

Diseño de un planificador

Área: Sistemas Operativos de Tiempo Real
José Hernández Carretero y Alberto Valverde Carretero

Introducción

El planificador es el elemento fundamental del sistema operativo en tiempo real (SOTR). Revisa las prioridades de las tareas que quieren ejecutarse y selecciona la más prioritaria. Para ello ha de controlar el estado de las tareas y decidir cuál hará uso del procesador.

Nuestro trabajo consiste en el diseño de un algoritmo de planificación que determine el orden de acceso de las tareas al procesador. Se basa en el método de planificación expulsiva 'primero el más urgente' (earliest deadline first) que se denota por sus siglas en inglés EDF. Las tareas a ejecutar se despacharán por orden de sus respectivos tiempos límite (plazos absolutos).

La primera tarea que se ejecuta es la más urgente, aquella cuyo plazo vence antes. Por ello es necesario calcular los tiempos límite durante la ejecución. Esta reseña es la que hace que el planificador sea caracterizado como un 'planificador dinámico'. Los planificadores dinámicos poseen un núcleo de multiprogramación más complejo lo que implica unos tiempos de proceso mayores. La característica mencionada hace que el SOTR requiera una buena parte del tiempo el uso del procesador.

Para intentar paliar el problema analizado en el párrafo anterior se programará el planificador en lenguaje ensamblador y así minimizar el tiempo de proceso. En concreto la programación se realizará para el microcontrolador HC08 cuyas características vienen ampliamente detalladas en el manual de usuario.

A pesar de programar para el HC08 el planificador presentado se podrá utilizar en cualquier microcontrolador con un juego de instrucciones compatible. Muchos microcontrolador ya incluyen un módulo planificador; el algoritmo diseñado será especialmente útil para aquellos a los que se le quiera dotar de un pequeño SOTR basado en planificación EDF por software.

Modelo de programación

Para poder utilizar el planificador diseñado se hace necesario realizar ciertos cambios en la forma habitual de programar para sistemas empujados. La diferencia principal radica en la sustitución del programa principal por un simple bucle infinito 'while(1);'. Se dividirá el código en fragmentos que formarán tareas las cuales se ejecutarán tras la aparición de algún evento que las active. Como es lógico, habrá que incluir el fichero 'edf.c' en el proyecto y escribir la línea '#include "edf.h"' al comienzo de todos los ficheros que describan tareas.

Cualquier evento será producido por una interrupción ya que es la única forma de salir del bucle infinito de forma que cualquier módulo del microcontrolador pueda requerir el uso del procesador para ejecutar una tarea. Las rutinas de atención a las interrupciones (ISR) se programarán de la siguiente forma:

1. Borrar el flag que produjo la interrupción
2. Activar eventos mediante la función 'thread_task'
3. Ejecutar la macro 'end_interrupt'

Serán una excepción las interrupciones muy prioritarias que posean una ISR muy corta (como pueden ser las del módulo SCI), en este caso será preferible no asignarles ninguna tarea para no perder tiempo en cambios de contexto.

Las tareas se programarán como funciones normales a las que se añadirá al final la macro 'end_task'. A cada tarea se le asignará un número que será el que se le pase como parámetro a la función 'thread_task'. Además es imprescindible incluir la tarea en el array 'init_cpu_state' que posee las direcciones de comienzo en memoria ROM de los fragmentos de código. Por último se calculará el plazo límite de finalización como un número entero de veces el periodo de la interrupción periódica y se indicará en el array 'init_dead_time'.

La interrupción periódica (ITR) jugará un papel muy importante, será la responsable de realizar los siguientes encargos:

- Sondeos necesarios para producir eventos
- Eventos periódicos (ya sean condicionados o no)
- Llevar la cuenta del tiempo
- Reiniciar el WatchDog

El periodo de la ITR se tendrá que elegir cuidadosamente teniendo en cuenta la periodicidad de los eventos que activa y no debe superar en ningún caso el tiempo que tarda el WatchDog en reiniciar el sistema.

Planificador

Este pequeño SOTR tiene una forma sencilla y muy efectiva de actuar:

Posee una cola con todas las tareas pendientes de ejecución. Al producirse un nuevo evento y por lo tanto una nueva tarea que ha de ser ejecutada se introduce esta última en la cola adelantando a todas aquellas tareas que sean menos urgentes (menor tiempo límite de finalización). El procesador ejecuta por orden todas las tareas de la cola de forma que cuando la cola queda vacía vuelve al bucle infinito.

A continuación se incluyen los ficheros necesarios para programar con el planificador. Conviene leer los comentarios para entender su funcionamiento. Para un funcionamiento correcto es imprescindible que al iniciar una tarea y justo antes de ejecutar 'end_task' la pila no haya sido modificada. Lo normal es que sea suficiente con poner el 'end_task' justo al final de la tarea pero hay compiladores que tienen una política diferente de funcionamiento como el CodeWarrior en el que hay que activar 'Disable any low level common subexpression elimination' en las opciones de optimización del compilador. Además habrá que modificar las definiciones de 'TSK_QU', 'TSK_DES' y 'DEAD_TIME' en el caso de estar mapeadas en direcciones RAM diferentes (en CodeWarrior se deberá mirar en el fichero '.map' que se crea tras la compilación y buscar las variables 'task_queue', 'task_displaced' e 'init_cpu_state').

edf.h

```
// The routines will only work for less than 127 tasks
// !!! In CodeWarrior check P&E FCS Settings/Compiler for HC08/Options/Optimizations/Disable any
low level common subexpression elimination
```

```
#ifndef _EDF_H // Conditional definition of _EDF_H
#define _EDF_H
```

```
//Variable type
typedef unsigned char byte;
typedef unsigned int word;
```

```
// Declaration of EDF functions
void thread_task (byte thread_tsk);
void Config_EDF (void);
```

```
// Definition of tasks
#define N 4 // Number of tasks
#define Infinite_Loop 0 // Task 0: Infinite Loop
#define TASK1 1
#define TASK2 2
#define TASK3 3
```

```
// Struct declaration
typedef struct // Bytes de estado que se guardan en pila
{
    byte H; // Index register (High byte)
    byte CCR; // Condition code register
    byte A; // Accumulator
    byte X; // Index register (Low byte)
    word PC; // Program counter
} stack;
```

```
// Declaration of task functions
void task_0_Infinite_Loop (void);
void task_1_TASK1 (void);
void task_2_TASK2 (void);
void task_3_TASK3 (void);
```

```
extern byte aux; // Auxiliar byte
extern byte task; // Actual task
extern byte task_ptr; // 'task' pointer in queue
extern byte queue_ptr; // Pointer to last task in queue
extern byte task_queue[0x10]; // Queue of pending tasks
extern byte task_desplaced[N]; // Tasks that have been displaced
extern byte task_dead_time[N]; // In this time the task must be finished
extern stack task_cpu_state[N]; // Displaced tasks state
extern byte counter; // Counter incremented in ITR
```

```
extern const byte init_dead_time[N]; // Initial time to finish task
extern const word init_cpu_state[N]; // Initial state of tasks
```

```

#define TSK_QU          0x42          // !!! Address of 'task_queue' (Look '.map' file)
#define TSK_DES        0x52          // !!! Address of 'task_desplaced' (Look '.map' file)
#define DEAD_TIME      0x60          // !!! Address of 'init_cpu_state' (Look '.map' file)

// Assembly routines
#define end_interrupt()                /* Run to finish an interrupt          */\
    save_context();                  /* Save CPU state                       */\
    load_pc();                        /* Load Program Counter                 */

#define end_task()                    /* Run to finish a task                 */\
    init_context();                  /* Reset task_cpu_state                 */\
    next_task();                     /* Calculate next task to process       */\
    load_context();                  /* Load CPU state                       */

#define thread()                      /* Add pending task to queue           */\
    asm CLRH;                         \
    asm LDX 1,SP;                      \
    asm LDA DEAD_TIME,X;              \
    asm PSHA;                          \
    asm LDA queue_ptr;                \
    asm PSHA;                          /* task_ptr = queue_ptr                */\
    asm INCA;                          /* queue_ptr++                          */\
    asm AND #0x0F;                     \
    asm STA queue_ptr;                \
    asm BRA Label1;                   /* Jump to Label1                       */\
    asm Label2;                        /* Label2                                */\
    asm TXA;                           \
    asm LDX aux;                       \
    asm STA TSK_QU,X;                 \
    asm Label1;                        /* Label1                                */\
    asm PULA;                          \
    asm STA aux;                       \
    asm DECA;                          /* task_ptr--                            */\
    asm AND #0x0F;                     \
    asm PSHA;                          \
    asm TAX;                           \
    asm LDX TSK_QU,X;                  /* X = dead_time[queued_task]          */\
    asm LDA 2,SP;                      /* A = dead_time[thread_tsk]            */\
    asm CMP DEAD_TIME,X;              /* If X < A                              */\
    asm BLO Label2;                   /* Jump to Label2                       */\
    asm LDX aux;                       \
    asm AIS #2;                        \
    asm LDA 1,SP;                      \
    asm STA TSK_QU,X;                  /* task_queue[task_ptr] = thread_tsk   */

#define save_context()                /* Save CPU state                       */\
    asm CLRH;                         \
    asm LDX task_ptr;                  \
    asm LDA TSK_QU,X;                  \
    asm CMP task;                      /* If (task == task_queue[task_ptr])   */\
    asm BNE Label3;                   /* Return                                */\
    asm PULH;                          \
    asm RTI;                           \

```

```

asm Label3;;                                /* Label3                                */\
asm LDX task;                               \|
asm STA task;                               /* task = task_queue[task_ptr]          */\
asm TSTX;                                   /* If (task = 0)                         */\
asm BNE Label4;                             /* Jump to Label5                       */\
asm AIS #6;                                  \|
asm BRA Label5;                             \|
asm Label4;;                                \|
asm STX TSK_DES,X;                          /* task_desplaced[task] = 1             */\
asm LDA #0x06;                              /* task_cpu_state[task]                 */\
asm MUL;                                     \|
/* asm PSHX;   For more than 42 tasks */\
/* asm PULH;   For more than 42 tasks */\
asm TAX;                                     \|
asm PULA;                                    /* Pull registers from Stack            */\
asm STA @task_cpu_state,X;                  \|
asm PULA;                                    \|
asm STA @task_cpu_state+1,X;                \|
asm PULA;                                    \|
asm STA @task_cpu_state+2,X;                \|
asm PULA;                                    \|
asm STA @task_cpu_state+3,X;                \|
asm PULA;                                    \|
asm STA @task_cpu_state+4,X;                \|
asm PULA;                                    \|
asm STA @task_cpu_state+5,X;                \|
asm Label5;;                                /* Label5                                *//

#define load_pc()                            /* Load Program Counter                */\
/* asm CLRH;   For more than 42 tasks */\
asm LDX task;                               \|
asm LSLX;                                   /* init_cpu_state[task]                 */\
asm LDA @init_cpu_state,X;                 /* Load PCH                             */\
asm PSHA;                                    \|
asm LDX @init_cpu_state+1,X;               /* Load PCL                             */\
asm PULH;                                    \|
asm CLI;                                    /* Enable interrupts                    */\
asm JMP ,X;                                 /* Return                                *//

#define init_context()                       /* Reset task_cpu_state                 */\
asm SEI;                                    /* Disable interrupts                    */\
asm CLRH;                                    \|
asm LDX task;                               \|
asm LDA @init_dead_time,X;                 /* dead_time[task] = init_time[task]    */\
asm STA DEAD_TIME,X;                       \|
asm CLR TSK_DES,X;                         /* task_desplaced[task] = 0             *//

#define next_task()                          /* Calculate next task to process        */\
asm LDX task_ptr;                          \|
asm CLR TSK_QU,X;                          /* task_queue[task_ptr] = 0            */\
asm TXA;                                    \|
asm INCA;                                   /* task_ptr++                            */\
asm AND #0x0F;                              \|
asm STA task_ptr;                          \|

```

```

asm TAX; \
asm LDX TSK_QU,X; /* task = task_queue[task_ptr] */
asm STX task;

#define load_context() /* Load CPU state */
asm TST TSK_DES,X; /* If task_desplaced[task] != 0 */
asm BNE Label6; /* Jump to Label6 */
asm LSLX; /* init_cpu_state[task] */
asm LDA @init_cpu_state,X; /* Load PCH */
asm PSHA; \
asm LDX @init_cpu_state+1,X; /* Load PCL */
asm PULH; \
asm CLI; \
asm JMP ,X; /* Return */
asm Label6; /* Label6 */
asm LDA #0x06; /* task_cpu_state[task] */
asm MUL; \
/* asm PSHX; For more than 42 tasks */
/* asm PULH; For more than 42 tasks */
asm TAX; \
asm LDA @task_cpu_state+5,X; /* Load PCL */
asm PSHA; \
asm LDA @task_cpu_state+4,X; /* Load PCH */
asm PSHA; \
asm LDA @task_cpu_state+3,X; /* Load X */
asm PSHA; \
asm LDA @task_cpu_state+2,X; /* Load A */
asm PSHA; \
asm LDA @task_cpu_state+1,X; /* Load CCR */
asm PSHA; \
asm LDA @task_cpu_state,X; /* Load H */
asm PSHA; \
asm PULH; /* Return */
asm RTI;

#define decrement_time() /* task_dead_time-- */
asm CLRH; \
asm LDA task_ptr; \
asm Label7; \
asm TAX; \
asm LDX TSK_QU,X; \
asm BEQ Label9; \
asm DEC DEAD_TIME,X; /* task_dead_time[task]-- */
asm BPL Label8; \
asm INC DEAD_TIME,X; \
asm Label8; \
asm INCA; \
asm AND #0x0F; \
asm BRA Label7; \
asm Label9;
#endif _EDF_H

```

edf.c

```
#include "edf.h"

byte counter; // Contador
const byte init_dead_time[N] = {0,5,7,4}; // Plazos iniciales de finalización
const word init_cpu_state[N] = // Estados iniciales de las tareas
{
    (word)task_0_Infinite_Loop+3,
    (word)task_1_TASK1,
    (word)task_2_TASK2,
    (word)task_3_TASK3
};

void thread_task (byte thread_tsk) // Añadir nueva tarea pendiente
{
    thread_tsk &= 0x7F; // Bulo para que se meta 'thread_tsk' en pila
    thread();
}

void Config_EDF (void)
{
    byte i;

    if(task_queue[0] && task_desplaced [0] && task_dead_time[0])
    { // Bulo para que se compilen las tareas
        // Este código nunca llega a ejecutarse
        task_0_Infinite_Loop();
        task_1_TASK1();
        task_2_TASK2();
        task_3_TASK3();
        task &= 0x7F;
    }

    for(i=0; i<N; i++) // Copiar estados iniciales
        task_dead_time[i] = init_dead_time[i];
}

void task_0_Infinite_Loop (void) // Tarea 0: Bucle infinito
{
    asm AIS #2; // Dejar la pila vacía
    asm CLI; // Habilitar interrupciones
    while (1); // Bucle infinito
}

#pragma DATA_SEG MY_ZEROPAGE // Variables en direccionamiento directo
byte aux; // Byte auxiliar
byte task; // Tarea en curso
byte task_ptr; // Puntero a la tarea en curso
byte queue_ptr; // Puntero a la última tarea de la cola
byte task_queue[0x10]; // Cola de tareas pendientes
byte task_desplaced[N]; // Tareas desalojadas
byte task_dead_time[N]; // Plazos de finalización
stack task_cpu_state[N]; // Estados de las tareas
```

itr.c

```
#include "edf.h"
```

```
void Config_ITR(void)
```

```
{  
    T2SC_TOIE = 1;           // Habilitación de interrupción por Overflow  
    T2SC_PS2 = 1;           // Poner el preescaler a 2^7  
    T2SC_PS1 = 1;  
    T2SC_PS0 = 0;  
    T2MOD = 768;           // Interrupción por overflow cada 0.02s  
    T2SC_TSTOP = 0;        // Arrancar contador libre  
}
```

```
interrupt 9 void ISR_ITR(void)
```

```
{  
    T2SC_TOF = 0;           // Borrar el flag  
  
    decrement_time();       // Decrementar tiempo en las tareas pendientes  
    asm STA COPCTL ;        // Resetear el WatchDog  
  
    counter++;  
    // Evento periódico  
    if((counter % 5) == 0) thread_task(TASK1);  
    // Evento periódico condicionado  
    if((counter % 10) == 1 && state == STATE0) thread_task(TASK2);  
    // Evento producido al finalizar una cuenta atrás  
    if (Back_counter)  
    {  
        Back_counter--;  
        if(!Back_counter) thread_task(TASK3);  
    }  
  
    end_interrupt();        // Ejecutar siempre al finalizar una interrupción  
}
```

main.c

```
#include "edf.h"
#define STATE0 0

void Config_ITR(void);

void main (void)
{
    byte state; // Byte de estado
    byte Back_counter; // Cuenta atrás para activar un evento

    // Configuración
    Config_ITR();
    Config_EDF();

    // Bucle infinito
    task_0_Infinite_Loop(); // Iniciar tarea
}

void task_1_TASK1 (void) // Tarea 1
{
    // Fragmento del código asociado a esta tarea
    end_task();
}

void task_2_TASK2 (void) // Tarea 2
{
    // Fragmento del código asociado a esta tarea
    end_task();
}

void task_3_TASK3 (void) // Tarea 3
{
    // Fragmento del código asociado a esta tarea
    end_task();
}
```