ReTMiK

(Real-Time Micro-mouse Kernel)

Breve manual

Luís Almeida, 2003/10/03

Bruno Gravato, Bruno Pereira, 2004/01/02

1- Introdução

O kernel ReTMiK foi desenvolvido em finais de 1997 para o Concurso Micro-Rato da UA, tendo sido disponibilizado aos participantes a partir da edição de 1998. Funciona sobre a plataforma kit188 (também referida como DET188) e foi sompletamente desenvolvido em linguagem C usando o TurboC 2.0. Este kernel *multitasking* permite definir tarefas periódicas e com relação de fase, as quais são activadas automaticamente, de forma transparente para o utilizador. O código das tarefas é reentrante e o kernel permite preempção. Utiliza um escalonador de tempo-real baseado, inicialmente, em prioridades fixas indexadas inversamente ao período das tarefas (*rate monotonic*). Posteriormente generalizou-se para Deadline Monotonic (inversamente à deadline da tarefa) e acrescentou-se EDF (Earliest Deadline First), proporcional à distância à deadline da tarefa no instante de decisão. Está particularmente adaptado a suportar o desenvolvimento de programas de controlo de robôs baseados em comportamentos autónomos, constituíndo uma ajuda preciosa ao programador. O kernel ReTMiK foi utlizado por várias equipas nas edições de 1998 a 2000. Após essas edições, por a plataforma Kit188 ter caído em desuso, deixou de ser usado de forma ampla. A excepção foi a equipa Bulldozer, que utilizou este kernel desde a sua primeira participação em 1998, até 2003. Durante estas 6 edições venceu em 1998 e 2001 e classificou-se sempre entre os primeiros 4. O maior robô que utilizou o ReTMiK foi o RUAv3, que representou a UA no FIST'98 em Bourges, França.
Em 2003 no âmbito do trabalho final da cadeira de STR (Sistemas de Tempo Real) leccionada pelo professor Luís Almeida foi modificado, pelos alunos Bruno Gravato e Bruno Pereira, tendo sido acrescentadas novas funcionalidades.

2 - A livraria *retmik.obj*

Esta livraria contém as funções de um *kernel* (ou executivo) *multitasking* e tempo-real que facilita a programação do robot baseada em tarefas independentes e periódicas. Basicamente, este kernel permite transformar normais funções de C em tarefas periódicas cuja activação e execução concorrente é controlada pelo próprio kernel e completamente transparente para o utilizador. A resolução temporal do *kernel* (tempo que demora um *tick*) bem como o número máximo de tarefas permitido são parâmetros de entrada da função de inicialização *init_system()*. É possível agrupar tarefas para posteriormente poderem ser iniciadas ou terminadas em simultâneo, cada grupo têm um número máximo de tarefas definido estaticamente no kernel (actualmente esse valor é 10).
Ao criar-se cada tarefa é necessário fornecer quatro parâmetros que descrevem o respectivo período, grupo a que a tarefa pertence, deadline da tarefa e o instante da primeira activação para permitir controlar a fase de activação entre tarefas com o mesmo período. Após a inicialização do sistema com

a função *init_system()*, utilizando a função *set_sched_alg()* é possível escolher o algoritmo de escalonamento, estando definido por omissão o algoritmo Deadline Monotonic. Sempre que uma tarefa de maior prioridade fica activa durante a execução de outra de menor prioridade esta última é interrompida de modo a que a primeira possa usar o CPU, isto se a preempção estiver activa (o que acontece por omissão), é possível em qualquer momento (des)activar a preempção com as macros *set_preempt()* e *set_nopreempt()*. Após a terminação da tarefa de maior prioridade a de menor prioridade reata a sua execução no ponto de interrupção. Assim, note-se que a tarefa de maior prioridade no sistema nunca é interrompida mas as restantes podem ser. As várias tarefas podem comunicar através de variáveis globais. A gestão de memória é feita dinamicamente com recurso a um par de funções *get_mem()* e *free_mem()* que funcionam de forma semelhante às funções *malloc* e *free* do DOS.

2.1 - Funções disponíveis

void init_system (int tick_in_ms, int n_max_task);

Inicía o *kernel*, i.e., as respectivas estruturas internas bem como as interrupções de relógio e o respectivo *handler*. Note-se que o sistema usa o *timer2* do 188 de forma que este *timer* não pode ser usado por nenhuma tarefa. O parâmetro de entrada *tick_in_ms* especifica a duração dos *ticks* em milisegundos (na versão do Kit188 a 5MHz recomenda-se que seja maior ou igual a 10 por questões de *overhead*). Este parâmetro define a resolução temporal de todo o sistema. O parâmetro n_max_tasks define o número máximo de tarefas que o sistema admite. Este valor deverá ser igual ou superior ao número de tarefas efectivamente usadas no programa.

int create_task (TASK_DESC *task_descript);

Cria uma tarefa deixando-a em estado *sleep_forever*, i.e., a passagem do tempo não é contabilizada para a respectiva activação periódica, logo a tarefa nunca é activada enquanto estiver neste estado.

O parâmetro de entrada é um ponteiro para uma estrutura do tipo TASK_DESC (task_descriptor) que possui os seguintes campos:

```
typedef struct {
    void (*func_entry)();

        ponteiro para a função com o código que a
        tarefa deve executar

     int period;

        período de
        activação em
        ticks >=10

    int first;

        instante da primeira activação relativa ao
        "acordar da tarefa", em ticks >=1

    int deadline;

        deadline da tarefa, em ticks >=1

    int group;

        grupo a que pertence a tarefa, >=1
        número de tarefas <= 10

    int stack_size;

        tamanho de stack que a tarefa necessita
        >=32 bytes
        (recomenda-se 64)


    } TASK_DESC;
```

O valor devolvido é a identificação da tarefa (*TID*) se >=0. Se <0 então é um parâmetro de erro.

int delete_task (int id_task);

Esta função permite eliminar uma tarefa do kernel, que tenha sido previamente criada com a função create_task(). Se a tarefa estiver em execução apenas é apagada quando terminar. Se o id for inválido é retornado o valor de erro ERR_INV_TID, caso contrário é devolvido o valor OK. O bloco de memória

ocupado por esta tarefa fica livre para ser usado por um novo create_task().

void start_all (void);

Conforme dito atrás, quando uma função é criada é colocada no estado de *sleep_forever*, i.e., com a execução periódica desactivada (a tarefa nunca executa enquanto estiver neste estado). Esta função *start_all* permite iniciar sincronamente a execução periódica das tarefas. Isto não quer dizer que as tarefas iniciem execução logo após *start_all*. Quer dizer, sim, que a contagem do tempo (em *ticks*) para a primeira activação de todas as tarefas (que estavam em *sleep_forever*) começa assim que se chama a função *start_all*.

void start_task (int id_task);

Esta função permite iniciar a execução de uma terefa, que já tenha sido previamente inicializada com a função *create_task()*. Para tal deve ser dado como parâmetro de entrada o id da tarefa (devolvido pela função *create_task()*). Na primeira execução é tido em conta o instante da primeira activação definido no *task_descriptor*.

void stop_task (int id_task);

Esta função permite parar a execução de uma determinada tarefa (previamente criada com o *create_task()*), passando ao estado *sleep forever* após terminar a sua execução. Se o id for inválido, não acontece nada.

int start_group (int group);

Esta função permite iniciar sincronamente a execução periódica de um grupo de tarefas.
O número máximo de tarefas num grupo é definido estaticamente no kernel e actualmente tem o valor 10.
As tarefas são iniciadas tendo em conta o tempo da primeira execução especificado no task_descriptor no momento da sua criação.

int stop_group (int group);

Esta função permite terminar com a execução periódica de um grupo de tarefas. Assim a partir do momento que esta função é chamada as tarefas pertencentes ao grupo especificado no parâmetro de entrada vão ser terminadas e colocadas no estado sleep_forever (após terminarem a sua execução).

int sleep_task (int sleep_tick s);

A função *sleep_task* permite que uma tarefa se autosuspenda por um determinado período de tempo especificado no parâmetro de entrada, em *ticks*. Quando
o período de auto-suspensão termina o sistema acorda a tarefa que fica novamente pronta para continuar execução. Neste caso o valor retornado é 0.

void set_sched_alg (char scheduling);

Esta função permite especificar o algoritmo de escalonamento a ser utilizado.
Estão disponíveis dois algoritmos de escalonamento: DM (Deadline Monotonic) e EDF (Earliest Deadline First).
O parâmetro de entrada deve ser um dos seguintes:

S_DM - para Deadline Monotonic
S_EDF - para Earliest Deadline First

É possível alterar o algoritmo de escalonamento em qualquer momento, mesmo durante a execução das tarefas (embora nos instantes seguintes o comportamento do sistema possa não ser imediatamente o esperado).
Na função init_system() é definido o DM por omissão, caso se pretenda usar outro, deve ser especificado após a chamada desta função.

char get_status (int id_task);

Esta função permite obter o estado de execução da tarefa com id id_task.
O valor devolvido é um dos seguintes:

T_STARTING
T_READY
T_IDLE
T_SLEEP
ERR_INV_TID (caso o id seja inválido)

void *get_mem (unsigned size_req);

O pedido de memória ao sistema faz-se através desta função que funciona de forma semelhante à função *malloc* do DOS. A função devolve um ponteiro para um bloco contíguo de tamanho igual ao indicado. O parâmetro de entrada é o número de bytes necessários. O parâmetro de saída é o ponteiro para o bloco pedido. No caso de erro o valor devolvido é NULL.

void *free_mem(void *blk_to_release);

Esta função permite devolver ao sistema um bloco de memória anteriormente pedido com a função *get_mem()*. O parâmetro de entrada é o ponteiro para o bloco a devolver que tem que ser coincidente com o ponteiro obtido com a função *get_mem*.

2.2 – Macros disponíveis

long get_abs_ticks ()

Esta macro devolve o número de ticks que ocorreram desde o início do funcionamento do sistema. Pode ser utilizada para medições temporais (em ticks) quer absolutas quer relativas calculando a diferença entre o respectivo valor em dois instantes diferentes.

int get_id ()

Esta macro devolve o identificador da tarefa que a invoca. Os identficadores são atribuídos pela função *create_task()* sequencialmente, a partir do identificador 1 (o *main* ou tarefa de sistema, tem sempre o identificador reservado 0).

char get_deadl_stat ()

Esta macro devolve o estado da tarefa em termos de prazo de execução (*deadline*). Se a tarefa ainda está dentro do respectivo prazo (neste caso igual ao período) é devolvido 0. Se a tarefa se atrasou para além do prazo (o que implica que perdeu pelo menos uma activação periódica) então esta macro devolve um valor diferente de 0.

char get_CPU_util ()

Esta macro devolve uma aproximação da carga actual do CPU, entre 0 e 100. Durante um intervalo pré-definido (em ticks) conta o nº de ticks em que não há tarefas para executar (no início do tick) e apresenta a respectiva razão (multiplicada por 100).

get_sched_alg()

Retorna qual o algoritmo de escalonamento que está a ser utilizado (ver set_sched_alg).

set_preempt()

Coloca o sistema com preempção, o que possibilita tarefas com maior prioridade interromper as de menor prioridade que estejam a ser executadas (definição por omissão na função *init_system()*).

set_nopreempt()

O sistema fica sem preempção, ou seja quando uma tarefa está a ser executada não pode ser interrompida.

A preempção pode ser (des)activada em qualquer momento.

get_preempt ()

Devolve o estado da preempção:
1 – com preempção
0 – sem preempção

2.3 - Utilização do kernel *ReTMiK*

Para utlizar o *kernel* retmik deverá ter em atenção o seguinte:

- não usar o *timer2* do µP.
- em geral, não inibir as interrupções durante a execução das tarefas já que isso levaria à suspensão da contagem temporal do sistema e a erros nos períodos de activação das tarefas. Contudo, a inibição pode ser utilizada durante pequenos intervalos, por exemplo para garantir o acesso atómico a estruturas (ou variáveis) partilhadas.
- incluir o ficheiro *retmik.h* com as definições e declarações necessárias à utilização do sistema.

• escrever o código das tarefas sem ciclos globais. A repetição periódica das tarefas é completamente controlada pelo sistema.

• por seu lado, o *main()* deverá começar por chamar a função *init_system()*, criar as tarefas necessárias chamando a função *create_task()*, disparar a activação periódica das tarefas com *start_all()* e entrar num ciclo infinito tipo *while(1);*. Repare-se que o processamento necessário deve ser todo executado ao nível das tarefas. O que for executado dentro do ciclo infinito corre em *background*, apenas quando não há tarefas para executar e sofre sempre preempção mesmo que o sistemas esteja sem preempção (set_nopreempt()).

• para não aumentar demasiado o tempo de execução das tarefas dever-se-á ter o cuidado de não efectuar bloqueios dentro destas, como por exemplo, esperar pela recepção ou transmição de um caracter pela porta série, ou fazer um ciclo infinito.

3 - Geração de código

O programa de controlo do robot, depois de escrito em C, deverá ser compilado usando o TurboC 2.0 incluindo na *linkagem* os ficheiros **stup.obj** e **retmik.obj** (e outras livrarias eventualmente necessárias, e.g. *robot.obj*) O ficheiro resultante, tipo *.exe*, deverá ser preparado para *download* usando o *code* re*locator* **exe2kit**. Esta operação gera um ficheiro com o código a ser carregado na placa, com o tipo *.kit*. Este procedimento é feito de forma automática pela *makefile* **mk_retmk.bat**.. Ter em atenção que esta *makefile* usa o re*locator* para preparar o código para ser carregado a partir do endereço 0x0010:0x0000. Depois de carregar o código usando o comando *L10:0 "progname.kit"* do monitor, deverá iniciar o programa com o comando *G10:0*.