Estudo da utilização de um sistema *RTOS* para desenvolvimento de um *software* executivo para CLPs

Leonardo Diego Pacheco[1]
Marcos Zuccolotto[2]

Resumo

Atualmente, encontram-se, no mercado, a um custo competitivo, algumas linhas de microcontroladores com alta capacidade de processamento, gerenciamento avançado de memória e suporte a interfaces e protocolos de rede, bem como sistemas operacionais embarcados que reduzem drasticamente o tempo de desenvolvimento de novos produtos. Buscando apropriar-se desta tecnologia, este trabalho tem por objetivo a implementação de um *software* executivo para CLPs, baseado em um sistema operacional de tempo real. Para tanto, exploram-se os conceitos de sistemas operacionais multitarefa e de tempo real, descrevem-se o desenvolvimento e a codificação de um sistema executivo para CLPs com suporte a rede de comunicação e, por fim, apresenta-se uma análise crítica do uso destas tecnologias.

Palavras-chave: controlador lógico programável, *RTOS*, sistemas multitarefa.

Abstract

Nowadays, it is available a line of microcontrollers with a set of features, like high processing power, advanced memory management, network support and embedded multitask operating systems, which reduce the period of development of new products. Thus, the goal of the present work is to use this new technology to develop a software for PLCs, based on a real time operating system. This paper describes the design of a PLCs operating system with network support based on RTOS system. In addition, it presents the basic concepts of multitask and multithread and an analisys of the use of these technologies.

Keywords: programable logic controller, RTOS, multitask systems.

1 Introdução

Quando os primeiros controladores lógico-programáveis surgiram, apresentavam diversas limitações impostas pela tecnologia da época, como baixa capacidade de processamento, poucas alternativas de conectividade e ferramentas de desenvolvimento que dificultavam a criação de aplicações complexas. O *hardware* tinha um custo elevado, e o *software* da aplicação, além de executar o controle do processo, era responsável por administrar as funções de entrada e saída e os protocolos de comunicação. Os programas aplicativos, em sua maioria escritos em linguagem *assembly,* facilmente produziam um código extenso e complexo, de difícil manutenção, restringindo o uso de CLPs a aplicações simples, como painéis de comando de máquinas ou laços realimentados de uma variável, deixando o controle de processos de grande porte para sistemas dedicados.

Atualmente, a demanda de aplicações utilizando CLPs é muito maior, exigindo do equipamento uma capacidade de execução de algoritmos complexos, como laços de controle multivariável ou *fuzzy*, além da capacidade de intercomunicação com outros sistemas através de redes de comunicação. É possível perceber que os CLPs assemelham-se aos computadores pessoais em capacidade de processamento e de interconexão; no entanto, diferem em confiabilidade.

Com desenvolvimento das tecnologias de *hardware* e *software*, dispõe-se de microcontroladores que, com recursos de gerenciamento de memória, maior desempenho na execução do código, periféricos mais elaborados, suporte a protocolos de rede e, ainda, um *framework* de programação *multitarefa,* podem elevar o conceito de CLP a um novo patamar. As técnicas de programação utilizadas inicialmente não anteciparam

[1] Engenheiro Eletricista, Projetista de Produto Altus S.A. Monografia apresentada para obtenção do diploma de Engenheiro Eletricista na Universidade do Rio do Sinos – UNISINOS. E-mail: binho.pacheco@gmail.com

[2] Mestre em Engenharia Elétrica pela UFRGS. Professor do curso de Engenharia Elétrica da UNISINOS e do curso Técnico em Eletrônica da Fundação Escola Técnica Liberato Salzano Vieira da Cunha. E-mail: marcosz@unisinos.br

essas novas tecnologias, impossibilitando o desenvolvimento de um CLP que explora todos os recursos, atualmente, disponíveis nos microcontroladores e que atendam às demandas atuais.

O paradigma de execução "paralela" de rotinas, em substituição à execução seqüencial, simplifica o projeto dos *softwares*, utiliza eficientemente os recursos oferecidos pela plataforma de *hardware* e oferece melhores condições para a manutenção e expansão das rotinas desenvolvidas. A adoção deste paradigma exige a utilização de sistemas operacionais multitarefa que, em aplicações embarcadas, são denominados *RTOS* (*Real Time Operating Systems*)(2007).

O objetivo deste trabalho é estudar, planejar e implementar um *software* executivo para CLP, utilizando microcontroladores atuais com recursos de *hardware* embarcados de forma a auxiliarem o desenvolvedor no processo de codificação. Também é introduzida a utilização de sistemas operacionais *RTOS*, para o desenvolvimento do executivo minimizando dificuldades de codificação da execução dos processos internos que o CLP deve efetuar e sincronização dos mesmos. O *software* é composto pela arquitetura básica de um executivo de CLP e tem a capacidade de executar uma aplicação de usuário, além disso, apresenta a possibilidade de troca da aplicação.

2 Sistemas *multitasking* e *multithreading*

Segundo Ball (2002), um sistema operacional RTOS é capaz de compartilhar a utilização de um processador por vários programas, respeitando as necessidades temporais de cada um destes programas ou rotinas.

Quando os primeiros sistemas operacionais surgiram, o conceito de *multitasking* ou multitarefa não era conhecido, pois somente um programa era executado, *single task* (tarefa única). A evolução veio através dos sistemas multitarefa capazes de executar vários programas com apenas uma central de processamento (UCP). O sistema operacional executa o trabalho de alternar as tarefas e direcioná-las para o processador. Desta forma, temos a impressão de que as tarefas estão sendo executadas simultaneamente, mas o que realmente ocorre é uma tarefa ocupando exclusivamente o processador em um curto intervalo de tempo, até que o sistema operacional a interrompa, e outra tarefa tome o seu lugar. Esta troca é chamada de *context switching* (chaveamento de contexto).

Existem dois tipos de sistemas operacionais multitarefa: os preemptivos e os não-preemptivos. No caso dos preemptivos, as tarefas não têm controle do seu próprio tempo de execução, ao contrário dos não-preemptivos, em que a tarefa é responsável por devolver a UCP ao sistema operacional.

Os sistemas operacionais *multitask* exigem suporte de *hardware* para sua execução, ou seja, o processador deve oferecer recursos de gerenciamento de memória e de chaveamento de contexto, de tal forma que cada tarefa tenha sua área de memória exclusiva, evitando que uma *task* altere os conteúdos das variáveis que não lhe pertencem. A troca de dados entre *tasks* deve ser gerenciada pelo *RTOS*, como ilustrado na figura 1.

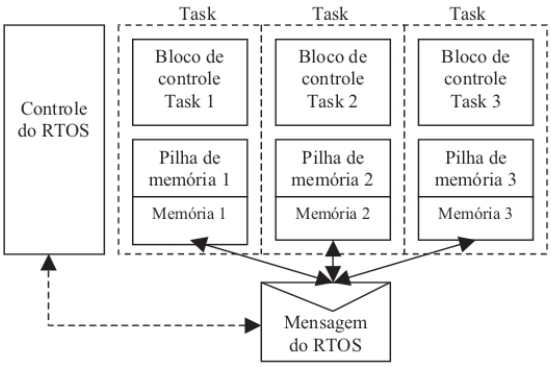

Figura 1 – Modelo *multitasking*

Pode-se estender a idéia de várias "tarefas" sendo executadas em paralelo para a execução de um único programa. Isto implica mudança no planejamento do programa, pois se torna necessário identificar as rotinas que podem ser executadas de maneira concorrente e as estratégias de troca de dados entre as mesmas. Os programas que adotam este paradigma são chamados de *multithreading*. Uma *thread* pode ser entendida como um processo único e seqüencial de um programa.

A principal diferença entre *tasks* e *threads* está no acesso à memória e aos recursos do processador. Todas as *threads* compartilham a memória e os recursos reservados ao programa a que pertencem, como apresentado na figura 2. Assim, uma *thread* é capaz de acessar os dados utilizados por outra *thread,* ou ainda, quando uma *thread* fecha um arquivo, todas as outras *threads* "vêem" o arquivo sendo fechado.

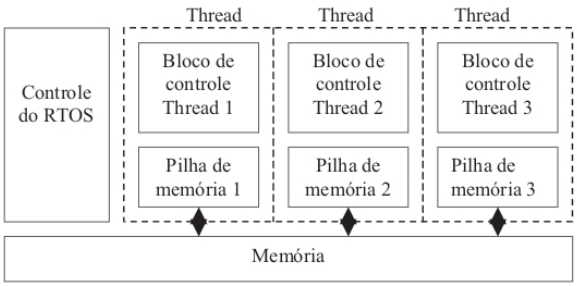

Figura 2 – Modelo *multithreading*

Prassad (1997) aponta uma série de benefícios obtidos com o uso de programação *multithreading*:

– *Threads* são consideradas ideais para programas ou sistemas em que o número de tarefas a serem executadas num dado momento não podem ser determinadas. *Threads* podem ser criadas e destruídas facilmente, aumentando e diminuindo o atendimento das tarefas requisitadas pelos usuários do programa.
– É possível aumentar o desempenho de um programa em sistemas multiprocessados somente com o uso de *threads*. A aplicação pode ser paralelizada e, então, irá executar mais rapidamente que a mesma aplicação *single task*.

– A troca de dados com o meio externo (acesso aos pontos de E/S – entrada e saída – de dados) é mais lenta que o processamento interno. Utilizando *multithread*, é possível otimizar a ocupação do processador, pois enquanto as E/S são acessadas por uma *thread*, que garante o atendimento ao tempo de acesso necessário, outras *threads*, que não dependem das E/S, podem continuar em execução.
– A codificação de um programa pode ser simplificada se este estiver estruturado em pequenos blocos, reduzindo-o a partes menores que possam ser executadas concorrentemente. A definição da estrutura do programa pode ser um pouco mais complexa em sistemas com *threads*, mas o programa resultante tem sua codificação mais clara e aumenta sua modularidade, facilitando sua manutenção e expansão.

Entretanto, esta série de benefícios vem acompanhada por algumas desvantagens e cuidados no planejamento:
– O desenvolvimento de programas *multithread* é mais complexo que o de programas seqüenciais, pois é necessário um cuidado maior na escolha das rotinas a serem paralelizadas, bem como nos processos de sincronização de dados entre essas rotinas.
– A depuração do programa é dificultada, pois agora não existe mais uma execução seqüencial, e sim paralela. A falha que surge em uma *thread* pode ter origem em outra *thread*, ou na sincronização entre diferentes *threads.*
– A sincronização de dados entre *threads* é fundamental, pois um acesso simultâneo de duas (ou mais) *threads* na mesma área de dados pode provocar uma falha grave na execução do programa. Devem-se adotar estratégias de sincronização no gerenciamento de memória como, por exemplo, o uso de semáforos ou caixas de mensagens.
– A utilização de esquemas de sinalização do uso de recursos compartilhados pode acarretar o *deadlock* ou "travamento" da execução de uma *thread.* Isto pode ocorrer quando determinado recurso (memória ou E/S) nunca fica disponível para a *thread* que o necessita.

3 Controladores lógicos programáveis

Os primeiros controladores lógicos programáveis (CLPs) surgiram na década de 60 como forma de atender a uma demanda da indústria automobilística norte-americana que desejava um equivalente computadorizado dos tradicionais painéis de relés com objetivo de reduzir os custos de manutenção.

A função do CLP é efetuar o controle ou supervisão de uma máquina ou processo. Como as regras de controle não são genéricas, é necessário, de alguma maneira, programar estas regras no CLP. Esta programação é chamada de programa aplicativo ou simplesmente aplicativo. A execução deste aplicativo requer uma série de operações, como varredura dos pontos de E/S, tratamento de comunicações com unidades remotas e ações dependentes de solicitações assíncronas (interrupções) originadas no processo. Essas operações são de responsabilidade do sistema operacional do CLP que é denominado (por alguns fabricantes) de sistema executivo ou, de modo sintético, executivo. O processo de execução do aplicativo é denominado ciclo de execução, e o tempo que o CLP gasta para realizar um ciclo de execução é um parâmetro importante de desempenho do equipamento. A figura 3 apresenta um fluxograma típico de um executivo de CLP.

Como explica Georgini (2003), antes da execução cíclica do aplicativo o CLP, por ser um equipamento modular, necessita identificar se a configuração de *hardware* está presente e operando; para tanto, é necessário interpretar e verificar uma estrutura de dados chamada módulo de configuração. Essa estrutura descreve os operandos necessários à execução do aplicativo e os módulos de *hardware* ou elementos de rede que fornecerão os dados dos respectivos operandos. Operando é o nome dado às variáveis do aplicativo, que são responsáveis por armazenar valores dos pontos de E/S e valores intermediários produzidos pela execução do aplicativo.

As operações do executivo podem ser pensadas como tarefas a serem executadas por um sistema multitarefa, e é este o enfoque adotado.

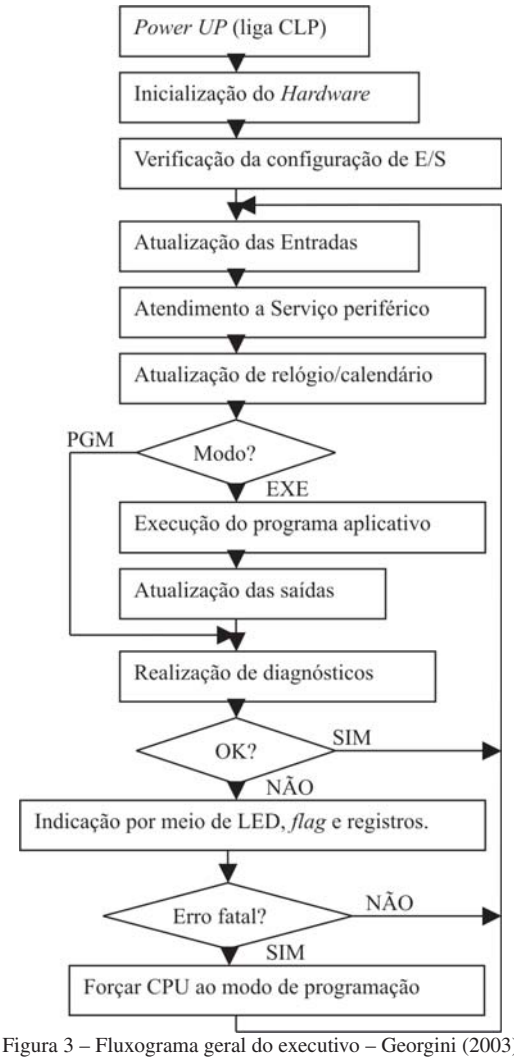

Figura 3 – Fluxograma geral do executivo – Georgini (2003)

4 Plataforma de desenvolvimento

Para o desenvolvimento deste trabalho, foi escolhido o componente SC12, da empresa alemã Beck, que é um microcoprocessador de 16 bits da família 80186, rodando a um *clock* de 20MHz, com 512KB de memória *RAM,* disponíveis como memória principal e 512KB de memória *Flash,* utilizada como memória de massa. Como canais de comunicação, o SC12 possui um dispositivo *ethernet* 10BaseT e duas portas seriais de alta velocidade. Estão disponíveis 14 pinos de entradas e/ou saídas programáveis.

O SC12 contém um sistema operacional *RTOS* que dá suporte à execução de até 35 tarefas simultâneas, 78 temporizadores, semáforos, eventos de grupo e até 10 *mailboxes* (caixas de correio) para troca de mensagens entre as tarefas. Alguns destes recursos são utilizados pelo próprio *RTOS* para gerenciamento, e os demais estão disponíveis para o usuário. O sistema de arquivos utilizado pelas unidades de memória de massa é o FAT16. O *RTOS* permite que até 15 tarefas acessem o sistema de arquivos simultaneamente. O *RTOS* também oferece suporte à rede, implementando o protocolo IPv4, possui camada TCP, comunicações UDP e suporte a ARP, ICMP, IP *Multicasting* e DHCP. É possível abrir até 64 *sockets* em modo servidor ou em cliente. As interfaces internas para dispositivos são de servidor PPP, cliente PPP e *loopback*. As aplicações TCP/IP disponíveis são de servidor *web http*, servidor FTP, servidor Telnet, cliente DHCP e servidor TFTP.

São disponibilizadas ao programador bibliotecas de funções que contêm as rotinas de acesso às *API* (*application protocol interface*) do *RTOS*, simplificando o uso dos recursos e acelerando a execução das rotinas. Além das bibliotecas, a empresa fornece aos programadores um pacote de desenvolvimento de programas em linguagem C (BECK, 2007).

5 Especificação do Executivo

Os requisitos mínimos que um executivo para CLP deve possuir para suportar a execução de um programa aplicativo e manter as funções de um CLP podem ser descritos da seguinte forma:

a) *driver* serial genérico;
b) capacidade de monitoração dos dados via porta de comunicação, tipicamente canal serial ou *ethernet*;
c) capacidade de alteração de configurações dos recursos via programa aplicativo;
d) alocação de memória para armazenar o diretório de operandos, e os próprios operandos;
e) diagnósticos do sistema;
f) varredura de entradas e saídas;
g) capacidade de carregamento do aplicativo;
h) temporização de eventos.

Para estruturação do sistema, inicialmente foi necessário identificar as operações que podem ser tratadas como *threads*, e, a partir disso, foi elaborado um modelo de implementação (figura 4).

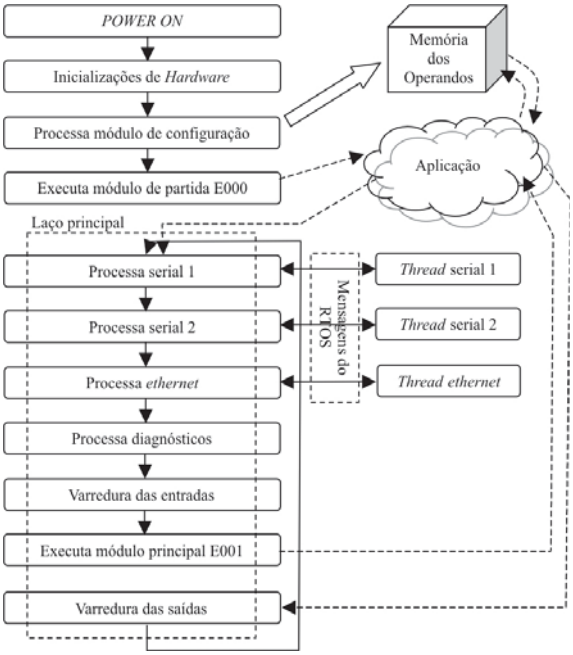

Figura 4 – Estrutura base do *software* do executivo do CLP

A execução do programa aplicativo, das rotinas de varredura de entradas e saídas, dos interpretadores de comandos e dos diagnósticos é realizada de forma seqüencial. Já os *drivers* seriais e *ethernet* devem ser executados em paralelo, pois necessitam de varredura periódica para evitar perda de informações recebidas ou travamento do sistema.

As *threads* seriais foram implementadas através de um *driver* serial genérico, pois essa estratégia permite alterar o protocolo de comunicação sem a necessidade de alteração do *driver* em si, apenas é necessária a criação de um novo interpretador de comandos para este novo protocolo.

Escolheu-se como protocolo de comunicação os protocolos ALNETI (canal serial) e ALNETII (canal *ethernet*), protocolo proprietário utilizado pelos CLPs da empresa Altus. Esse protocolo foi implementado utilizando os recursos do *driver* serial genérico para demonstrar as características do *driver* e a forma como utilizá-lo. Esta escolha possibilita a utilização da ferramenta MasterTool® na etapa de testes para monitorar e forçar os operandos do CLP.

O aplicativo de usuário pode ser considerado a tarefa mais importante executada pelo CLP. Os pré-requisitos principais do aplicativo são:

– programação em uma linguagem conhecida;
– interface para programação;
– possibilidade de carga *online* do aplicativo.

O grande desafio na execução do programa aplicativo é como interagir com os operandos do CLP. Estes operandos são facilmente acessados pelo executivo, pois foi quem alocou a memória; já o aplicativo desconhece a localização dos operandos.

Pode-se, inicialmente, enumerar quatro formas de estabelecer uma interatividade entre o executivo e o aplicativo:

– programa aplicativo montado pelo programador e desmontado pelo executivo;
– programa aplicativo compilado com compartilhamento de memória através de vetores de interrupções;
– programa aplicativo compilado com compartilhamento de memória através de troca de mensagens do *RTOS*;
– programa aplicativo compartilhado, utilizando processador sem recurso de memória protegida.

Na primeira forma, uma vez que o aplicativo é montado numa "linguagem" conhecida, o executivo desmonta seu código e descobre que funções devem ser executadas. Se o executivo for o decodificador dos comandos, conhece, portanto, a localidade de todos os operandos do CLP, então, facilmente, o aplicativo poderá ser executado e não haverá nenhum problema de interatividade com a memória. Uma das desvantagens é a velocidade de processamento do executivo. A decodificação do aplicativo é lenta, comparada com a execução de uma tarefa compilada, desperdiçando processamento que poderia ser utilizado para aumentar o desempenho do CLP. Outra desvantagem é a restrição na criação de novos comandos utilizados no aplicativo. Como o executivo é quem interpreta os comandos, este deve ser alterado quando forem criados comandos novos na aplicação. Deve-se evitar essa vinculação, pois ela cria uma incompatibilidade entre os aplicativos novos e os executivos antigos.

Um programa aplicativo compilado é tido como ótimo método de execução já que é executado na linguagem nativa do processador. Porém, para sua utilização, deve-se resolver o problema da localização dos operandos. Os operandos são alocados na memória pelo programa executivo, e esta alocação pode ser estática ou dinâmica. Como o aplicativo e o programa executivo são *threads* distintas, é necessária uma estratégia de troca de informação entre estas *threads.*

Na segunda forma, é proposto um meio de se resolver o problema citado com a utilização dos vetores de interrupção do processador. Os vetores de interrupção por *software* permitem o estabelecimento de pontos conhecidos para a chamada de rotinas básicas, recurso muito utilizado pelos sistemas operacionais. Pode-se alocar num vetor de interrupção uma rotina que, quando chamada pelo aplicativo, informa ao mesmo a localização dos operandos na memória. Da mesma forma, aloca-se uma interrupção para o executivo disparar o aplicativo. Não há problemas de dois programas (executivo e aplicativo) tentarem acessar a mesma área de memória (como os operandos), porque o aplicativo é executado como uma função seqüencial do executivo.

A terceira forma é a sincronização de dados através de *mailbox* (caixa de mensagem) entre o aplicativo e o executivo. A troca de mensagens é um serviço que deve ser oferecido pelo *RTOS* e que funciona como intermediário entre o aplicativo e o executivo. Nele estão disponíveis as funções para criar e localizar caixas de mensagens e, como o executivo e o aplicativo podem chamar estas funções, é possível estabelecer a troca de dados entre eles. O executivo pode, através de mensagens, requisitar ao aplicativo que execute suas funções. O aplicativo, por sua vez, também pode requisitar o valor dos operandos utilizados ou alterá-los. O que restringe o uso das *mailboxes* é o tamanho da mensagem disponível no *RTOS* utilizado, 12 *bytes* limitando a troca de dados a um operando por mensagem. Como uma aplicação pode ter até 49.196 operandos, a troca de mensagens seria intensa, ocupando muito o tempo de processamento e tornando muito lenta a execução do aplicativo. Além disso, este método tornaria a codificação do aplicativo muito complexa e extremamente dependente do *RTOS* utilizado.

A quarta forma utiliza o conceito de *mailboxes* associado ao compartilhamento de memória das *threads*. O executivo e o aplicativo são implementados como *threads* e, durante a fase de inicialização do executivo, este aloca a memória necessária para os operandos e cria *mailboxes* para troca de mensagens com o aplicativo. Feito isto, dispara a *thread* correspondente ao aplicativo que, através das *mailboxes*, toma conhecimento da localização dos operandos na memória e estabelece a sincronização com o executivo. Este método é o que possui o melhor custo-benefício entre codificação, portabilidade e velocidade, pois explora o compartilhamento de memória, como forma rápida de troca de uma grande quantidade de dados, e utiliza os recursos da troca de mensagens do *RTOS* para sincronização destes dados, evitando os problemas de sobrescrita de operandos (por acesso simultâneo do aplicativo e executivo).

Uma vez definido o método de troca de dados entre executivo e aplicativo, é necessário padronizar este acesso. Esta padronização foi feita através da criação de macro funções que o programador do aplicativo pode utilizar para acessar os operandos e as funções disponibilizadas pelo executivo. Foram criados três tipos de macros para auxiliarem no desenvolvimento de uma aplicação:
– macros de operandos: acessam os operandos criados pelo executivo. Foram implementados os seguintes operandos: M (memória), I (inteiro), F (ponto flutuante), X (inteiro longo), E (pontos de entrada), S (pontos de saída) e A (auxiliares);
– macros de bits: realizam o acesso a bit nos operandos;
– macros de temporização: utilizam os temporizadores criados pelo executivo.

Uma vez estruturado, codificado e compilado, é necessário submeter o executivo a testes para avaliar o desempenho do sistema. Assim, foi necessário desenvolver uma aplicação capaz de explorar os recursos oferecidos. Para a codificação da aplicação, optou-se novamente pela linguagem C, visto que a construção de um programa gerador de aplicativo está fora do escopo deste trabalho.

Para a bateria de testes funcionais, foram utilizadas as aplicações desenvolvidas na disciplina de Eletrotécnica do curso de Engenharia Elétrica da UNISINOS (CUSINATO, 2004). Como estas aplicações já foram

desenvolvidas e testadas em equipamentos comerciais, conhece-se previamente o comportamento da lógica de controle, necessitando apenas recodificá-la.

O excerto do programa, a seguir, exemplifica a codificação de uma aplicação com três temporizadores, I(1), I(2) e I(3). O primeiro é incrementado a cada 100ms, o segundo é incrementado a cada 500ms se o ponto 1 do operando A0 estiver ativo, e o terceiro é incrementado a cada 100ms, se o ponto 0 do operando A0 estiver ativo.

```
void huge E001(void)
{
   I(0)++;
   //conta I(1) a cada 100ms
   if ( isTEMPORIZA(I(10), 100) )
   {
      I(10) -= 100;
      I(1)++;
   }
   //conta I(2) a cada 500ms se A(0).1 = 1
   if ( isTEE( isBIT(A(0), 1), I(11), 500) )
   {
      I(11) -= 500;
      I(2)++;
   }
   //conta I(3) a cada 1000ms se A(0).0 = 0
   if ( isTED( isBIT(A(0), 0), I(12), 1000) )
   {
      I(12) -= 1000;
      I(3)++;
   }
}
```

Foram desenvolvidas três aplicações diferentes que incluíram não só a realização da função de controle, como também a monitoração dos operandos via sistema supervisório através do canal serial e da interface *ethernet*.

6 Considerações finais

Depois de realizados os testes práticos para verificar o funcionamento deste projeto, foi possível, a partir dos resultados obtidos, avaliar características que motivam sua aplicação, bem como suas restrições. Destaca-se, como maior vantagem, a funcionalidade e a interatividade para a operação do usuário, que não necessita saber como funciona o *software* executivo, apenas conhecer a linguagem de programação.

Portanto, com a utilização de sistemas *RTOS*, é possível obter uma ótima ferramenta na confecção do *software* executivo de CLPs, além do mais, como os *RTOS* estruturam-se de forma similar e possuem recursos semelhantes, a portabilidade deste tipo de aplicação é elevada, ou seja, não importa o *RTOS* utilizado, é possível migrar facilmente para outros *RTOS*, obtendo maior flexibilidade ao projetista.

Este trabalho pode ser encarado como uma base, um ponto de referência para a construção de um *software* executivo para CLPs mais completos.

Referências

BALL, Stuart R. **Embedded microprocessor systems – real world design**. 3ª ed. USA: Elsevier Science Editor, 2002.

BECK. <www.beck-ipc.com>, **IPC@CHIP® RTOS Documentation [Build 05.03.2007].** Acesso em 1º de maio de 2007.

CUSINATO, Liur J. **Controladores lógicos programáveis (CLP Altus)**, polígrafo de aula, 2004.

FreeRTOS™ Homepage, <www.freertos.org>. Acesso em 1º de maio de 2007.

GEORGINI, Marcelo. **Automação aplicada:** descrição e implementação de sistemas seqüenciais com PLCs. São Paulo: Érica Ltda., 2003.

PRASSAD, Shashi. **Multithreading programming techniques**. 1ª ed. USA: McGraw-Hill, 1997.