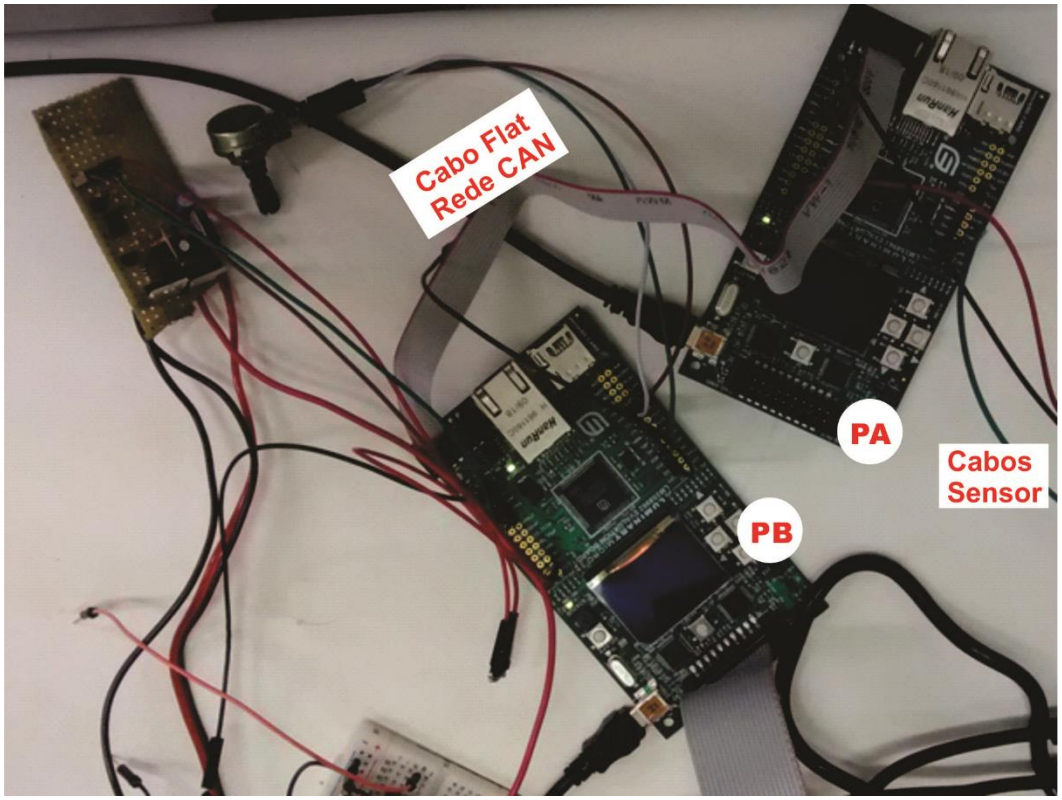


Figura 25 – Duas placas interconectadas através de rede CAN para efetuar o controle.

4.4.4.1 Primeiro ensaio distribuído

Com o sistema estabilizado em 23% (SP e PV) e a válvula de descida "V" completamente aberta, foi feita a mudança do SP para 51%. Após a estabilização foi alterado novamente o SP para 89% conforme Figura 26. O comportamento foi também monitorado por uma aquisição, direta, dos sinais elétricos no potenciômetro e no sensor (SP e PV) via osciloscópio.

Figura 27 (SP:CH2 ciano, PV:CH1 amarelo) para uma comparação com os resultados fornecidos pelo algoritmo. O sinal do atuador não foi medido no osciloscópio devido à limitação do número de canais do mesmo. A ausência da aquisição do osciloscópio nos ensaios com sistema centralizado justifica-se pelo fato daqueles serem mais simples. Devido à falta de pré-processamento as escalas do SP e da PV no osciloscópio são ligeiramente diferentes (offset do sensor de nível) e, por isso, foram mantidos separados para facilitar a visualização.

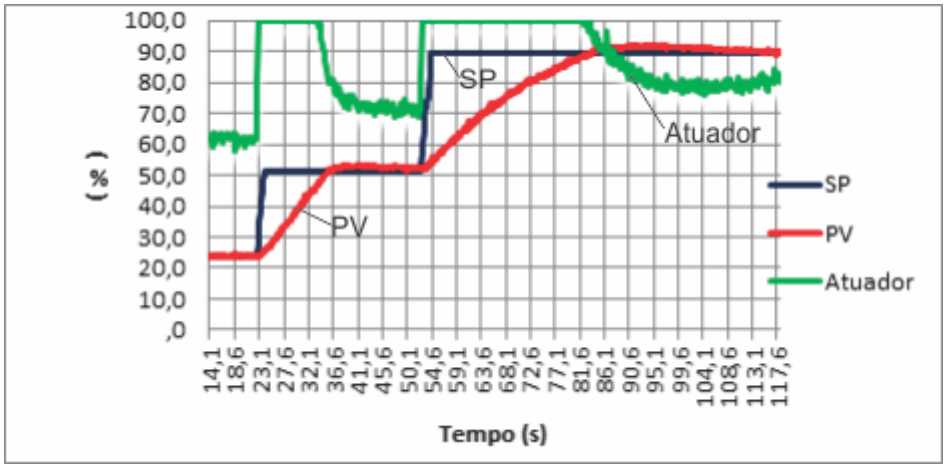


Figura 26 – Mudança de SP de 23% para 51% e em seguida para 89%.

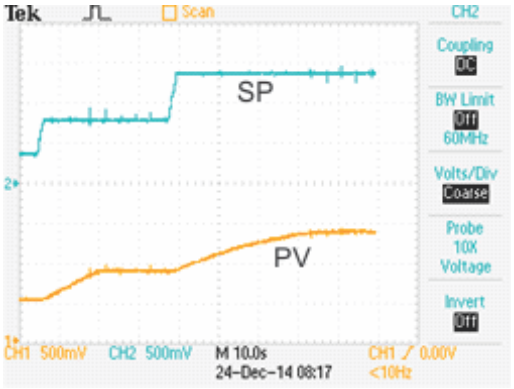


Figura 27 – Mudança de SP de 23% para 51% eem seguida para 89% visto pelo osciloscópio (SP:CH2 ciano; PV:CH1 amarelo).

Comparando este resultado com aquele da Figura 21 (primeiro ensaio do sistema centralizado) pode-se concluir que a distribuição do controle (ou ao menos da aquisição do sinal de nível) não alterou o comportamento do sistema (tempo de resposta e estabilização).

4.4.4.2 Segundo ensaio distribuído

O objetivo deste ensaio é testar o sistema quando reduzido o valor do SP. Notar que a redução do nível na planta ocorre sempre por gravidade através da válvula de descida “V”. Com o sistema estabilizado e o SP em 89% foi feita a mudança do SP para 30% conforme a Figura 28 e pelo osciloscópio na Figura 29.

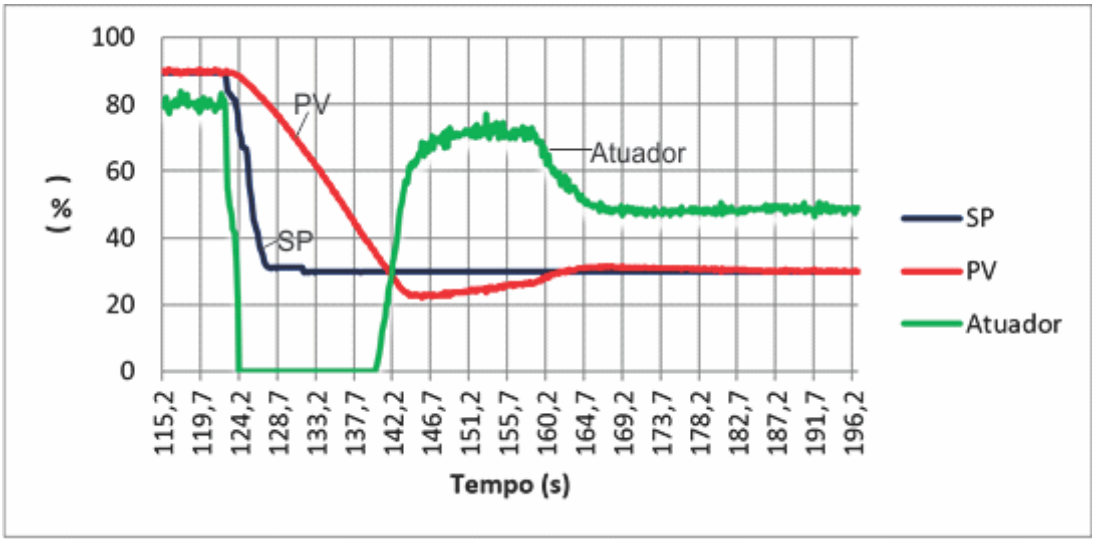


Figura 28 – Redução do SP de 89% para 30%.

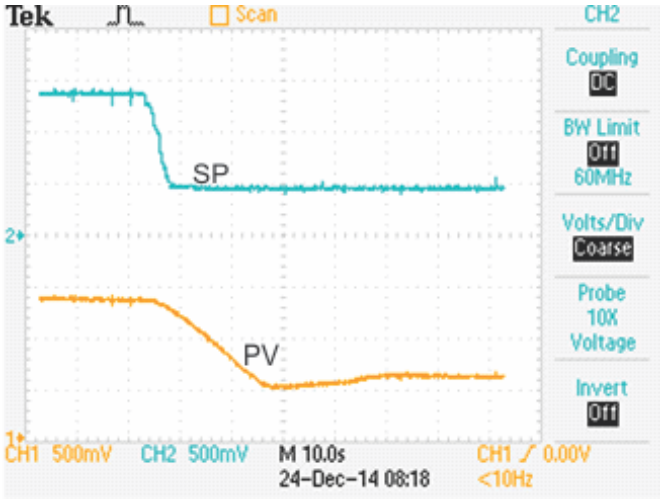


Figura 29 – Redução do SP de 89% para 30%
visto pelo osciloscópio (SP:CH2 ciano; PV:CH1 amarelo).

4.4.4.3 Terceiro ensaio distribuído

Neste ensaio o objetivo é analisar o comportamento do sistema considerando-se um aumento da vazão de saída. Com o sistema estabilizado em 46% foi efetuado um pequeno fechamento da válvula "V" e, sem aguardar a estabilização, ela foi completamente aberta em seguida. Na Figura 30 e Figura 31 é mostrado este comportamento.

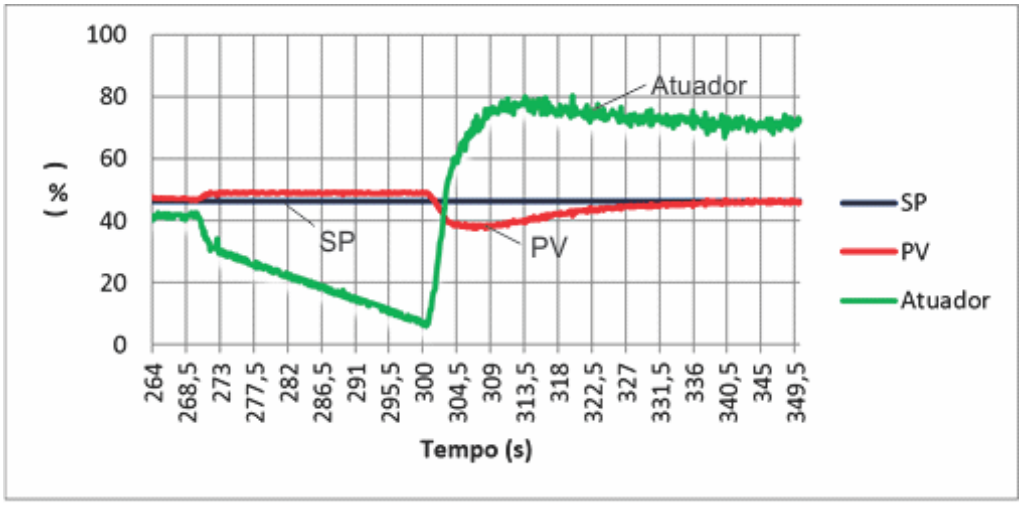


Figura 30 – Pequeno fechamento da válvula "V" seguido de abertura total da mesma.

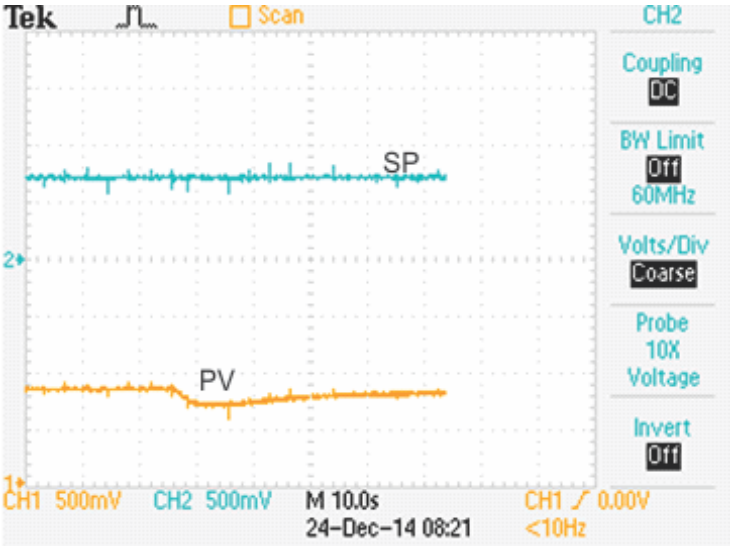


Figura 31 – Pequeno fechamento da válvula "V" seguido de abertura total da mesma com monitoração pelo osciloscópio (SP:CH2 ciano; PV:CH1 laranja).

4.5 Testes de desempenho

Alguns testes de desempenho foram feitos com o auxílio de um frequencímetro convencional para definir os limites máximos de velocidade da Plataforma 3C no kit de desenvolvimento Stellaris.

4.5.1 Geração de onda quadrada

No primeiro teste foi feito um loop infinito para se medir a máxima frequência gerada atuando-se em um pino do microcontrolador por meio da interface h3c.

```
while true do
      h3c.dOut(17, 1) -- coloca a saída 17 em nível alto
      h3c.dOut(17, 0) -- coloca a saída 17 em nível baixo
end
```

No exemplo acima é obtida uma onda quadrada de aproximadamente 6,5kHz para o microcontrolador (Cortex-M3) do kit de desenvolvimento, executando a 50MHz. Uma pequena otimização no *script* permite quase sextuplicar esta frequência (37,8kHz) devido à busca do nome na tabela de módulos ser feita uma única vez:

```
o=h3c.dOut
while true do
      o(17, 1) -- coloca a saída 17 em nível alto
      o(17, 0) -- coloca a saída 17 em nível baixo
end
```

Para fins de comparação com código nativo, foi feita uma substituição da chamada da máquina virtual Lua com o *script* por um código nativo equivalente ao do *script* (mesma estrutura de tarefa e prioridades), consegue-se 922kHz, ou seja 24,39 vezes mais rápido em código nativo.

4.5.2 Desempenho do PID

O código do Apêndice A usado pelo experimento de controle de nível foi alterado para permitir a medição do seu tempo médio de execução. Adicionou-se uma variável para obter o tempo inicial, outra para contar o número de ciclos executados e foi comentado (removido da execução) o tempo de delay “o3c.dms(50)”. Ao final dos ciclos definidos uma média de tempo por ciclos foi calculada:

```
...
elapsed = o3c.ems
start =elapsed()
cycles = 0
while true do
  pid.pv = getLevel(ha)
  pid.sp = getSetPoint(ha)
  runPid(pid)
--  o3c.dms(50)
  if (duty ~= pid.out) then
      duty = pid.out
    hao(2, duty)
  end
  cycles = cycles+1
  if cycles==20000 then
    start = elapsed()-start
    print('cycles: ' .. cycles .. ' time: ' .. start .. ' mean:' .. start/cycles ..
'\r')
    cycles = 0
    start = elapsed()
  end
end
```

Saída do script (media em ms/ciclo):

cycles: 20000 time: 11550 mean:0.57749998569489

cycles: 20000 time: 11549 mean:0.57744997739792

cycles: 20000 time: 11549 mean:0.57744997739792

cycles: 20000 time: 11549 mean:0.57744997739792

...

4.5.3 *Jitter* do PID

Novamente, o código do Apêndice A usado pelo experimento de controle de nível foi alterado para permitir a medição do *jitter* de execução através de um pino de saída digital. O trecho do algoritmo alterado foi o seguinte:

```
  ...
ow=h3c.dOut
while true do
  pid.pv = getLevel()
  pid.sp = getSetPoint()
  runPid(pid)
  ow(17,0)  -- coloca o pino em nível baixo
  o3c.dms(50)
  ow(17,1)  -- coloca o pino em nível alto
  ...
-- comentado teste que executava o print apenas a cada 3 iterações (150ms)
--  outputprint = outputprint + 1
-- if outputprint == 3 then
print(pid.sp .. ' '.. pid.pv .. ' ' .. pid.out .. ' ' .. pid.k .. ' ' .. pid.resetRate .. '\r')
-- outputprint = 0
-- end
```

Conectado este pino em um osciloscópio pôde-se registrar a variação da execução do algoritmo pela largura de pulso do nível alto. Sem haver mudança nos valores dos parâmetros da função *print* a variação máxima de execução foi de 53µs. Contudo, quando houve mudança frequente nos valores dos parâmetros de *print*, a variação máxima chegou a 1400µs, quase três vezes a média de execução calculada no teste anterior.

4.6 Conclusão

Neste capítulo foram apresentados os resultados experimentais que mostram as capacidades e limitações da Plataforma 3C. Os dois primeiros experimentos demostraram os atrasos decorrentes do processamento da Plataforma 3C durante a execução dos respectivos *scripts*. Também mostraram que o *jitter* foi aceitável (menor que 10% do tempo de execução) para aqueles *scripts* devido a provável ausência de alocações e, consequentemente, de chamadas do **GC** durante a execução do script.

No experimento seguinte a Plataforma 3C implementa um controlador PI convencional para controle de nível. As aplicações usando controladores PID (e suas variantes) são bastante usuais na indústria. É muito importante que a Plataforma 3C seja capaz de executar bem as estratégias de controle mais comuns. Este experimento mostrou que tanto com controle centralizado quanto com a distribuição de parte da aplicação, todas as curvas de resposta típicas desse tipo de controle foram obtidas nos diversos cenários. Ainda assim, o *jitter* é uma informação importante para a análise dos especialistas da área de controle sobre esta viabilidade. Não é possível afirmar que ele não ocorra porque não se pode prever o algoritmo que o usuário irá usar, neste experimento ele não ocorreu quando foi retirada a função *print*, que mostra as variáveis internas, pois não houveram pedidos de alocação. O experimento 4.5.3 demonstra que o *jitter* é dependente do *script* usado, podendo ser desprezível ou extremamente relevante. Além do simples atraso de execução é necessário muito cuidado com outros fatores que mudam também a frequência de execução de um algoritmo. Tomando-se por base o simples PID daquele experimento, como se confiou na frequência de execução do algoritmo cadenciado a cada 50ms por “o3c.dms(50)”, sua diminuição (aumento do período decorrente do *jitter*) reduz o ganho do derivativo e aumenta o ganho do integral. É possível contornar isso usando um relógio do sistema ou adotando-se uma execução garantida da frequência do loop principal. Por meio de funções do sistema operacional é também possível aguardar um tempo variável de forma a compensar iterações

mais lentas, ao custo de implementar uma nova interface no módulo o3c e fixar um período de execução superior a algumas vezes a execução mais rápida encontrada. Alguns fabricantes de PLC, por meio da interface de programação, deixam parecer que é feita uma execução especulativa de alguns ciclos e o sistema sugere, com alguma margem, um valor confiável mínimo de execução para um dado algoritmo do usuário. A título de exemplo, no software do PLC ControlLogix o usuário pode especificar o período máximo de execução da Ladder. Se este valor for ultrapassado o controlador entra em modo de falha e registra o tempo que foi ultrapassado.

Também importante, a velocidade máxima de execução do experimento 4.5.1 serve principalmente para se descartar o uso com determinados requisitos mínimos de desempenho estabelecidos. O desempenho do algoritmo do PID do experimento 4.5.2 foi promissor, vários transmissores do sistema de referência possuem PIDs executando na casa das dezenas de milissegundos (SMAR, 2010), este ficou bem abaixo de um milissegundo. A técnica de estabelecimento de um máximo período controlado pelo sistema operacional permitiria uma boa margem de segurança para uma execução com frequência constante. Vale notar que tais transmissores possuem clocks com baixas velocidades e em alguns casos com capacidade de execução inferiores aos do kit usado, para redução de consumo e adequação de normas de segurança. Mesmo assim, se for feito um cálculo para reduzir para 10x esta performance (clock de 50MHz para 5MHz), ou seja, frequência de execução a cada 5ms ainda teria-se margem para chegar a frequência de execução garantida a cada 15ms (3x segundo o experimento 4.5.3), uma velocidade superior a de vários equipamentos existentes.

5 CONCLUSÃO

Os dois experimentos de leitura e escritas digitais mostram os atrasos decorrentes do processamento da Plataforma 3C e o baixo *jitter* para *scripts* simples justificada pela ausência de alocações e consequente inoperância do **GC**. No experimento seguinte, o controlador PI codificado no *script* para o controle de nível em uma planta é exercitado sob diversas circunstancias: degrau de entrada e degrau na saída. Os mesmos testes foram feitos com sistema concentrado e sistema distribuído usando dois controladores transmitindo informação pela rede CAN. Os gráficos de cada ensaio foram gerados e apresentaram as curvas típicas de tais sistemas. Os testes de desempenho cumpriram o papel de identificar os limites da Plataforma 3C. O teste com geração de onda quadrada mostra claramente que a forma de escrever o *script* é extremamente relevante para a sua execução uma vez que o compilador de Lua não realiza otimizações, aparentemente, simples. A futura ferramenta de geração de código Lua a partir de outra linguagem como Ladder, talvez possa gerar um código melhor ou pré otimizar um código do usuário nos casos mais comuns. O desempenho do PID implementado no *script* foi superior ao de vários transmissores digitais que gastam de 8 a 25ms por ciclo. O *jitter* do PID melhorou muito quando removida a chamada da função *print*, isto é, ficou abaixo de 60µs, aproximadamente 10% da média de execução. Tendo chamadas de *print*, ou outras construções no *script* que demandem alocações frequentes, o **GC** tem mais trabalho e o *jitter* pode superar em algumas vezes o tempo médio do algoritmo. Existem diversos paliativos para reduzir o *jitter* causado pelo **GC**, dentre eles executá-lo frequentemente e por tempo determinado como sugerido por Kalogirou (2014), desabilitá-lo em trechos críticos ou ainda reservar tempo extra no ciclo de execução esperado com o auxílio de rotinas do escalonador do sistema operacional para que no evento de variação do tempo de execução este possa ser absorvido.

A Plataforma 3C trás uma nova abordagem para solucionar vários problemas encontrados nos sistemas atuais. Como afirmado anteriormente, embora não seja uma solução ideal (não atende a processos de batelada), ela resolve os principais problemas que motivaram o seu desenvolvimento como: interoperabilidade, portabilidade de *scripts* do usuário, algoritmos de controle flexíveis e personalizados, formação técnica facilitada, participação mais efetiva da academia no desenvolvimento de novas estratégias de controle, redução de impactos ambientais pela adoção de novas estratégias de controle ou da redução dos custos com hardware e com desenvolvimento por parte dos fabricantes. Mesmo que não seja

adotada, algumas dessas soluções podem fomentar a criação de novos padrões ou o aprimoramento de padrões existentes.

As perspectivas para trabalhos futuros e evolução são vastas, neste momento vislumbra-se, em especial, a seguinte: criar uma ferramenta de configuração avançada, que permita a entrada de outras linguagens de programação usuais, coletar informações estatísticas sobre a execução em parâmetros da MIB e sugira um ciclo seguro compensando o *jitter* ou encontre uma solução para resolvê-lo. Tal ferramenta é fundamental para aceitação da Plataforma 3C por parte dos usuários finais. Há também diversas melhorias necessárias para a maturidade da Plataforma 3C destacando-se: mecanismo para tratamento comum de escalas, definições para sincronismo de tempo, análises sobre determinismo de execução, segurança contra invasão (acessos não autorizados) e *port* da Plataforma 3C para outras arquiteturas de hardware e protocolos de rede de forma a exercitar as interfaces de portabilidade e identificar problemas.

Finalmente, existe uma enorme gama de aplicações nas quais a Plataforma 3C pode ser aplicada dando maior flexibilidade e funcionalidade. Por exemplo, em sistemas de automação residêncial e predial, em rastreadores veiculares e automação de sistemas compostos por diversas máquinas agrícolas no campo. O setor de energia poderia se beneficiar das características da Plataforma 3C na implementação de *smart grids* pois pode-se ter atuação local, comunicação remota entre equipamentos para coordenação além da atualização remota de algoritmos. Este trabalho é apenas o ponto de partida da Plataforma 3C, sua evolução pode-se dar em diferentes frentes devido à característica de desenvolvimento de código fonte aberto.

.

REFERÊNCIAS

A Linguagem de Programação Lua. **Lua**, 2014. Disponivel em: <http://www.lua.org/portugues.html>. Acesso em: 18 mar. 2014.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR IEC 60079-14**. [S.l.]. 2006.

BOLZ, C. F.; RIGO, A. **How to not write Virtual Machines for Dynamic Languages**. In: Proceedings of the 3rd Workshop on Dynamic Languages and Applications, 2007, [S.l.].

BROWNING, J. B. **Open-Source Solutions in Education:** Theory and Practice. [S.l.]: Informing Science Press, 2010.

ELUA. **eLua**, 2014. Disponivel em: <http://www.eluaproject.net/>. Acesso em: 20 set. 2014.

FIELDBUS FOUNDATION. FOUNDATION(TM) Specification Function Block Application Process, v. Part 4, n. FS 1.2, 2008. FF-893.

GAJ, P.; JASPERNEITE, J.; FELSER, M. Computer Communication Within Industrial Distributed Environment. **IEEE Transactions on Industrial Informatics**, v. 9, n. 1, p. 182-189, 2013.

GOLDBERG, I.; WAGNER, D.; THOMAS, R.; BREWER, E. A. **A Secure Environment for Untrusted Helper Applications Confining the Wily Hackeriscrete-Event System Simulation**. In: Proceedings of the Sixth USENIX UNIX Security Symposium, 1996, San Jose, California.

HANNA, Y. **SLEDE:** Lightweight Verification of Sensor Network Security Protocol Implementations. In: Proc. of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering., 2007, New York. p. 591-594.

IERUSALIMSCHY, R. **Programming in Lua**. 3rd. ed. Rio de Janeiro: Feisty Duck Digital, 2013.

IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H.; CELES, W. **The evolution of an extension language:** A history of Lua. In: V Simpósio Brasileiro de Linguagens de Programação, 2001, Curitiba. p. 14-28.

IV Machine or Forth Machine. **ZSerge**, 2012. Disponivel em: <http://zserge.bitbucket.org/j1vm.html>. Acesso em: 27 outubro 2014.

JUNG, K.; BROWN, A. Beginning Lua Programming. Indianapolis: Wiley Publishing, Inc., 2007. Cap. 10, p. 303.

KAFFE FAQ. **GitHub**, 2007. Disponivel em: <https://github.com/kaffe/kaffe/blob/master/FAQ/FAQ.arm>. Acesso em: 26 outubro 2014.

KALOGIROU, C. Predictable garbage collection with Lua. **kalogirou**, 2014. Disponivel em: <http://kalogirou.net/2011/07/23/predictable-garbage-collection-with-lua/>. Acesso em: 23 abr. 2014.

KANTER, A. S.; BORLAND, R.; MOURICE, B. et al. **The Importance of Using Open Source Technologies and Common Standards for Interoperability within eHealth**. In: Health Information Technology in the International Context, 2012, [S.l.] Emerald Group Publishing Limited. p. 189-204.

KLOTZBUECHER, M.; BRUYNINCKX, H. **Hard real-time Control and Coordination of Robot Tasks using Lua**. In: 13th Real-Time Linux Workshop, 2011, Prague. p. 1-7.

LUA.ORG. News. **Lua**, 2014. Disponivel em: <http://www.lua.org/news.html>. Acesso em: 10 dez. 2014.

MARQUET, K.; COURBOT, A.; GRIMAUD, G.; SIMPLOT-RYL, D. **Ahead of time deployment in ROM of a Java-OS.** In: 2nd International Conference on Embedded Software and System (ICESS 2005), 2004, Xi'an ICESS.

MICRO Python. **Micro Python**, 2014. Disponivel em: <https://micropython.org/>. Acesso em: 20 nov. 2014.

MONTPELIER OPEN SOURCE. The Benefits of Open Source. **Montpelier Open Source**, 2015. Disponivel em: <http://www.montpelieropensource.com/benefits.htm#phil>. Acesso em: 16 mar. 2015.

PANTON, R. P.; TORRISI, N.; BRANDÃO, D. **An open and non-proprietary device description for fieldbus devices for public IP networks**. In: 5th IEEE International Conference on Industrial Informatics, 2007, Vienna. p. 189-194.

PILAT, M. L.; OPPACHER, F. **Robotic Control Using Hierarchical Genetic Programming**. In: GECCO, 2000, [S.l.]. p. 190-194.

POSTEL, J. User Datagram Protocol, RFC 768, Agosto 1980.

POSTEL, J. INTERNET PROTOCOL, RFC 791, Setembro 1981.

PREVELAKIS, V.; SPINELLIS, D. **Sandboxing Applications**. In: USENIX 2001 Technical Conference Proceedings: FreeNIX Track, 2001, Berkeley USENIX Association. p. 119-125.

PYMITE VM. **PyMite**, 2014. Disponivel em: <https://wiki.python.org/moin/PyMite>. Acesso em: 2 Dezembro 2014.

SAUTER, T. The three generations of field-level networks - Evolution and compatibility issues. **IEEE Trans. Ind. Electron.**, v. 57, n. 11, p. 3585-3595, Nov. 2010.

SBRK. **Wikipedia**, 2015. Disponivel em: <http://en.wikipedia.org/wiki/Sbrk>. Acesso em: 8 Fevereiro 2015.

SCOTT, M. L. **Programming Language Pragmatics**. 3. ed. Burlington: Elsevier, 2009.

SHAHRAEINI, M.; JAVIDI, M. H.; GHAZIZADEH, M. S. Comparison Between Communication Infrastructures of Centralized and Decentralized Wide Area Measurement Systems. **IEEE Transactions On Smart Grid**, v. 2, n. 1, p. 206-211, 2011.

SHAW, J. A. **The PID Control Algorithm - How it works, how to tune it, and how to use it**. 2nd. ed. Rochester: [s.n.], 2006.

SHAYLOR, N.; SIMON, D. N.; BUSH, W. R. **A java virtual machine architecture for very small devices**. In: ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, 2003, Uppsala ACMPress. p. 34-41.

SMAR. Introduction to function block application. **Smar**, p. 58, 2010. Disponivel em: <http://www.smar.com/PDFs/manuals/FBLCLAFFME.pdf>. Acesso em: 30 mar. 2015.

TANENBAUM, A. S. Computer Networks. In: TANEMBAUM, A. S. **Computer Networks**. 4. ed. [S.l.]: Prentice Hall, 2002. p. 221-222. ISBN 0130661023.

VREELAND, S.; MITSCHKE, S. **Practical Importance of the FOUNDATION Fieldbus Interoperability Test System**. In: TECHNICAL PAPERS- ISA, 1999, Chicago Instrument Society of America. p. 133-142.

WILSON, P. R.; JOHNSTONE, M. S. **Truly Real-Time Non-Copying Garbage Collection**. In: Proceedings of OOPSLA/ECOOP’93 Workshop on Garbage Collection in Object-Oriented Systems, 1993, [S.l.].

GLOSSÁRIO

Atuador: dispositivo de controle com capacidade de atuação em um processo.

Bloco flexível:bloco funcional que permite definição de seu algoritmo interno.

Bloco funcional:algoritmo fixo que implementa uma dada função de controle (PID, Lead, Lag, Seletor, Média, Raiz, Funções lógicas etc.).

Dispositivo: neste contexto, equipamento eletrônico que faz parte de uma rede de comunicação (um nó de uma rede).

Fragmentação: em computação é o termo dado ao espalhamento de blocos de recursos, normalmente memória RAM, causado por alocações e liberações de recursos de forma descontínua (fragmentada). De tal sorte que a soma dos blocos livres não pode ser unificada em um único bloco contíguo devido aos "fragmentos" intermediários em uso.

Ladder: linguagem de programação lógica que imita instalações elétricas com contatos e bobinas. Originalmente desenvolvida para facilitar o uso de CLPs por eletricistas.

Plataforma: ambiente de software e/ou hardware usado como base para uma determinada aplicação.

realloc(): função padrão da biblioteca ANSI C que permite alocar e "realocar" (alterar tamanho de) blocos de memória.

Sistema: conjunto formado por dispositivos eletrônicos capazes de processar informação a partir de um programa e gerar resultados de alguma forma.

Transmissor: neste contexto, dispositivo com capacidade de medir alguma grandeza física e transmitir esta informação através de algum meio analógico ou digital.

APÊNDICE A – *SCRIPT* PARA O EXPERIMENTO 4.4.3

```
local function runPid(pid)
  error = 0
  pvD=0
  outTmp=0
  if pid.en==false then
    pid.lpv = pid.pv
    pid.fbk = 0
    pid.out = pid.lout;
  else
    pvD = pid.pv + (pid.pv - pid.lpv)*pid.td*pid.rptmin;
    pid.lpv = pid.pv
    error = pid.sp - pvD
    pid.lerror = error
--    error = -error;
    outTmp = error * pid.k + pid.fbk;
    if (outTmp > pid.max) then
      outTmp = pid.max
    else
      if (outTmp < pid.min) then
        outTmp = pid.min
      end
    end
    pid.out = outTmp;
    pid.lout = outTmp;
    pid.fbk = pid.fbk + (outTmp-pid.fbk)*pid.resetRate/pid.rptmin;
    if pid.fbk > 100000 then
      pid.fbk = 100000
    else
      if pid.fbk < -100000 then
        pid.fbk = -100000
      end
    end
  end
end

local function getLevel()
  v = ((h3c.aIn(0)-70)*100)/(430);
  if v>100 then
    v = 100
  end
  if (v<0) then
    v = 0
  end
  return v
end

local function getSetPoint()
  v = ((h3c.aIn(2))*100)/(1023);
  if v>100 then
    v = 100
  end
  if (v<0) then
    v = 0
  end
  return v
end

local function getSelectMult()
  m = 1
  if (h3c.dIn(1)==0) then
    m = 10
  end
  return m
end
```

```
pid = {}
pid.en = true
pid.k=4.84
pid.resetRate = 4.18
pid.td = 0
pid.lout = 0
pid.fbk = 0
pid.rptmin = 60*20
pid.out = 0
pid.max = 100
pid.min = 0
pid.pv = 0
pid.lpv = 0
pid.sp = 0
pid.lerror=0
duty = 0
lastpv = 0
pid.sp = 5

pid.en = false
pid.pv = getLevel()
pid.sp = getSetPoint()
runPid(pid)
duty = pid.out
h3c.aOut(2, duty)
pid.en = true
outputprint = 0;
while true do
  pid.pv = getLevel()
  pid.sp = getSetPoint()
  runPid(pid)
  o3c.dms(50)
  if (duty ~= pid.out) then
      duty = pid.out
    h3c.aOut(2, duty)
  end
  outputprint = outputprint + 1
  if outputprint == 3 then
    print(pid.sp .. ' '.. pid.pv .. ' ' .. pid.out .. ' ' .. pid.k .. ' ' .. pid.resetRate .. '\r')
    outputprint = 0
  end

  if (h3c.dIn(2)==0) then
    pid.k = pid.k + 0.2*getSelectMult()
    o3c.dms(50)
    while (h3c.dIn(2)==0) do
    end
  end
  if (h3c.dIn(3)==0) then
    pid.k = pid.k - 0.2*getSelectMult()
    if (pid.k<0) then
      pid.k = 0
    end
    o3c.dms(50)
    while (h3c.dIn(3)==0) do
    end
  end
  if (h3c.dIn(4)==0) then
    pid.resetRate = pid.resetRate + 0.1*getSelectMult()
    o3c.dms(50)
    while (h3c.dIn(4)==0) do
    end
  end
  if (h3c.dIn(5)==0) then
    pid.resetRate = pid.resetRate - 0.1*getSelectMult()
    if pid.resetRate<0 then
      pid.resetRate = 0
    end
    o3c.dms(50)
    while (h3c.dIn(5)==0) do
    end
  end
end
```

APÊNDICE B – *SCRIPT* PARA O EXPERIMENTO 4.4.4

```
-----------------------------------------------------------------
-- script do atuador (recebe a informação de nível via rede e atua na bomba)
-----------------------------------------------------------------

local function runPid(pid)
  error = 0
  pvD=0
  outTmp=0
  if pid.en==false then
    pid.lpv = pid.pv
    pid.fbk = 0
    pid.out = pid.lout;
  else
    pvD = pid.pv + (pid.pv - pid.lpv)*pid.td*pid.rptmin;
    pid.lpv = pid.pv
    error = pid.sp - pvD
    pid.lerror = error
--    error = -error;
    outTmp = error * pid.k + pid.fbk;
    if (outTmp > pid.max) then
      outTmp = pid.max
    else
      if (outTmp < pid.min) then
        outTmp = pid.min
      end
    end
    pid.out = outTmp;
    pid.lout = outTmp;
    pid.fbk = pid.fbk + (outTmp-pid.fbk)*pid.resetRate/pid.rptmin;
    if pid.fbk > 100000 then
      pid.fbk = 100000
    else
      if pid.fbk < -100000 then
        pid.fbk = -100000
      end
    end
  end
end



local function getLevel(plevel)
  r,id,data = n3c.rx()-- obtem o nível pela rede, enviado pelo transmissor
  if r==true then
    if (id == 10) then
      plevel = data
    end
  end
  return plevel
end



local function getSetPoint()
  v = ((h3c.aIn(2))*100)/(1023);
  if v>100 then
    v = 100
  end
  if (v<0) then
    v = 0
  end
  return v
end
```

```
local function getSelectMult()
  m = 1
  if (h3c.dIn(1)==0) then
    m = 10
  end
  return m
end



idl = 10
n3c.listen(idl)


pid = {}
pid.en = true
pid.k=4.84
pid.resetRate = 4.18
pid.td = 0
pid.lout = 0
pid.fbk = 0
pid.rptmin = 60*20
pid.out = 0
pid.max = 100
pid.min = 0
pid.pv = 0
pid.lpv = 0
pid.sp = 0
pid.lerror=0
duty = 0
lastpv = 0
pid.sp = 5


pid.en = false
lastlevel=0
lastlevel = getLevel(lastlevel)
pid.pv=lastlevel
pid.sp = getSetPoint()
runPid(pid)
duty = pid.out
h3c.aOut(2, duty)
pid.en = true


outputprint = 0;
while true do
  lastlevel = getLevel(lastlevel)
  pid.pv=lastlevel
  pid.sp = getSetPoint()
  runPid(pid)
  o3c.dms(50)
  if (duty ~= pid.out) then
      duty = pid.out
    h3c.aOut(2, duty)
  end
  outputprint = outputprint + 1
  if outputprint == 3 then
    print(pid.sp .. ' '.. pid.pv .. ' ' .. pid.out .. ' ' .. pid.k .. ' ' .. pid.resetRate .. '\r')
    outputprint = 0
  end


  if (h3c.dIn(2)==0) then
    pid.k = pid.k + 0.2*getSelectMult()
    o3c.dms(50)
    while (h3c.dIn(2)==0) do
    end
  end


  if (h3c.dIn(3)==0) then
    pid.k = pid.k - 0.2*getSelectMult()
    if (pid.k<0) then
```

```
      pid.k = 0
    end
    o3c.dms(50)
    while (h3c.dIn(3)==0) do
    end
  end



  if (h3c.dIn(4)==0) then
    pid.resetRate = pid.resetRate + 0.1*getSelectMult()
    o3c.dms(50)
    while (h3c.dIn(4)==0) do
    end
  end



  if (h3c.dIn(5)==0) then
    pid.resetRate = pid.resetRate - 0.1*getSelectMult()
    if pid.resetRate<0 then
      pid.resetRate = 0
    end
    o3c.dms(50)
    while (h3c.dIn(5)==0) do
    end
  end
end
```

```
-----------------------------------------------------------------
-- script do transmissor (lê do sensor e publica a informação de nível na rede)
-----------------------------------------------------------------

local function getLevel()
  v = ((h3c.aIn(0)-70)*100)/(430);
  if v>100 then
    v = 100
  end
  if (v<0) then
    v = 0
  end
  return v
end



id = 10
outputprint = 0
while true do
  level = getLevel()
  n3c.tx(id, level)
  o3c.dms(50)
  outputprint = outputprint + 1
  if outputprint == 3 then
    print(level .. '\r')
    outputprint = 0
  end

end
```