MULTITHREADING EM SUPERVISÃO INDUSTRIAL: DESCRIÇÃO DE UM KERNEL DE TEMPO-REAL

Sérgio J. Dias
Unidade de Automação Industrial e Electrónica
IDITE-Minho
Braga

Ricardo J. Machado
Dep. Sistemas de Informação
Escola de Engenharia
Universidade do Minho
Guimarães

João M. Fernandes
Dep. Informática
Escola de Engenharia
Universidade do Minho
Braga

Sumário

O objectivo principal deste artigo consiste em apresentar uma abordagem à supervisão de processos e equipamentos industriais baseada em componentes embebidos multitarefa. O artigo discute, desta forma, a estratégia de escalonamento das várias tarefas a supervisionar e analisa algumas partes da implementação do *kernel* que a suporta.

1. INTRODUÇÃO

O presente trabalho encontra-se inserido no âmbito do projecto INFRACOM da iniciativa intitulada SIPROFIT (Iniciativa para a Dinamização do Desenvolvimento e da Produção de Sistemas Produtivos para a Fileira Têxtil). O objectivo global desta iniciativa consiste na criação de competências tecnológicas na fileira têxtil (Tinturaria, Acabamentos, Confecção, Tecelagem, etc.), concretizando-se na concepção, desenvolvimento e produção de sistemas e equipamentos produtivos.
O objectivo do projecto VIRTUAL AUTOMATION é o desenvolvimento de uma metodologia e respectivas ferramentas de apoio ao projecto de sistemas de informação orientadas ao controlo, para a supervisão de processos nas industrias têxtil e do vestuário , no suporte à gestão da produção, qualidade e manutenção. Pretendendo-se fornecer sistemas modulares, escaláveis e parametrizáveis. Para a validação do projecto foi desenvolvido um protótipo numa empresa têxtil.
Este artigo foca o desenvolvimento de um *kernel* (para microcontroladores da família MCS-51) capaz de suportar o escalonamento multitarefa em aplicações embebidas de supervisão e monitorização industriais, em níveis CIM (***c****omputer-****i****ntegrated* ***m****anufacturing*) onde as questões de tempo-real se colocam com alguma premência.

2. RTOS

Actualmente, quase todos os RTOS (***r****eal-****t****ime* ***o****peration* ***s****ystems*) comercialmente disponíveis suportam um ORB (***o****bject* ***r****equest* ***b****roker*) para que as soluções adoptadas para implementar os CBS (***c****omputer-****b****ased* ***s****ystems*) tempo-real embebidos possam ser o mais tecnologicamente abertas possível [1]. A abertura tecnológica no âmbito dos RTOS tem evoluído enormemente nas últimas duas décadas, nomeadamente no que diz respeito às camadas funcionais a que dão suporte directo, dando o seu, cada vez maior, contributo para dotarem os CBS tempo-real embebidos com capacidades avançadas de comunicação inter-sistemas e de distribuição funcional das aplicações finais, tal como a fig. 1 pretende ilustrar [2]. Os valores numéricos apresentados na figura representam a percentagem do software total suportado pelo RTOS.
Desde a, já algo longínqua, iniciativa do IEEE em definir uma norma para implementação de RTOS para ambientes embebidos, o *Real-Time POSIX* [3, 4], até à recente disponibilização de uma versão aberta e completamente grátis do *LINUX* para sistemas tempo-real, o *Real-Time LINUX* [5, 6], o próprio conceito de RTOS tem evoluído e tem-se modificando à medida que as necessidades de abertura e distribuição se têm tornado cada vez mais prementes. Neste contexto de mudança paradigmática da forma como os CBS tempo-real embebidos têm sido recentemente olhados, em contraste directo com a sua anterior imagem de abandono e desprezo a que a comunidade informática relegou aquele tipo de sistemas durante muitos anos [7], é vital distinguir o importante do acessório no que concerne a características efectivamente relevantes aquando da selecção ou desenvolvimento de um RTOS [8]. Desde logo, surge a necessidade de decidir em que nível CIM o CBS tempo-real embebido em causa se encontra para definir as funcionalidades que o RTOS a utilizar deve disponibilizar e de que forma as deve implementar.

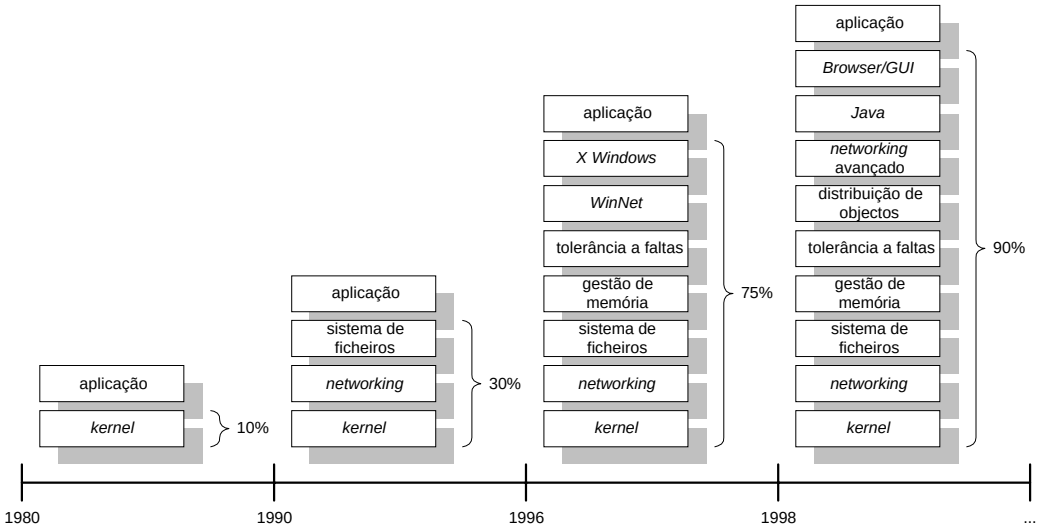

Figura 1 - Evolução funcional dos RTOS.

Sendo a preempção e as prioridades dinâmicas duas características desejáveis, do ponto de vista funcional, para resolver a questão do escalonamento em CBS tempo-real [9], a sua adopção no âmbito de um RTOS pode levantar o problema da dificuldade da sua implementação computacional. Um escalonamento com aquelas características deve lidar com a sincronização inter-processos para tratar os acessos múltiplos a recursos partilhados, recorrendo, por exemplo, a *semáforos*, *mensagens* ou *caixas do correio*. Adicionalmente, é necessário garantir que a comutação entre processos não ocorra no meio de uma região crítica, pelo que, nestas circunstâncias, utilizam-se, tipicamente, *monitores*. Perante esta escalada de mecanismos a implementar num RTOS, é lícito colocar em causa se esta é a forma mais adequada de conceber RTOS, tendo em conta os *overheads* de memória e de processamento a que aqueles, e outros, mecanismos obrigam [10].

Num extremo oposto, é possível adoptar um escalonamento cooperativo entre os processos, que, pelo facto de ser especificado no próprio algoritmo da aplicação, pode dispensar total, ou parcialmente, as funcionalidades oferecidas pelos RTOS. Esta abordagem é conhecida como *síntese de software* e está geralmente ligada à construção de compiladores para a geração automática de código para CBS embebidos [11].

Se para suportar o nível CIM 2 (*area control*) pode não ser muito penalizante recorrer a um RTOS com escalonamento preemptivo (o *WINDOWS NT* possui um escalonador tempo-real, pelo que é um RTOS), uma vez que o sistema computacional que suporta esse nível possui frequentemente uma elevada capacidade de processamento e os requisitos temporais são raramente do tipo estrito (*hard real-time*), já no nível CIM 1 um RTOS com escalonamento preemptivo poderá, eventualmente em certas circunstâncias, tornar-se numa má opção tecnológica. No nível CIM 1 (*unit control*) é típico prescindir da maior parte das funcionalidades (serviços) apresentadas na fig. 1, adoptando-se, desta forma, um RTOS (desde que necessário) com um conjunto limitado de funcionalidades, designado de *núcleo tempo-real* (*real-time kernel*). Os serviços disponibilizados pelos núcleos são geralmente os seguintes: (1) gestão dos processos e escalonamento do processador aos diferentes processos; (2) sincronização e mecanismos de exclusão mútua; (3) comunicação entre processos; (4) gestão e suporte de interrupções.

Mesmo com uma selecção criteriosa dos serviços de um RTOS estritamente necessários para suportar a aplicação a implementar num CBS tempo-real embebido, o escalonamento consiste no ponto principal ao qual o projectista deve dar a maior atenção. Ao contrário do que sucede nos sistemas orientados aos dados (*data oriented*) em que a regularidade das taxas de chegada dos dados e das acções a efectuar sobre eles permite adoptar técnicas de escalonamento já conhecidas e dominadas, o escalonamento em CBS tempo-real embebidos orientados ao controlo (*control oriented*) constitui ainda um tópico de investigação com uma actividade intensa, uma vez que, para sistemas muito complexos, ainda não existem formas suficientemente satisfatórias para [12]: (1) determinar o pior tempo de execução (WCET - ***w**orst-**c**ase **e**xecution **t**ime*) de um processo, sem ser exageradamente pessimista; (2) caracterizar as dependências entre os processos; (3) verificar a sua escalonabilidade com requisitos realistas.

3. A ARQUITECTURA DOS FMOTS

A abordagem seguida no desenvolvimento das aplicações embebidas foi orientada aos componentes parametrizáveis que, no âmbito do projecto VIRTUAL AUTOMATION, se designam de FMOTS (***f**unctional **m**odules **o**ff-**t**he-**s**helf*); ver fig. 2.

O desenvolvimento de FMOTS, à custa da interligação de objectos de alto (recursos aplicacionais) e de baixo-nível (recursos computacionais da arquitectura alvo que suporta o FMOTS), é, à primeira vista, uma tarefa fácil, no entanto, na prática, surgem alguns problemas que podem colocar em causa a viabilidade desta abordagem, dada a complexidade das relações, dependências e constrições que se deparam ao projectista (engenheiro de *software*) aquando da

utilização de linguagens que não suportam directamente o CBD. De facto, uma linguagem para ser designada de *component-oriented* deve suportar, para além de outras características, o encapsulamento de informação sobre vários objectos [13], em contraste com as linguagens *object-oriented* que devem suportar, para além de outras características, o encapsulamento de informação sobre objectos individuais [14].

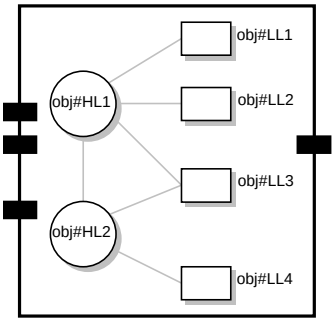

Figura 2 - Um FMOTS.

A utilização de linguagens OO (orientadas ao objecto) para suportar a abordagem CBD exige uma reflexão prévia com vista à definição de algumas regras para uma correcta adequação dos mecanismos de modelação OO à concepção de agregados de objectos que satisfaçam minimamente o cumprimento das características típicas de componentes [15]. A reutilização, segundo uma abordagem CBD, à custa de linguagens OO exige uma adequada utilização dos mecanismos de agregação (composição), herança e associação [16].

A composição e a herança constituem os mecanismos básicos que suportam a reutilização na concepção de FMOTS, mas só por si não garantem que um engenheiro de *software* os desenvolva correcta e adequadamente. A aplicação em prática da própria reutilização num contexto de CBD tem-se mostrado aquém das expectativas criadas no início dos anos 90, com o amadurecimento e a disseminação generalizada da tecnologia OO.

A concepção de FMOTS, segundo uma abordagem CBD, obriga a considerar o tratamento de várias questões [17]:

(1) *Coordenação*. Assumindo que a arquitectura alvo é uniprocessador e que não é utilizado nenhum RTOS, a existência de mais do que um objecto de alto-nível num único FMOTS coloca o problema da coordenação. Ou seja, os objectos são todos activos? Se sim, como serializar o comportamento do FMOTS? Se não, qual deles é que coordena?

(2) *Partilha de recursos*. A existência de vários objectos de alto-nível que necessitam de deter um mesmo objecto de baixo-nível é uma realidade incontornável. Perante esta constatação, surge a questão de como implementar esta partilha de objectos de baixo-nível. Por herança? Por composição? Por passagem de parâmetros?

(3) *Reutilização*. A utilização da abordagem CBD na concepção de FMOTS coloca a questão de como beneficiar da reutilização aquando da decisão da arquitectura interna dos FMOTS. Como implementar a reutilização? À custa da herança e/ou composição de quê? Outros objectos (classes)? Outros FMOTS? Como suportar a reutilização?

No âmbito do projecto VIRTUAL AUTOMATION, na resposta àquelas questões considerou-se que [15]:

(1) *Objecto principal*. Cada FMOTS deve possuir, para além dos objectos de alto-nível funcionalmente necessários para cumprir os requisitos algorítmicos que justificam a sua existência, um objecto de alto-nível principal (*obj#HLP*) que assume o papel de objecto coordenador do FMOTS, ou seja, controla o fluxo de execução principal intra-FMOTS. Em situações particulares, um, e só um, dos objectos de alto-nível do FMOTS pode acumular as funções de objecto de alto-nível principal, deixando de existir um objecto de alto-nível principal em exclusividade de funções.

(2) *Objectos de alto-nível*. Por questões de reutilização, os objectos de alto-nível de um FMOTS podem constituir refinamentos ou implementações de classes já existentes no repositório, a partir da qual é possível herdar um comportamento próximo do desejado. Desta forma, cada objecto de alto-nível estabelece uma relação de herança com a sua superclasse (de facto, será a classe do objecto de alto-nível que estabelece a relação de herança com a sua superclasse). Tipicamente, a relação de herança será do tipo especificação, uma vez que a superclasse (*class#HL1* e *class#HL2*, na fig. 3) já existente numa determinada região de decomposição do repositório (*Reg. B* e *Reg. C*, na fig. 3) deve consistir numa especificação abstracta incompleta relativamente ao relacionamento com os objectos de baixo-nível (*obj#LL1*, *obj#LL2*, *obj#LL3* e *obj#LL4*, na fig. 3), específicos para cada arquitectura alvo.

(3) *Objectos de baixo-nível*. Os objectos de baixo-nível necessários para cada FMOTS são provenientes da região de decomposição correspondente à arquitectura alvo que suporta computacionalmente o FMOTS. Estes objectos de baixo-nível devem ser criados pela classe de alto-nível principal que, aquando da criação das classes de alto-nível não principais, deve passar a estas as referências das instâncias das classes de baixo-nível por ele criadas. Este mecanismo algo "rebuscado" deve-se ao facto de existir a necessidade de garantir que os vários objectos de alto-nível acedam, caso precisem, a um mesmo objecto de baixo-nível.

A fig. 3 retracta o cenário em que a coordenação do FMOTS se encontra centralizada num único objecto principal. Para evitar o eventual problema da explosão

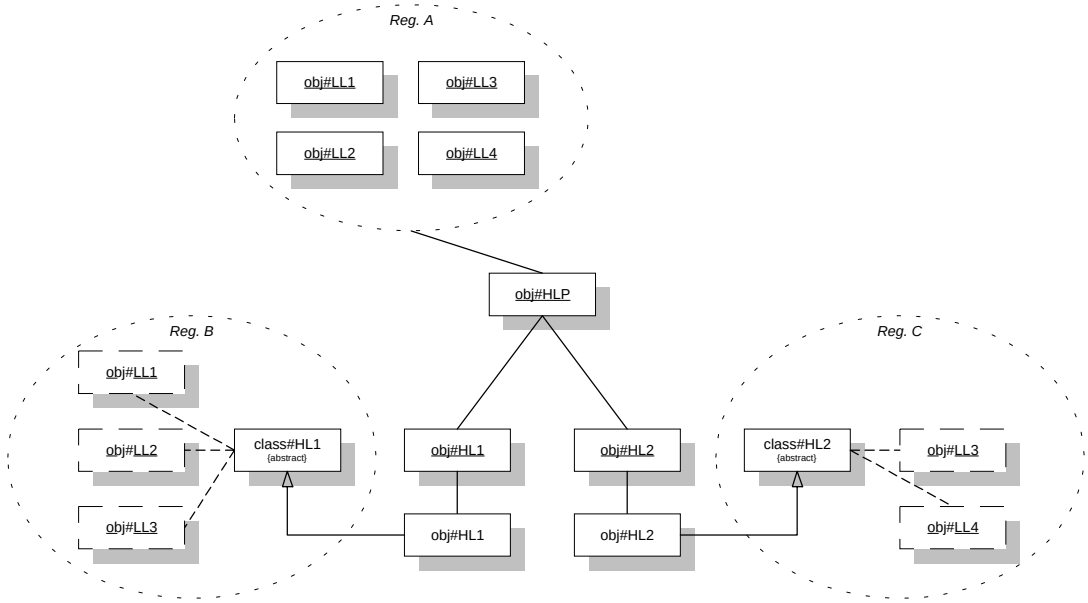

Figura 3 - Diagrama de classes e de objectos de um FMOTS.

do espaço de estados, decorrente do controlo (supervisão) concorrente de diversas actividades independentes à custa de uma coordenação centralizada num único objecto principal, é conveniente suportar múltiplos objectos principais, sendo cada um responsável por coordenar o seu conjunto de objectos de alto- e de baixo-nível (sub-*thread* principal). Neste cenário, de fluxos algorítmicos intra-FMOTS explicitamente concorrentes inter-objectos principais, mantêm-se válidas, para cada um deles, as regras definidas no âmbito da explicação do esquema da fig. 3, com a excepção de que não é obrigatório que cada sub-*thread* principal dê origem a uma sub-região de decomposição da região de decomposição que especifica o FMOTS, pelo que, nestas circunstâncias, a classe que encabeça esta região de decomposição não deve coincidir com nenhum dos objectos principais existentes

4. O KERNEL VASch

Cada FMOTS deve ser capaz de supervisionar mais do que um equipamento e/ou processo industrial em simultâneo e com características distintas, ou seja, com valores distintos para os parâmetros, se bem que pertencentes à mesma classe funcional. Esta exigência é relativamente simples de cumprir quando o sistema supervisionador (mais propriamente o algoritmo de supervisão) é propositadamente desenvolvido para tratar as peculiaridades do sistema a supervisionar, quer no que diz respeito ao número dos seus sub-sistemas, quer relativamente ao tipo e diversidade deles, uma vez que estas características não funcionais são facilmente incorporadas no algoritmo de supervisão, numa abordagem típica de escalonamento cooperativo das diversas actividades de supervisão. No entanto, apresenta-se extremamente complicado desenvolver FMOTS que adoptem a abordagem cooperativa com um grau de generalidade elevado, pelo que, nestas circunstâncias, a utilização de um núcleo de RTOS, externo à aplicação do FMOTS, para realizar o escalonamento das tarefas de supervisão dos diversos equipamentos e/ou processos industriais existentes, apresenta-se como a solução mais adequada, do ponto de vista do controlo da complexidade do desenvolvimento do FMOTS.

Tal como já referido, é necessário ser criterioso na selecção/desenvolvimento de RTOS para CBS tempo-real embebidos, devido, essencialmente, ao impacto no desempenho e no consumo dos recursos que acarreta a incorporação de um RTOS. Tendo em conta esta preocupação, foi desenvolvido um escalonador para os FMOTS, designado de *VASch* (***v****irtual* ***a****utomation* ***sch****eduler*).

Com a fig. 4 pretende representar-se os objectos existentes no interior de um FMOTS. Este FMOTS possui um objecto de alto-nível principal (*obj#HLP*), dois objectos de alto-nível (*obj#HL1* e *obj#HL2*), o objecto *timer2*, o objecto *kernel* e o objecto *can2*. O fluxo de execução normal do FMOTS encontra-se fundamentalmente divido em três *threads* (não é suposto existirem interacções entre objectos de alto-nível fora destes três fluxos):

(1) *Thread principal*. Este "fio de execução" corresponde ao fluxo de processamento desencadeado pelo *obj#HLP* que executa operações próprias e invoca operações a outros objectos de alto- e de baixo-nível, de forma a cumprir o algoritmo característico do FMOTS. Esta *thread* aparece representada na fig. 5 com vectores a preto e, tal como pode ser constatado pela sequência ilustrada, é a de mais baixa prioridade, ou seja, pode ser interrompida por qualquer das outras duas. No caso de existirem vários objectos principais, esta *thread* encontra-se multiplexada no tempo, à custa da intervenção de

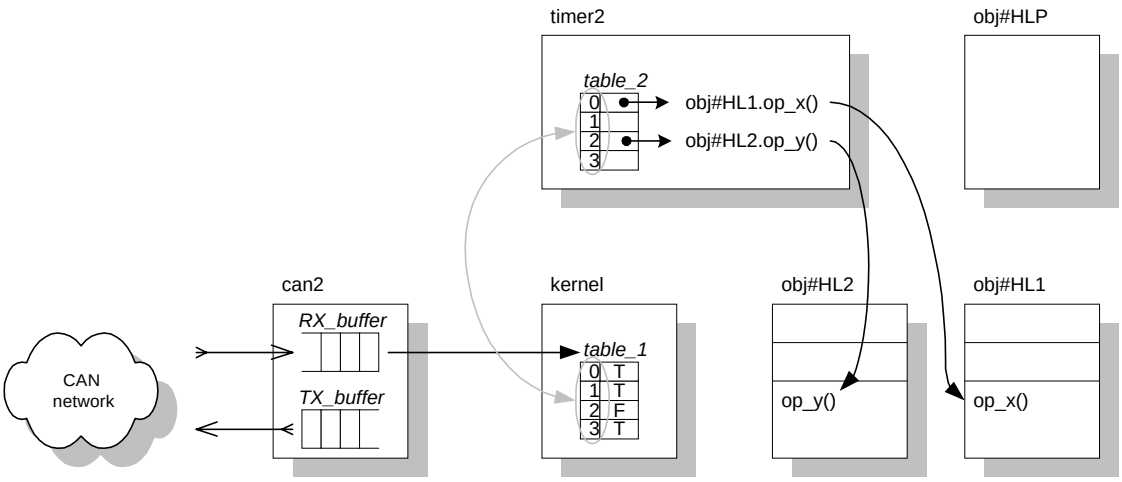

Figura 4 - Mecanismo de recepção de mensagens.

um escalonador, entre as várias sub-*threads* principais.

(2) *Thread do can*. Este "fio de execução" corresponde ao fluxo de processamento desencadeado pelos objectos *can1* e *can2* que, sempre que chegue um pacote de informação via rede *CAN* para o FMOTS, interrompe a *thread* em execução, armazena o pacote no FIFO de pacotes recebidos (*RX buffer*) e assinala a chegada do pacote, preenchendo a *table_1*, localizada na zona de memória partilhada do objecto *kernel*. Esta *thread* aparece representada na fig. 5 com vectores a cinza escuro e é a de mais alta prioridade, ou seja, pode interromper qualquer uma das outras duas.

(3) *Thread do timer2*. Este "fio de execução" corresponde ao fluxo de processamento desencadeado pelo objecto *timer2* que é executado periodicamente (*71,1 ms*) para verificar, consultando a *table_1*, se existem pacotes no *RX buffer* para serem processados. Caso existam, a *table_2* fornece uma referência para a operação que deve realizar tal processamento. Esta *thread* aparece representada na fig. 5 com vectores a cinza claro e possui uma prioridade intermédia, ou seja, pode interromper a *thread* principal e ser interrompida pela *thread* do *can*.

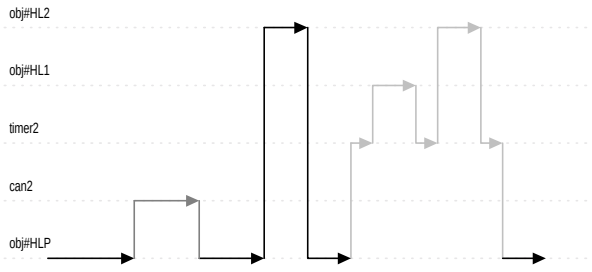

Figura 5 - Diagrama vectorial do fluxo de execução de um FMOTS.

A fig. 6 ilustra o escalonamento global de um FMOTS detentor de vários objectos principais. É possível observar a existência das três *threads* referidas anteriormente: (1) a *thread* principal que, à custa da interrupção de *software* do *Timer 0*, é multiplexada no tempo pelo escalonador *VASch* para permitir o processamento dos algoritmos de supervisão dos diversos equipamentos e/ou processos industriais que se encontrem sob a responsabilidade do FMOTS; (2) a *thread* do *timer2* que, à custa da interrupção de *software* do *Timer 2*, chama os objectos de alto-nível necessários para tratar os dados provenientes das redes de comunicações; (3) a *thread* do *can* que permite receber dados das duas portas *CAN* existentes, a interna (à custa da interrupção de *software* do *Can*) e a externa (à custa da interrupção de *hardware* do *Int 0*).

O *VASch* realiza o escalonamento dinâmico, preemptível e com prioridades estáticas dos objectos *controller* (agregado de objectos responsável pelo processamento interno de um FMOTS) e *interface controller* (agregado de objectos responsável pela interface de um FMOTS com o ambiente) de cada um dos equipamentos e/ou processos em supervisão. Para tal, recorre à memória externa para guardar os contextos computacionais das tarefas (objectos a escalonar), gerindo dois segmentos de memória: (1) no segmento *XSTACK*, dividido em tantos blocos quantas as tarefas, são guardadas as variáveis locais e os parâmetros dos métodos pertencentes aos objectos a escalonar; (2) no segmento *XMEM* é guardada: (i) uma lista (vector de estruturas) com os descritores das tarefas, onde constam, por exemplo, os valores do SP (*stack pointer*) e do XSP (*external stack pointer*) de cada uma das tarefas; (ii) uma lista com as *stacks* internas de todas as tarefas, onde constam, por exemplo, os valores de todos os registos especiais do processador (SFR - *special function registers*).

É importante referir que os métodos pertencentes a objectos de baixo-nível (não escalonáveis) são invocados em regiões críticas, uma vez que correspondem a operações que lidam com os recursos básicos da arquitectura alvo e, como tal, são não preemptíveis. Todos os outros métodos (pertencentes a objectos de alto-nível escalonáveis) são preemptíveis

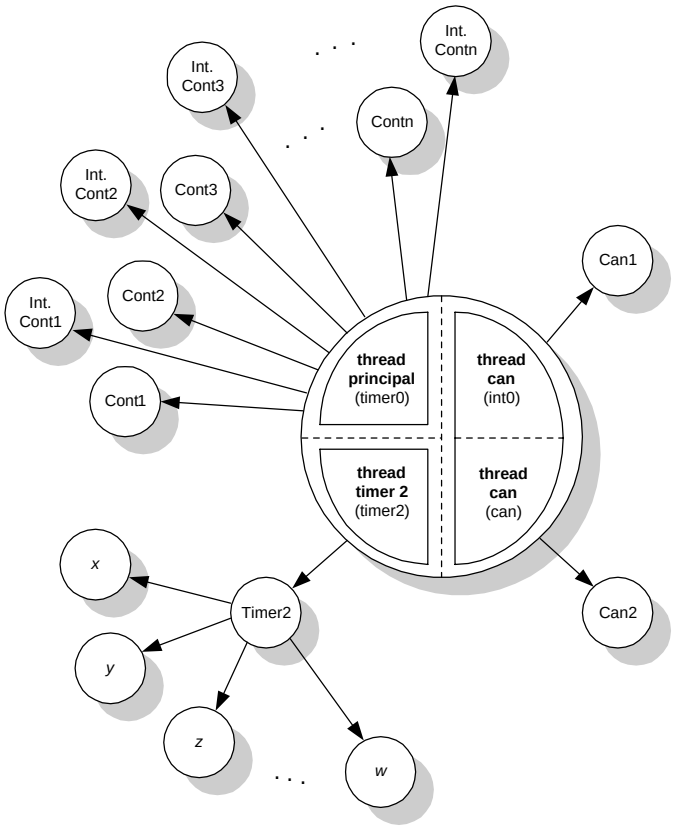

Figura 6: Escalonamento global de um FMOTS.

entre si, desde que no processo de compilação se garanta a geração de código reentrante, ou seja, desde que não existam sobreposições de endereços físicos entre os vários blocos do segmento *XSTACK*. Apesar destes dois últimos cenários típicos parecerem os únicos possíveis no âmbito do escalonamento global ilustrado na fig. 6, é frequente métodos de objectos de alto-nível serem invocados mas no âmbito de uma tarefa que está a escalonar um objecto distinto do primeiro. Nestas circunstâncias, o método deve ser executado dentro de uma região crítica, para que a compilação não gere código reentrante e se garanta que esta invocação não vai ser executada com o contexto computacional do mesmo método quando ele está ser executado no âmbito da tarefa que escalona o objecto a que ele pertence.

Estruturação do Código

Aquando da síntese, em línguagem *C*, da aplicação embebida, por cada classe de alto nível, são criados dois ficheiros (*className.h* e *className.c*). No primeiro são declarados os protótipos dos métodos da classe, bem como a declaração de duas estruturas de dados (*struct className* e *struct static_className*). A primeira estrutura contém as variáveis da classe, enquanto que a segunda estrutura contém as variáveis estáticas da classe (também chamadas variáveis de classe). O segundo ficheiro inclui a implementação dos métodos classe. Os nomes dos métodos são transformados de *className.methodName()* para *className_methodName().*

Poderão ser criadas várias instâncias da mesma classe (múltiplos objectos). A estrutura *className* é vectorizada de forma a que cada índice (*InstanceIndex*) do vector *className* corresponda a uma instância do objecto. A estrutura *static_className* não necessita de ser vectorizada pois contém as variáveis estáticas da classe (ver exemplo abaixo).

Linguagem OO

```
Class XPTO
{       var int: a;
        var int: b;
        var int: num_XPTO;
        int operation (var int x, var int y)
        {       var int: n;
        This.a = x + y;
        This.b = x - y;
                n = inc();
                return = n;
        }
        int inc (object)
        {       This. num_XPTO ++;
        }
}
```

ficheiro *XPTO.h*

```
typedef struct XPTO_TAG =
{       int a;
        int b;
} XPTO_DEC;
```

```
struct static_XPTO_TAG =
{       int num_XPTO;
} static_XPTO_DEC;

int XPTO_operation (int x, int y);
int XPTO_inc (void);
```

ficheiro *XPTO.C*

```
XPTO_DEC XPTO [INSTANCEMAX];

static_XPTO_DEC static_XPTO;

int XPTO_operation (int x, int y)
{       int n;
        XPTO [InstanceIndex].a = x+y;
        XPTO [InstanceIndex].b = x-y;
        n = XPTO_inc ();
        return  n;
}

void XPTO_inc (void)
{       static_XPTO.num_XPTO ++;
}
```

Comutação de Contexto

Quando há comutação de tarefa, é necessário guardar a sua imagem, para que mais tarde, esta possa ser reposta. As variáveis do objecto escalonado são vectorizadas, logo, só é necessário alterar o índice (*InstanceIndex)* do vector *classeName*. Os Registos SFR, são movidos para a *STACK* interna, e posteriormente guardados num buffer (RTOS_*memStack [INSTANCEMAX]*) na memória externa. No momento da criação de uma instância de um determinado objecto é reservado um sub-segmento de memória no segmento *XSTACK*. Desta forma não é necessário guardar os valores contidos na *STACK* externa, bastando alterar o valor do apontador para *XSTACK*, para o início do sub-segmento da próxima tarefa.

O VASch dispõe de um conjunto de primitivas inseridas automaticamente aquando da síntese de software, destacando-se as seguintes:

```
// iniciação das variáveis de controlo
void RTOS_initialize ();

// criação de instâncias
BYTE RTOS_create (timeTask,InstanceIndex, (void *) ptrFunc);

// arranque do escalonador
void RTOS_start (void);

// início de zona crítica
RTOS_criticalZone (void);

// fim de zona crítica
RTOS_endCriticalZone (void);

// activar  uma tarefa
RTOS_start (BYTE PID);

// suspender uma tarefa
RTOS_stop (BYTE PID);

// terminar  uma tarefa
RTOS_kill (BYTE PID);

// envio de um sinal
RTOS_sendSignal (BYTE PID, BYTE signal);

// espera de um sinal
RTOS_waitSignal (BYTE signal);

// pedido de comutação de tarefa
RTOS_switchContext (void);
```

5. CONCLUSÕES

Para aplicações embebidas a concepção de RTOS deve ter em conta as limitações de memória e de processamento que são impostas pelas arquitecturas alvo tipicamente utilizáveis, sendo frequente adoptar *kernels* dedicados.
Neste artigo apresentou-se uma solução de escalonamento para aplicações embebidas concebidas segundo a abordagem da orientação aos componentes. O *kernel* descrito (o VASch) possibilita a supervisão de duas dezenas de processos industriais independentes num único processador da família MCS-51.

6. BIBLIOGRAFIA

[1] B. H. Lee, *Embedded Internet Systems: Poised for Takeoff*, IEEE Internet Computing, pp. 24-29, Maio/Junho, 1998.

[2] L. Werner, *Embedded Operating Systems Face Greater Productivity Demands*, Electronic Design, pp. 51-60, Outubro, 1998.

[3] *IEEE POSIX 1003.4: Real-Time Extensions to POSIX.*

[4] R. M. Stein, *Real-Time POSIX*, Byte, pp. 177-82, Agosto, 1992.

[5] *Real-Time LINUX*, http://www.rtlinux.org/.

[6] J. Hardin, *Inside Real-Time LINUX*, Dr. Dobb's Journal, pp. 72-78, Março, 2000.

[7] E. A. Lee, *What's Ahead for Embedded Software?*, IEEE Computer, vol. 33, no. 9, pp. 18-26, Setembro, 2000.

[8] R. G. Landman, *Selecting a Real-Time Operating System*, Embedded Systems Programming, pp. 79-95, Abril, 1996.

[9] J. Stankovic, M. Spuri, M. Di Natale, G. Buttazzo, *Implications of Classical Scheduling Results for Real-Time Systems*, IEEE Computer, vol. 28, no. 6, pp. 16-25, Junho, 1995.

[10] B. Kauler, *Flow Design for Embedded Systems*, 2nd Edition, R&D Books, 1999.

[11] M. Cornero, F. Thoen, G. Goossens, F. Curatelli, *Software Synthesis for Real-Time Information Processing Systems*, em P. Marwedel, G. Goossens (eds), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[12] F. Balarin, L. Lavagno, P. Murthy, A. Sangiovanni-Vincentelli, *Scheduling for Embedded Real-Time Systems*, IEEE Design & Test of Computers, pp. 71-82, Janeiro/Março, 1998.

[13] C. Pfister, C. Szyperski, *Why Objects Are Not Enough?*, T. Jell (ed.), *CUC96: Component-Based Software Engineering*, Managing Object Technology Series, Cambridge University Press, SIGS Books, 1998.

[14] P. Wegner, *Dimensions of Object-Based Language Design*, 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), pp. 168-182, 1987.

[15] R. J. Machado, J. M. Fernandes, H. D. Santos. *A Methodology for Complex Embedded Systems Design: Petri Nets within a UML Approach*. Architecture and Design of Distributed Embedded Systems, Kluwer Academic Publishers, Boston, E.U.A. [a publicar em 2001].

[16] J. M. Fernandes, R. J. Machado, H. D. Santos *Modeling Industrial Embedded Systems with UML*. 8th IEEE/IFIP/ACM International Workshop on Hardware/Software Co-Design (CODES'00), pp. 18-22, San Diego, E.U.A., Maio, 2000, ACM Press.

[17] R. J. Machado, *Metodologias de Desenvolvimento em Projectos de Engenharia de Computadores no Suporte à Implementação de Sistemas de Informação Distribuídos Não Convencionais (Industriais)*, Tese de Doutoramento em Informática / Engenharia de Computadores, Escola de Engenharia, Universidade do Minho, Braga, Novembro, 2000.