Introducción a los sistemas operativos de tiempo real (1/2)

Sistemas embebidos para tiempo real

Índice

- Hoy:
  - Introducción
  - Tareas y el planificador (scheduler)
  - Tareas y datos
  - Datos compartidos
- Próxima clase:
  - Semáforos y datos compartidos

Introducción

- RTOS es muy diferente a un OS convencional
  - La aplicación y el RTOS se enlazan juntos.
  - Los RTOS más básicos no se protegen.
  - Se pueden eliminar servicios no usados para ahorrar memoria.
- Existe una amplia oferta
  - FreeRTOS, µC/OS, eCos, QNX, Nucleus, VxWorks, LynxOS, etc.
  - WSN/IoT: TinyOS, ContikiOS
  - Algunos conformes al estándar POSIX (IEEE 1003.4)

RTOS vs. Kernel (RTK)

- Para algunos:
  - RTOS = kernel = real-time kernel (RTK)
- Para otros:
  - RTOS > kernel
  - kernel: incluye sólo los servicios más básicos
  - RTOS: incluye soporte de red, herramientas de depurado, gestión de memoria, etc.
- Para el texto y para nosotros:
  - RTOS y Kernel son sinónimos

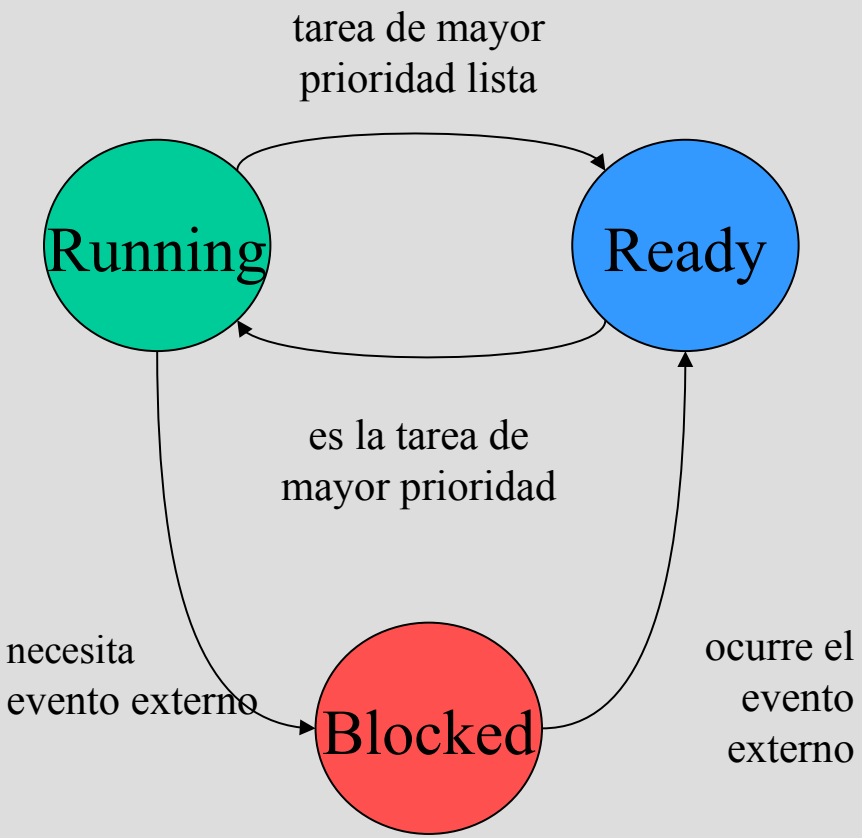
Tareas

- Tarea:
  - bloque básico de un sistema basado en un RTOS
  - una función (o varias) en C, no retorna nunca (bucle infinito).
  - número de tareas acotado sólo por memoria.
- Inicialización:
  - cada tarea se inicializa (mediante una llamada a una función del RTOS) especificando:
    - punto de entrada (función), prioridad, memoria que necesita, etc.

Ejemplo: uso de un RTOS

```
void main(void)
{
   /* Initialize (but don't start) the RTOS */
   InitRTOS();

   /* Tell the RTOS about our tasks */
   StartTask(vRespondToButton, HIGH_PRIORITY);
   StartTask(vCalculateTankLevels, LOW_PRIORITY);

   /* Actually start the RTOS.(This function never returns.) */
   StartRTOS();
}
```

Tareas: estados



- Running
  - se está ejecutando
  - una sola tarea en este estado
- Blocked
  - nada que hacer
  - esperando algún evento externo
- Ready
  - trabajo que hacer y esperando

Nota: puede haber otros estados, por ej. suspended, pended, waiting, dormant.

Planificador (scheduler)

- Planificador:
  - es la función del *kernel* que decide qué se ejecutará
- Cada tarea tiene:
  - prioridad definida y estado (running, ready, blocked)
- Planificador selecciona
  - tarea con la mayor prioridad que está lista para ejecutar.
- Transiciones:
  - tarea se bloquea sola cuando no tiene que hacer
  - tarea bloqueada pasa a *Ready*, si otra la despierta
  - conmutación entre *Ready* y *Running*: planificador

Planificador: implementación

- Schedulers en RTOS
  - sencillos
  - no trata de ser “justo”, a diferencia de Windows/Linux/Unix,
  - tareas de baja prioridad pueden “morirse de hambre”
  - responsabilidad del diseñador (no del RTOS!) asegurarse que todas las tareas cumplen con sus requerimientos

Planificador: FAQ

- ¿Cómo sabe el planificador que una tarea se ha bloqueado o desbloqueado?
- ¿Qué pasa si todas las tareas están bloqueadas?
- ¿Qué pasa si dos tareas con la misma prioridad están *ready*?
- Si mientras una tarea está ejecutándose, otra de mayor prioridad se desbloquea, ¿qué pasa?

Expropiación (Preemption)

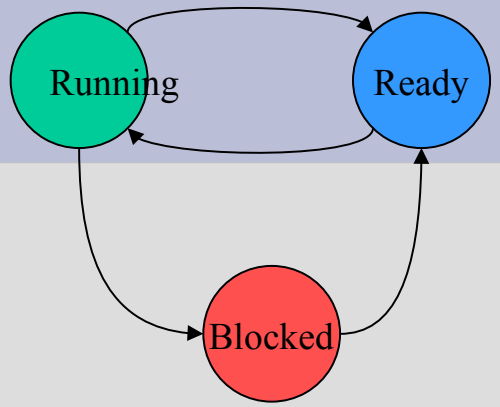
- RTOS expropiativo (preemptive RTOS)
  - detendrá la ejecución de una tarea tan pronto como una tarea de mayor prioridad pasa a *ready*.
- RTOS no-expropiativo (non-preemptive RTOS)
  - detendrá la ejecución de una tarea de baja prioridad solamente si esa tarea se bloquea (e.g., después de un *pend* or *delay*) o cede el control (*yield*)

Ejemplo de aplicación

```
struct {
   long lTankLevel;
   long lTimeUpdated;
}  tankdata[MAX_TANKS];

/* "Button Task" */
void vRespondToButton(void)
{  /* high priority task */
   int i;
   while (TRUE) {
      !! Block until button pressed
      i = !! ID of button pressed
      !! output lTankLevel
      !! output lTimeUpdated
    }
}
```

```
/* "Levels Task" */
void vCalculateTankLevels(void)
{   /* low priority task */
   int i = 0;
   while (TRUE) {
      !! read float levels in task i
      !! do bunches of calculations
      tankdata[i].lTimeUpdated =
         !! current time
       tankdata[i].lTankLevel =
         !! result of long calculation
      !! pick next tank to handle, etc.
       i = !! next tank
   }
}
```

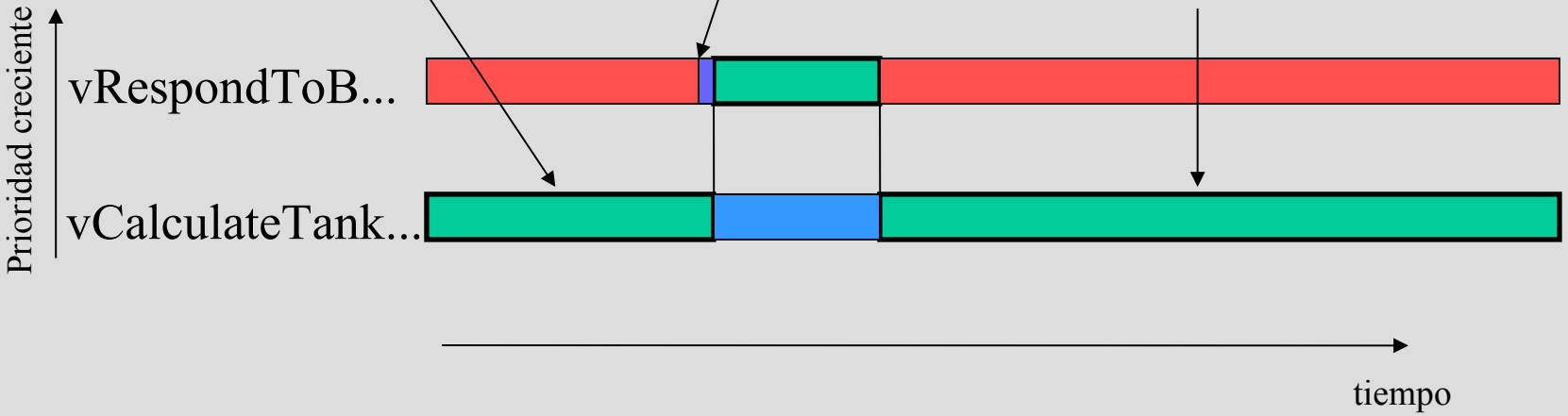
Ejemplo



Presionan un botón: RTOS le pasa el procesador a **vRespondtoButton** en cuanto **vCalculateTankLevels** queda *ready*.

vCalculateTank... ocupada haciendo calculos mientras **vRespondToB...** espera presionen un botón

VRespondToB... termina y se bloquea de nuevo. RTOS le devuelve el procesador a **vCalculateTank...**



Tareas y datos

- Cada tarea tiene su *contexto* (privado)
  - Registros
  - Contador de programa (PC)
  - Pila (Stack): variables locales
- Conservación del *contexto*
  - una tarea no sabe cuando dejará y volverá a ejecutarse
  - el contexto debe guardarse y restaurarse
- El resto de los datos -variables globales, estáticas- pueden ser compartidos entre tareas, entonces...

Tareas y datos compartidos

- Si varias tareas comparten datos, hay que tomar las mismas precauciones que vimos para otras arquitecturas:
  - Deshabilitar las interrupciones
  - Impedir la conmutación de tareas.
- Hay que ser cuidadoso, pues a veces los datos compartidos no están a la vista.

```
void Task1(void){
   ...
   vCountErrors (1);
   ...
}
void Task2(void) {
   ...
   vCountErrors (2);
   ...
}

...

void vCountErrors(int cNewErrors){
   static int cErrors;
   cErrors += cNewErrors;
}
```

```
   LDI R16, 0x01
   RCALL vCountErrors
```

```
vCountErrors:
   LDI  R30,(cErrors)
   LDI  R31,(cErrors+1)
   LD   R17, Z
   ADD  R16, R17
   ST   Z, R17
   RET
```

`cErrors = 0`

Task1: Running

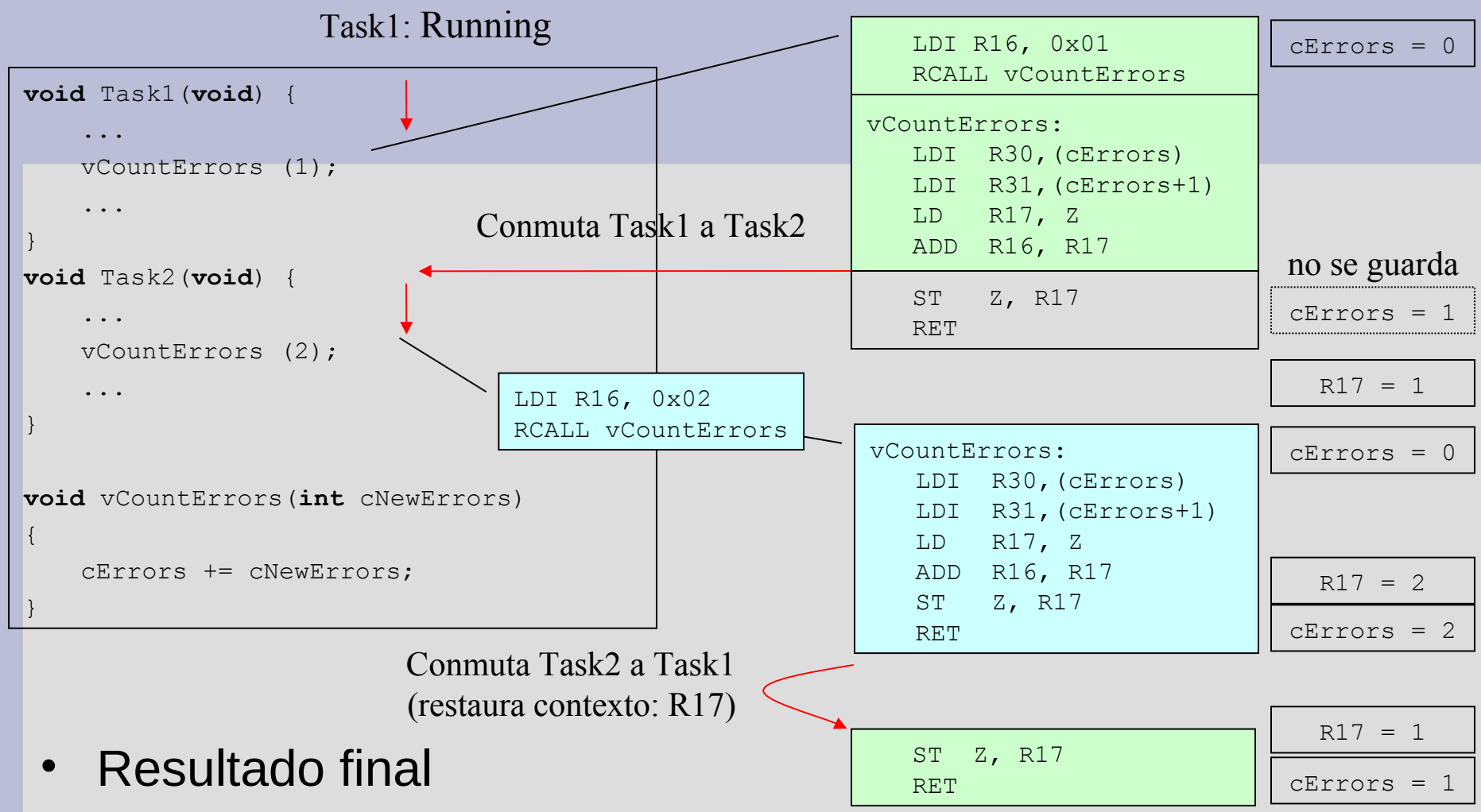
```
void Task1(void) {
    ...
    vCountErrors (1);
    ...
}
void Task2(void) {
    ...
    vCountErrors (2);
    ...
}

void vCountErrors(int cNewErrors)
{
    cErrors += cNewErrors;
}
```

`cErrors = 1`

Conmuta Task1 a Task2

- Resultado final
  - Debería ser: `cErrors = 3`

`cErrors = 3`



- Resultado final
  - Debería ser: `cErrors = 3`
  - Sin embargo da: `cErrors = 1`

Funciones reentrantes

- Pueden ser llamadas por más de una tarea y siempre funcionarán correctamente
  - aunque ocurra: conmutación de tareas o interrupciones
- Una función es *reentrante* si:
  1) Usa variables estáticas -compartidas- en forma atómica (variables automáticas OK)
  2) Llama solamente a funciones reentrantes.
  3) Usa el *hardware* de forma atómica.

Funciones reentrantes

```
int public_int;
int initialized = 4;
char *string = “Where does this string go?”;
void *vPointer;

void function (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   ...
}
```

Suponer que desde ésta función se acceden a todas las variables listadas:
¿Es la función reentrante?

- Recordar: donde son almacenadas las variables

Funciones reentrantes

```
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   ...
}
```

Problema:
variables estáticas

Funciones reentrantes

```
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   ...
}
```

Problema: variables estáticas

Sin Problema: variables automát.

Funciones reentrantes

```
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
    static int static_local;
    int local;
    ...
}
```

Problema: variables globales

Posible Problema: copia local puntero

Sin Problema: variables locales

Funciones reentrantes

```
static int num_errores = 0;

void ImprimeErrores() {
   if(num_errores > 10){
           num_errores -= 10;
           printf("Se han producido otros 10 errores más\n");
   }
}
```

¿Es ésta función reentrante?

- Fundamente...

Hardware compartido

- Problemas por código no entrante:
  - Impresión mezclada entre tareas... (Impresora)
  - Salida enmarañada en pantalla... (Monitor)
  - Paquetes mezclados en un enlace inalámbrico.
- Requerimiento para código reentrante:
  - Una vez que una tarea comienza una “transacción” de hardware debe completarla antes de liberar el hardware.

Resumen

- Entonces, ¿no se pueden compartir datos?
  - Sí se pueden, si se acceden de forma atómica.
- Atomicidad no ocurre “sola”
  - En algún microcontrolador tal vez...
    - Lenguaje C: `cErrors++;`
    - MSP430: `inc b, &cErrors`
    - ATmega: `LDI ...` (múltiples instrucciones)
  - Primero identificar secciones críticas y luego...
  - Hacerla atómica deshabilitando/habilitando interrupciones (u otros métodos)....

Bibliografía

- “An Embedded Software Primer”, David E. Simon
  - Chapter 6: Introduction to Real-Time Operating Systems