A Proposal for the Conception of Multitask Embedded Control Systems targeted to low-end DSPs

Marcos V. Linhares* Rômulo S. de Oliveira* Daniel J. Pagano*

*Universidade Federal de Santa Catarina – UFSC
Departamento de Automação e Sistemas – DAS
UFSC/CTC/DAS – Campus Universitário – Trindade
CEP: 88.115-400 – Florianópolis/SC
Email: {marcos, romulo, daniel}@das.ufsc.br

***Abstract*— Often, many control applications with time restrictions are implemented by the codification of complex programs in assembly language, programming timers, manipulating peripheral devices, tasks and interruption priorities in low level. The consequence of this approach is that the control software is produced by ad hoc techniques that can present low reliability and high cost. The design and implementation of an embedded real-time control system uses, basically, two types of professionals: the control engineer and the software engineer. The main tool used by the control engineer is the functional blocks diagram that, mapped to diagrams of components, supply the software engineer with the necessary information, but the simple use of functional blocks generates a monotasking system. This article describes a proposal that aims to improve the process of construction (design and implementation) of embedded systems with functional blocks diagram using the multitasking model. In this sense, the work consisted, mainly, of the multitasking framework proposal and the development of a real-time operational microkernel to support the developed framework. This work contributes to the improvement of software development processes for embedded systems in the context of control and automation.**

I. INTRODUÇÃO

Os sistemas de computação podem ser utilizados para controlar uma grande variedade de equipamentos, desde simples máquinas domésticas até completas instalações de produção. Quando estes sistemas estão firmemente fixados aos equipamentos ao ponto de que, se forem retirados, o equipamento deixa de funcionar então estes sistemas são chamados de sistemas embutidos (*embedded systems*)[1][2]. Normalmente, estes sistemas possuem restrições temporais na execução das tarefas para as quais foram projetados e programados e quando isto acontece tem-se um sistema de tempo real.

Os sistemas de tempo real são sistemas cujo funcionamento correto depende dos resultados produzidos por ele e do instante no qual esses resultados são produzidos[3][4][5].

Muitas das aplicações de controle com restrições de tempo são implementadas pela codificação de grandes programas em linguagem *assembly*, programando *timers*, manipulando em baixo nível dispositivos periféricos, tarefas e prioridades de interrupção. Apesar do código produzido por estas técnicas serem otimizados para uma execução eficiente, esta abordagem tem algumas desvantagens[6]:

- a implementação de grandes partes de código em linguagem *assembly* é muito mais difícil e consome muito mais tempo que a programação em linguagens de alto nível;
- exceto para os programadores que desenvolveram a aplicação, poucas pessoas conseguem entender o funcionamento completo do *software* produzido. E muitas vezes programadores bastantes experientes inserem mais complexidade ao código devido ao conhecimento adquirido;
- como a complexidade da programação aumenta, as modificações de grande parte do programa tornam-se muito difíceis, até mesmo para o programador original, o que leva muitas vezes a se jogar o código fora e se refazer o trabalho;
- sem o suporte de ferramentas e metodologias específicas, análises para verificação de restrições temporais são praticamente impossíveis.

O engenheiro, responsável por conceber a estratégia do controlador, usa nesse processo o diagrama de blocos funcionais. A simples utilização deste diagrama gera um sistema monotarefa onde são executadas uma seqüência de instruções, geralmente orientadas pela ocorrência da interrupção do *timer* (tempo este estipulado pelo engenheiro como ótimo para atuar sobre a planta), cujo o objetivo é controlar um determinado equipamento ou processo, recaindo-se invariavelmente em uma codificação de baixo nível. Por outro lado, as pressões do mercado exigem equipamentos não só robustos mas que ofereçam maiores funcionalidades a seus clientes e isto representa, para o engenheiro, mais tarefas a embutir em um determinado dispositivo. O modelo monotarefa começa a ser insuficiente, neste caso, para manter as restrições temporais do sistema final.

O objetivo deste trabalho foi estender um *framework* monotarefa já existente[7], que utiliza o mapeamento de diagramas de blocos para componentes de *software*, para um modelo multitarefa, mantendo as características principais do *framework* original.

Foi preocupação deste trabalho manter as características do projeto de sistemas utilizando diagramas de blocos e possibilitar a sua utilização em um ambiente multitarefa, executando em processadores DSP de pequeno porte.

Para tanto, dois esforços principais foram realizados:

- A criação de um *framework* multitarefa;
- O desenvolvimento de um núcleo operacional de tempo real (μKernel) para dar suporte de execução ao *framework* desenvolvido.

II. Framework proposto

O modelo monotarefa é amplamente utilizado, e indicado, quando se possui um *hardware* totalmente dedicado a uma única e determinada tarefa, como controlar o motor de um refrigerador. Usualmente, a execução da tarefa é dirigida pela ocorrência de uma interrupção, normalmente do *timer*, em intervalos periódicos.

Mas muitas aplicações reais são compostas por mais de uma tarefa, até por pressões do mercado que quer produtos mais atrativos com mais funcionalidades, como a existência de uma interface humano-máquina além do controlador.

Uma forma de resolver este problema é estender o modelo monotarefa para um modelo multitarefa, onde as diversas funcionalidades podem ser divididas em tarefas que concorrem entre si.

No modelo multitarefa proposto manteve-se o padrão de desenvolvimento de componentes (*Algorithm Standard*[8][9]), inseriu-se uma nova simbologia para tratar a multitarefa e criou-se estruturas adicionais para suportar a comunicação entre tarefas. Também foi criado um suporte de execução apropriado, na forma de um pequeno núcleo operacional, chamado de μKernel.

A. Componentes de Software

O projeto de sistemas baseado em diagramas de blocos é bastante intuitivo, conhecido pelo engenheiro de controle e utilizado em uma grande quantidade de ferramentas para controle de processos. O mapeamento destes blocos funcionais para componentes de *software* [7] torna esta metodologia ainda mais forte criando um canal confiável de comunicação entre o engenheiro de controle e o engenheiro de *software*.

O bloco funcional representa, um modelo matemático. Este modelo matemático realiza cálculos sobre valores de entrada gerando valores de saída, como pode ser visto no bloco genérico ilustrado na Figura 1.

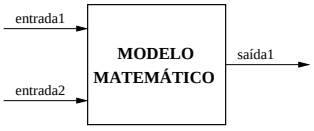

Fig. 1: Bloco Funcional genérico.

No mapeamento de um bloco funcional para um componente de *software* as entradas e saídas são atributos do componente e o modelo matemático corresponde a um algoritmo de computação. O componente genérico ilustrado pela Figura 2 mapeia o bloco genérico criado.

Fig. 2: Componente de *Software* genérico.

A partir deste mapeamento e seguindo as especificações formais do *Algorithm Standard* tem-se como resultado a geração de um algoritmo genérico que poderia ser aplicado a qualquer componente.

B. O Modelo Multitarefa

Para atender um sistema que necessita de várias tarefas executando deve-se utilizar um modelo multitarefa. A utilização deste modelos com um suporte adequado, onde as tarefas podem receber prioridades e tempos de atendimento individuais, com requisitos temporais (tarefas com maior prioridade podem interromper as de menor prioridade) torna este modelo muito atrativo. Com ele, aumenta-se o processamento de tarefas por unidade de tempo (*throughput*) além de diminuir os tempos de resposta.

Para aplicar o modelo multitarefa sobre a metodologia de desenvolvimento precisou-se criar um elemento para delimitar as fronteiras entre as tarefas. Cada tarefa é envolvida por este delimitador (linha tarcejada na Figura 3).

O papel de criação de um sistema embutido é um processo no qual é necessário, também, a interação entre o engenheiro de *hardware* e o engenheiro de *software*, que normalmente não são as mesmas pessoas. Para tanto, incluiu-se no modelo outro símbolo, e que faz parte da tarefa, para representar o *hardware* externo conectado ao componente de *software*: um retângulo tracejado com cantos aguçados. Isso possibilita ao engenheiro de *software* criar componentes específicos para um determinado dispositivo periférico (*device driver*). Por exemplo, na Figura 3 o bloco "MOTOR" é um exemplo desta classe, pois se trata de um *hardware* externo conectado a um periférico programado em *software*.

Supondo um sistema com três tarefas: controlar um motor e oferecer ao usuário uma interface de comunicação com recebimento e envio de dados, considerando componentes já desenvolvidos e a simbologia acima especificada, teríamos um sistema multitarefa ilustrado pela Figura 3.

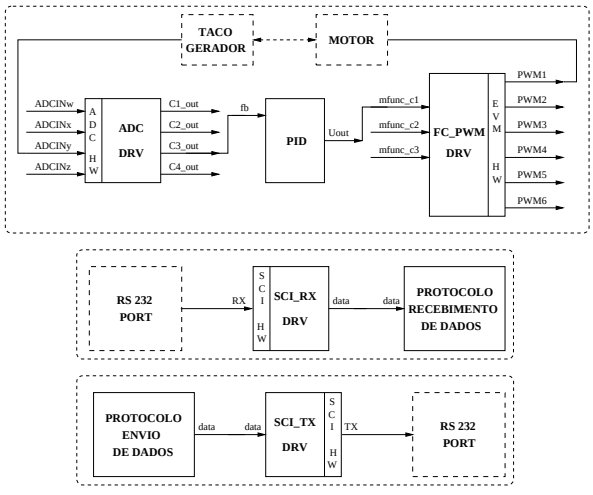

Fig. 3: Modelo Multitarefa.

A comunicação entre tarefas é uma importante característica que deve ser considerada em sistemas multitarefas, isso porque para realizar a comunicação são utilizados recursos que são

compartilhados entre todas as tarefas e num sistema multitarefa não deve-se permitir o acesso a um mesmo recurso por duas tarefas ao mesmo tempo, seja para leitura e/ou escrita (exclusão mútua).

C. Tipo abstrato FILA

A FILA (Figura 4) é um tipo abstrato criado para ser utilizado quando da necessidade de armazenamento de amostragens de um determinado valor. É utilizada, normalmente, na comunicação entre uma tarefa de controle e uma tarefa de emissão de dados, mas pode ser utilizada entre duas tarefas quaisquer quando existe a necessidade de se possuir um histórico de um determinado elemento.

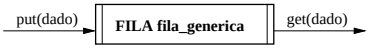

Fig. 4: Tipo abstrato FILA.

D. Tipo abstrato ATRIBUTO

O tipo abstrato ATRIBUTO (Figura 5) foi criado com o objetivo de conter atributos (parâmetros) de configuração ou de armazenamento em que somente o último valor interessa. É utilizado quando se tem a necessidade de armazenar valores únicos de um determinado parâmetro.

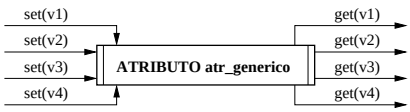

Fig. 5: Tipo abstrato ATRIBUTO.

III. Descrição do μKernel

Esta seção apresentará as características de projeto e codificação de um pequeno núcleo operacional multitarefa de tempo real, denominado μKernel, utilizado como suporte para o desenvolvimento de aplicações de controle embutidas em DSP de pequeno porte. É importante notar que o μKernel foi construído de forma a suportar o *framework* proposto na seção II.

Um *kernel* representa a parte mais interna de um sistema operacional e está ligado diretamente ao *hardware* da máquina, usualmente, um *kernel* realiza as seguintes atividades básicas: gerenciamento de processos; manipulação de interrupções; e sincronização entre processos.

A quantidade de serviços oferecidos por um *kernel* pode ser bastante vasta, desde o simples chaveamento de contexto até a criação dinâmica de tarefas incluindo um gerenciamento de memória através de coletores de lixo (*garbage colectors*). O que determina quais serviços serão oferecidos pelo *kernel* são as restrições da plataforma alvo (como: memória, capacidade de processamento, etc.) e as restrições da aplicação (como: multitarefa, comunicação entre tarefas, etc.). Avaliadas estas características pode-se levantar os requisitos necessários ao desenvolvimento do *kernel* [6].

Toda a teoria aqui discutida é aplicável em qualquer tipo de processador ou controlador desde que se mantenha fiel ao *framework* especificado e ao modelo de componentes.

O DSP utilizado no desenvolvimento do μKernel foi o TMS320C2407A da Texas Instruments embutido em uma placa de desenvolvimento chamada eZdsp acompanhada de um ambiente integrado de desenvolvimento (*Code Composer*) com compilação, depuração e emulação.

O TMS320C2407A é um controlador DSP da família C24x. Sua arquitetura é baseada em uma arquitetura *Harvard* modificada composta por dois barramentos separados, um para programa e outro para dados. Este DSP possui 2,5k *words* de memória RAM e 32k *words* de EEPROM para espaço de programa.

A. Estrutura do μKernel

De forma a suportar o *framework* proposto e respeitar as restrições da plataforma alvo, implementou-se no μKernel o mínimo de funções necessárias, organizadas conforme a estrutura hierárquica ilustrada na Figura 6 e descritas abaixo.

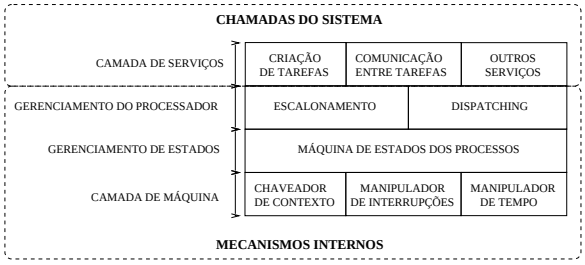

Fig. 6: Estrutura hierárquica do μKernel.

- Camada de máquina: esta camada interage diretamente com o *hardware* da máquina;
- Camada de gerenciamento de estados das tarefas: para manter os estados (pronto, executando, etc.) de diversas tarefas, o *microkernel*, precisa gerenciar algumas listas onde as tarefas estarão agrupadas por estado;
- Camada de gerenciamento do processador: diz respeito às operações de escalonamento e despacho de tarefas;
- Camada de serviços: Este nível fornece todos os serviços visíveis ao usuário, com um conjunto de chamadas de sistema.

O código fonte do μKernel tem aproximadamente 1.000 linhas de código, parte escrita em linguagem *C* e parte em linguagem *Assembly*. Alguns dados mais técnicos de utilização da memória do DSP podem ser vistos na Tabela I.

Itens avaliados	Valores
Memória de programa: fonte	$\simeq 0,9 kwords$
Dados estáticos: globais	$\simeq 20 words$
Dados dinâmicos: por tarefa	$\simeq 45 words$
Dados dinâmicos: por FILA	$\simeq 5 words$

TABLE I
Dados sobre o μKernel.

IV. APLICAÇÃO AO CONTROLE DE UM MOTOR

A aplicação utilizou um único *DSP* da família C2000 da Texas Instruments (TMS320C2407A) que deverá suportar múltiplas tarefas tais como: controle de motor, interface de entrada de dados (teclado), interface de saída de dados (*display*), recebimento e envio de dados remotos (via RS-232, serial padrão).

O controle do motor deverá ser a tarefa mais importante da aplicação devendo manter a velocidade do motor conforme a referência e utilizando os parâmetros configurados para o PID e para o PWM que serão feitos através da inserção de dados via interface de entrada (teclado) e visualizados via interface de saída (*display*). Esta configuração também poderá ser feita remotamente através do recebimento de dados, enviados via computador de mesa com interface serial. Este último também poderá receber dados amostrados atualizados, enviados remotamente pelo dispositivo.

A Figura 7 ilustra um protótipo da aplicação com suas respectivas tarefas.

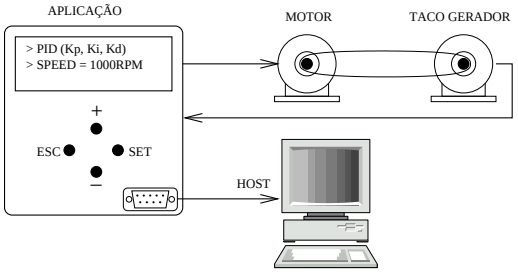

Fig. 7: Protótipo da aplicação.

Os itens descrevem, detalhadamente, as operações que podem ser realizadas pelo usuário com o objetivo de interagir com a aplicação.

1) O usuário poderá configurar os parâmetros do PID e do PWM via teclado ou via envio de dados serial;
2) O usuário poderá requisitar amostras das variáveis do sistema via interface serial.

Enquanto a falha em cumprir um requisito funcional individual pode degradar o sistema, a falha em cumprir um requisito não-funcional de sistema pode tornar o sistema inútil. Tendo em vista as restrições impostas pelo desenvolvimento de sistemas embutidos em DSP (tamanho da memória de programa, tamanho da memória de dados, etc.) e as restrições impostas pela aplicação (velocidade de atuação do controlador na malha), chegou-se aos seguintes requisitios não-funcionais:

1) A tarefa de controle deve atuar sobre o motor a cada 50ms (milisegundos);
2) As outras tarefas executam a cada 100ms (milisegundos) e seguem a seguinte ordem de prioridade: teclado, *display*, envio e recebimento de dados remotos;

A. Projeto da aplicação

Primeiramente são identificadas as tarefas e separadas conforme suas prioridades, criam-se então os blocos que representam os *hardwares* externos e por conseqüência os componentes que realizam a interface entre a tarefa e o *hardware* externo (*driver*). Após isto criam-se os componentes internos a cada tarefa. É necessário também inserir os mecanismos de comunicação entre as tarefas quando estes forem necessários.

Pode-se identificar, a partir da especificação, 5 (cinco) tarefas, onde as prioridades na execução são fixas e decrescentes:

- Uma tarefa para controlar um motor, de alta prioridade e com período de 50ms;
- Duas tarefas para interação humano-máquina, uma interface de entrada de dados (teclado) e uma interface de saída de dados (*display*) ambas com período de 100ms;
- Duas tarefas para comunicação remota, uma para envio de dados e uma para recebimento de dados via comunicação serial, também com períodos de 100ms.

B. Tarefa de controle do motor

A tarefa de controle do motor, principal tarefa do sistema, tem como objetivo levar e manter a velocidade do motor (*fb*) o mais próximo possível da referência (*ref*). O controle é realizado por um controlador PID (Proporcional Integral Derivativo) e exercido sobre o motor através de um PWM (*Pulse Width Modulation* - modulação por largura de pulso). O sistema é realimentado por um ADC (*Analogic to Digital Converter*) que está conectado a um taco gerador.

Os seguintes blocos foram definidos para atender a esta tarefa:

- *Hardware* externo: Motor e Taco Gerador;
- Manipuladores de dispositivos periféricos (*device drivers*): ADC_DRV e PWM_DRV;
- Algoritmo: Controlador PID.

Os parâmetros do PID e do PWM recebem, inicialmente, valores pré-configurados mas podem ser configurados através da entrada de dados. A tarefa de controle (Figura 8) funciona com a atuação do PWM sobre o motor que está conectado a uma taco gerador que realimenta a tarefa através do conversor analógico digital (ADC). O controlador verifica a realimentação e conforme os parâmetros configurados recalcula a largura de pulso do PWM.

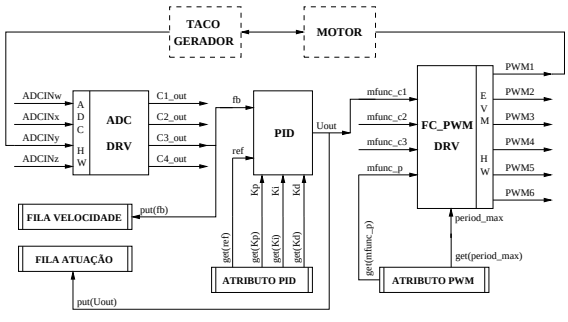

Fig. 8: Tarefa de controle do motor.

Como foi especificado anteriormente a tarefa de controle do motor faz parte de um conjunto multitarefa que está ilustrado na Figura 9.

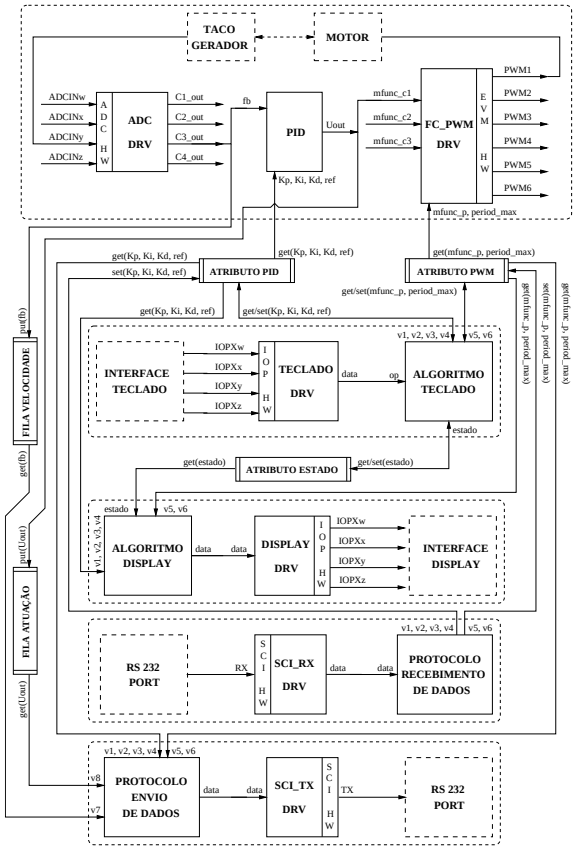

Fig. 9: Tarefas da aplicação

V. COMPORTAMENTO TEMPORAL

De forma complementar ao desenvolvimento do *framework* e do μKernel, foram levantados alguns dados empíricos que exprimem o comportamento temporal da aplicação juntamente com o núcleo de suporte. Esta seção ilustra como uma análise temporal poderia ser realizada pelo engenheiro de *software* dentro do *framework* proposto. Como comportamento temporal da aplicação podemos considerar:

- Tempo de computação das tarefas (C_x) - tempo que a tarefa gasta durante seu ciclo de execução;
- Tempo de *overhead* do μKernel (O_x) - o *overhead* é o preço a ser pago pelas facilidades do μKernel, é representado pela soma dos tempos gastos com a execução do núcleo de suporte desde o momento em que a tarefa x é inserida na lista de pronto até o término de seu ciclo de execução, e compreende:
  - o chaveamento de contexto entre as tarefas, salvamento dos registradores da tarefa que está deixando o processador e carregamento dos registradores com informações da tarefa que está ganhando o processador;
  - a manipulação das listas de estados, transição das tarefas entre os estados: espera, pronto e executando;
  - o tratador de interrupção do tempo, no caso da aplicação configurado para ocorrer a cada 5ms.
- Tempo de resposta das tarefas (R_x) - representado pela soma do tempo de computação da tarefa x, do tempo de interferência das tarefas de maior prioridade e dos tempos de *overhead* do μKernel até o término da execução da tarefa x;
- Tempo de interferência das tarefas de maior prioridade (I_x) - é a soma dos tempos de computação das tarefas mais prioritárias que a tarefa x durante seu tempo de resposta (R_x).

Para o cálculo do tempo de resposta pode-se utilizar:

$$R_x = C_x + I_x + O_x$$

onde I_x é dado por:

$$I_x = \sum_{i=1}^{x-1} \left\lceil \frac{R_x}{P_i} \right\rceil \cdot C_i$$

O tempo de *overhead* do μKernel é equivalente à latência máxima da interrupção, isso é devido ao μKernel executar sempre com as interrupções desabilitadas. Esta escolha simplificou a programação do mesmo, pois não existe concorrência interna no μKernel. Entretanto, perde-se a oportunidade de diminuir este *overhead*. A latência máxima medida para este conjunto de tarefas ficou em, aproximadamente, 400 μs.

O período de interrupção do tempo foi definido em 5 ms ($1/10$ do menor período). O período e prioridade das tarefas podem ser vistos na Tabela II.

Na aplicação proposta foram feitas medições de tempo (de forma empírica, através da utilização de um osciloscópio)para cada tarefa e que podem ser vistas na Tabela III.

O osciloscópio foi conectado fiscamente á porta digital E do DSP (8 saídas digitais) e cada tarefa colocava o nível de uma saída digital em alto quando iniciava seu ciclo e em baixo quando terminava seu ciclo. Para medir o tempo de computação da tarefa estipulou-se um período grande o suficiente para que nenhuma tarefa interferisse na outra. A latência máxima foi amostrada da mesma forma com a diferença que a saída digital era manipulada na entrada do μKernel para alto e na saída dele para baixo. O restante dos tempo foram calculados segundo as equações apresentadas.

x	Tarefa	Prioridade	Período (P_x)
1	**motor**	1	50 ms
2	**teclado**	2	100 ms
3	***display***	3	100 ms
4	**envio remoto**	4	100 ms
5	**recebimento remoto**	5	100 ms

TABLE II
PERÍODO E PRIORIDADE DAS TAREFAS.

Pode-se perceber pela análise das tabelas que a tarefa de controle do motor, tarefa de mais alta prioridade na aplicação, sofre somente a interferência do μKernel resultando em um tempo de resposta (R_x) de 9,5 ms (resultado da soma de: $C_x = 9,1$, com $O_x = 0,4$). O restante das tarefas, a medida que decrescem as prioridades, vão sofrendo maior interferência por parte das outras tarefas (I_x) e também do μKernel (O_x) resultando em tempos de resposta maiores.

Tarefa (x)	$R_x(ms)$	$C_x(ms)$	$I_x(ms)$	$O_x(ms)$
1	$9,5$	$9,1$	0	$0,4$
2	$17,5$	$7,5$	$9,1$	$0,9$
3	$25,5$	$7,5$	$16,6$	$1,4$
4	$33,5$	$7,5$	$24,1$	$1,9$
5	$42,5$	$7,5$	$31,6$	$3,4$

TABLE III
COMPORTAMENTO TEMPORAL DAS TAREFAS DA APLICAÇÃO.

VI. CONSIDERAÇÕES FINAIS

Este trabalho descreveu uma proposta para melhorar o processo de desenvolvimento (projeto e implementação) de sistemas embutidos multitarefa. Para tanto, o trabalho consistiu da proposta de um *framework* multitarefa e o desenvolvimento de um núcleo operacional de tempo real para dar suporte ao *framework* desenvolvido.

O modelo monotarefa minimiza muito as funcionalidades de um sistema já que este ficará restrito à execução de uma única tarefa. No caso de sistemas de controle, isso fica restrito a um controlador. Se o sistema for composto por tarefas que operam em freqüências diferentes o modelo monotarefa não é capaz de atender eficientemente. É sabido que os diagramas de blocos são uma linguagem de modelagem de sistemas muito utilizada entre os engenheiros e mudar este paradigma requer uma reestruturação do conhecimento que este engenheiro agregou durante grande parte de sua formação. Considerando-se isto optou-se por estender o modelo monotarefa, para um modelo multitarefa, sem no entanto perder as potencialidades do projeto utilizando diagramas de blocos.

Para suportar o *framework* multitarefa proposto foi desenvolvido um núcleo operacional, chamado de μKernel. Espera-se com este trabalho contribuir para a melhoria dos processos de desenvolvimento de *software* para sistemas embutidos no contexto do controle e automação.

REFERENCES

[1] A. S. Berger, *Embedded Systems Design: An Introduction to Process, Tools and Techniques*. CMP Books, 2002.

[2] D. E. Simon, *An Embedded Software Primer*. Addison-Wesley, 1999.

[3] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers, 1997.

[4] Q. Li and C. Yao, *Real-time concepts for embedded systems*. CMP Books, 2003.

[5] J. Stankovic and K. Ramamrithan, Eds., *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[6] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, 4th ed., ser. The Kluwer international series in engineering and computer science. Real-time systems. Kluwer Academic Publishers, 2002.

[7] D. Figoli, *A Software Modularity Strategy for Digital Control Systems*, ser. Technical Document - Application Report. Texas Instruments, DCSA, March 2001.

[8] S. Blonstein, *The TMS320 DSP Algorithm Standard*, ser. Technical Document - White Paper. Texas Instruments, May 2002, revision C.

[9] T. Instruments, Ed., *TMS320 DSP Algorithm Standard - Rules and Guidelines*, ser. Technical Document - User Guide. Texas Instruments, October 2002, revision E.