Tema 6:

Modelos de programación de software empotrado

Programación de sistemas y dispositivos

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Contenidos

- ✓ Sistemas guiados por tiempo vs. por eventos.
- ✓ Arquitectura super-loop.
- ✓ Arquitectura foreground/background.
- ✓ Arquitectura cola de funciones.
- ✓ Arquitectura RTOS.
- ✓ Programación con interrupciones.
- ✓ Secciones críticas y código reentrante.
- ✓ Mecanismos de comunicación.
- ✓ Aritmética en punto fijo.
- ✓ Funciones periódicas.
- ✓ Sistemas cyclic executive.
- ✓ Planificador cooperativo.
- ✓ Planificador híbrido.
- ✓ Sistemas multiestado
- ✓ Recuperación de bloqueos.
- ✓ Sistemas muestreados.
- ✓ Sistemas de control.

Tipos de sistemas

guiados por eventos vs. tiempo

- Sistema guiado por eventos:
  - Existe una colección de funciones asociadas a un conjunto de eventos externos.
  - La ocurrencia asíncrona de eventos determina el secuenciamiento de las funciones.
    - En caso de eventos simultáneos las funciones se ejecutan según prioridad.
  - El sistema detecta la ocurrencia de los eventos por pooling o por interrupción.

- Sistema guiado por tiempo:
  - Existe una colección de funciones asociadas a un conjunto de instantes de tiempo.
  - El transcurso síncrono del tiempo determina el secuenciamiento de las funciones.
    - En caso de simultaneidad las funciones se ejecutan según su prioridad.
  - El sistema mide el paso del tiempo usando un temporizador que interrumpe con periodicidad conocida (tick).

- Ambos sistemas están, en cierto modo, relacionados:
  - Un sistema guiado por tiempo puede emular a otro disparado por eventos.
    - Haciendo un pooling periódico (a la suficiente frecuencia) de la ocurrencia de eventos.
  - Un sistema guiado por tiempo es un caso particular de uno guiado por eventos:
    - Sólo tiene que responder a un único evento: el tick del temporizador

Arquitecturas básicas

super-loop (i)

- Un único bucle infinito chequea en orden cíclico el estado de los dispositivos de E/S y los sirve cuando es necesario.
  - Sin interrupciones, toda la E/S por pooling.
  - Una única hebra en ejecución: no existen datos globales compartidos.
  - Es simple.
- Problemas:
  - Las latencias/tiempos de respuesta* son poco predecibles.
    - En el peor caso, es el tiempo que tarda el bucle en realizar una iteración completa y en ejecutar todas la funciones asociadas a cada uno de los dispositivos.
  - La arquitectura es frágil.
    - Cualquier cambio en el código altera los tiempos de respuesta.
    - El comportamiento del sistema es muy dependiente de la estructura del código y de la secuencia de eventos.

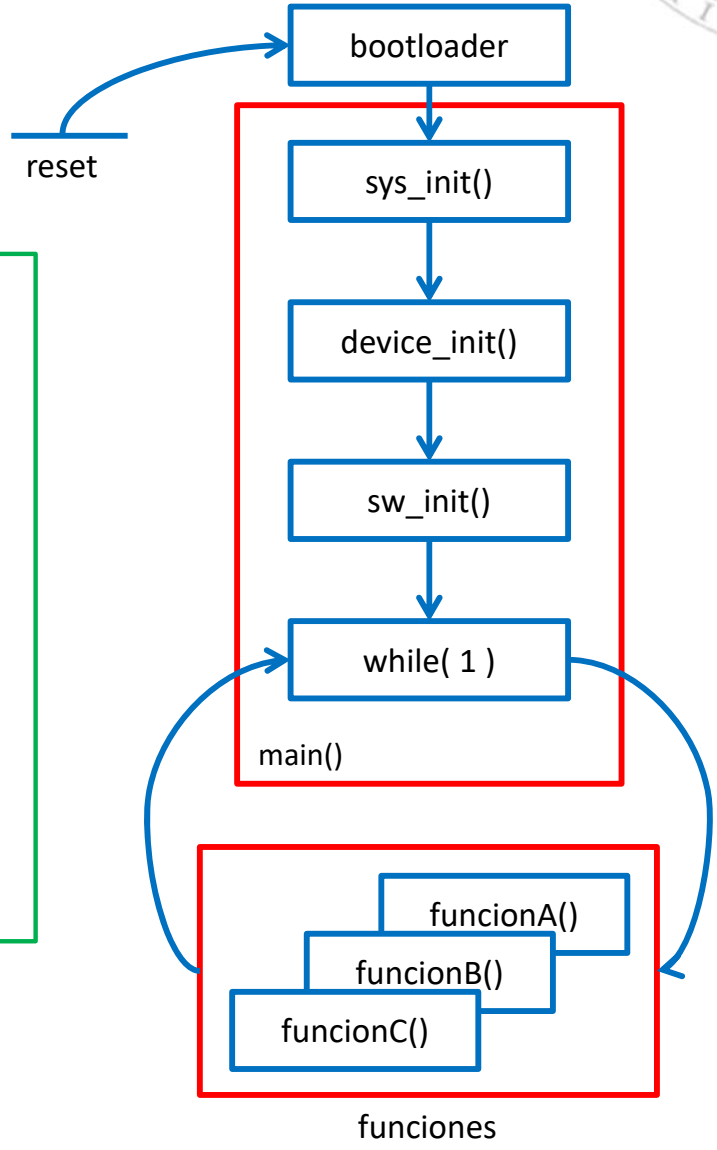
(*) Latencia: Tiempo que transcurre entre el momento en que ocurre un evento y el momento en que inicia la ejecución de la función asociada.

Tiempo de respuesta: Tiempo que transcurre entre el momento en que ocurre un evento y el momento en que finaliza la ejecución de la función asociada.

Arquitecturas básicas

super-loop (ii)

```
void main( void )
{
  while( 1 )
  {
    if( ...dispositivo A necesita ser servido... )
    {
       ...atiende al dispositivo A...
       funcionA();
    }
    if( ...dispositivo B necesita ser servido... )
    {
      ...atiende al dispositivo B...
      funcionB();
    }
    ...
  }
}
```



Arquitecturas básicas

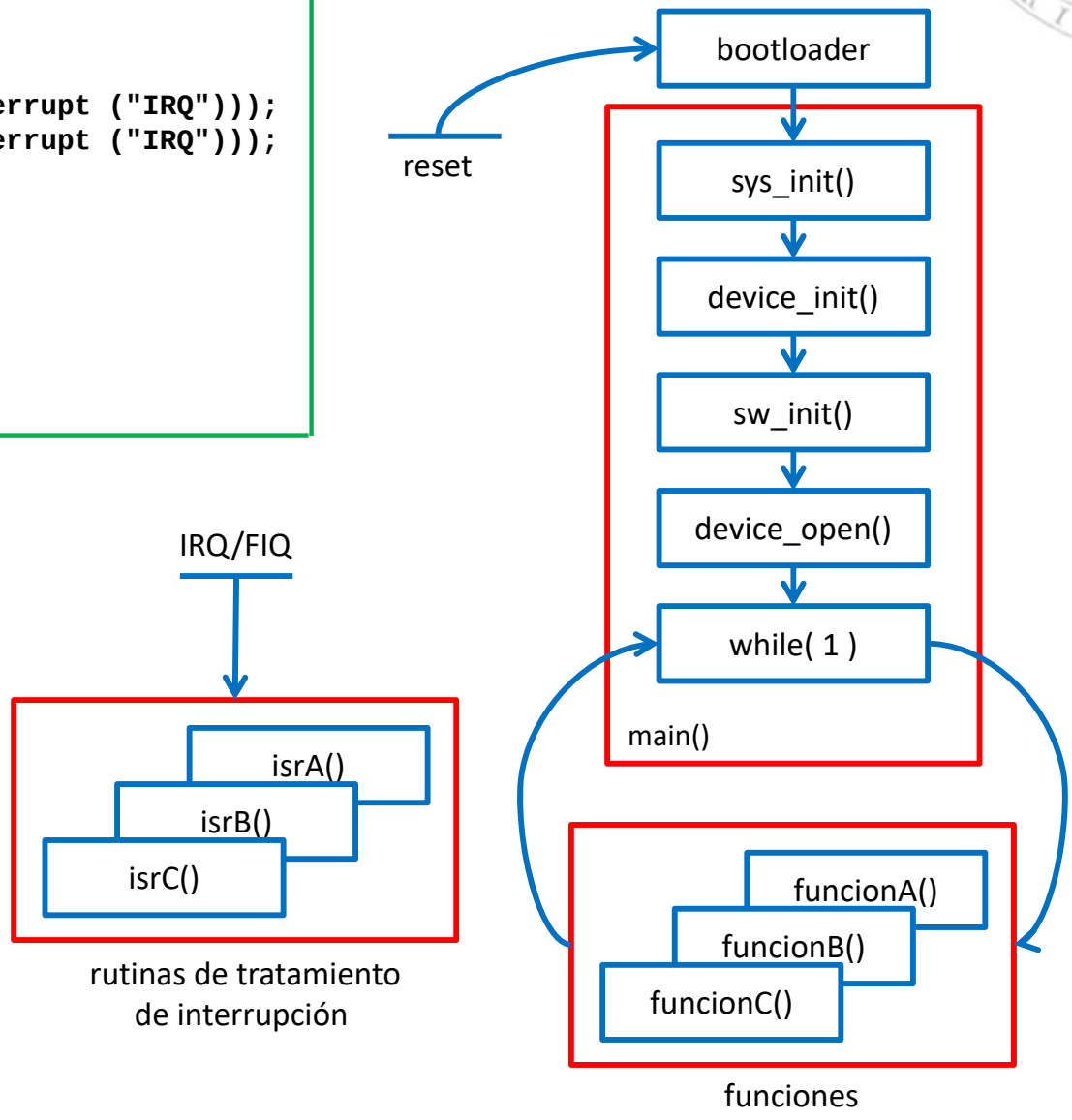
foreground/background (i)

- Super-loop con interrupciones
  - Varias hebras en foreground (RTI) atienden a los dispositivos de E/S según su prioridad cuando estos lo solicitan por interrupción.
    - Estas hebras, típicamente, solo señalizan eventos y/o escriben/leen datos en FIFOs.
    - Si las RTI son cortas no se permite el anidamiento de interrupciones.
  - Una hebra en background (main) realiza un bucle infinito de mínima prioridad.
    - Típicamente realiza todas las funciones no críticas en tiempo y todo el procesamiento de datos encolados.
  - Las hebras para comunicarse entre sí utilizan variables globales compartidas.
    - Es necesario detectar y resolver el acceso a secciones críticas.
- Problemas:
  - Si el procesamiento se realiza en background:
    - Las funciones asociadas a cualquier dispositivo tienen la misma prioridad.
    - En el peor caso, los procesamientos se realizan con retrasos similares al modelo super-loop.
  - Si el procesamiento se realiza en foreground:
    - Las RTI se alargan, y aunque las funciones se realizan según su prioridad, penalizan el tiempo de respuesta de interrupciones menos prioritarias.

Arquitecturas básicas

foreground/background (ii)

```
volatile boolean flagA = 0;
volatile boolean flagB = 0;
...
void isrA( void ) __attributte__ ((interrupt ("IRQ")));
void isrB( void ) __attributte__ ((interrupt ("IRQ")));
...
void main( void )
{
  while( 1 )
  {
    if( flagA )
    {
       flagA = 0;
       funcionA();
    }
    if( flagB )
    {
       flagB = 0;
       funcionB();
    }
    ...
}
void isrA( void )
{
  ...atiende al dispositivo A...
  flagA = 1;
}
void isrB( void )
{
  ...atiende al dispositivo B...
  flagB = 1;
}
...
```



Arquitecturas básicas

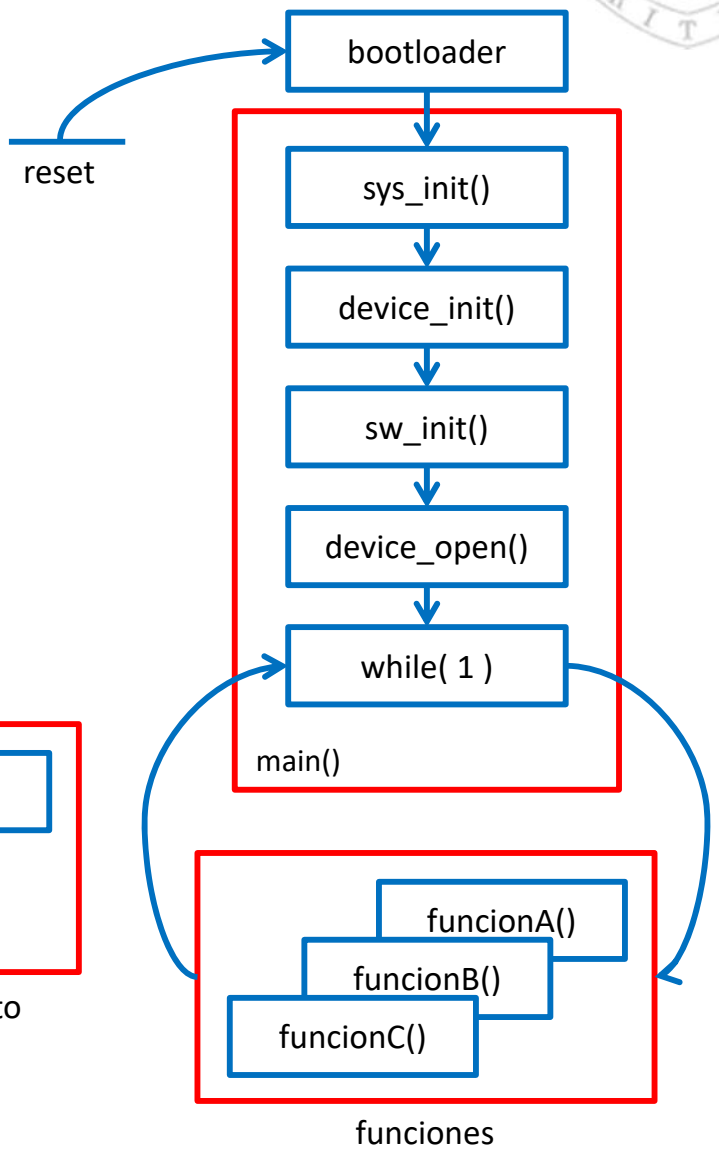
cola de funciones (i)

- Foreground/background con gestión de prioridades
  - Las hebras en foreground (RTI) atienden las interrupciones de los dispositivos de E/S según su prioridad, encolando las funciones encargadas del procesamiento.
    - Adicionalmente, pueden señalizar eventos y/o escribir/leer datos en FIFOs.
    - Si las RTI son cortas no se permite el anidamiento de interrupciones.
  - La hebra en background (main) realiza un bucle infinito ejecutando en orden las funciones encoladas.
    - El orden puede ser el de llegada o según el esquema de prioridad que se desee implantar.
  - Las hebras para comunicarse entre sí utilizan variables globales compartidas.
    - Es necesario detectar y resolver el acceso a secciones críticas.
- Problemas:
  - Los procesamientos largos (peor cuando son poco prioritarios) pueden retrasar la ejecución de funciones más prioritarias.

Arquitecturas básicas

cola de funciones (ii)

```
volatile queue_t queue;
...
void isrA( void ) __attributte__ ((interrupt ("IRQ")));
void isrB( void ) __attributte__ ((interrupt ("IRQ")));
...
void main( void )
{
  while( 1 )
  {
    if( !queue_empty() )
    {
      pf = dequeue();
      (*pf)();
    }
  }
}
void isrA( void )
{
  ...atiende al dispositivo A...
  if( !queue_full() )
    enqueue( funcionA );
}
void isrB( void )
{
  ...atiende al dispositivo B...
  if( !queue_full() )
    enqueue( funcionB );
}
...
```



Arquitecturas básicas

RTOS (i)

- Sistema operativo con planificación basada en prioridades
  - Varias hebras en foreground (RTI) atienden a los dispositivos de E/S según su prioridad cuando estos lo solicitan por interrupción.
    - Señalizan eventos y comunican datos usando primitivas del RTOS (semáforos, FIFOs, etc.)
  - Varias hebras en background (tareas) realizan indefinidamente diversas funciones.
    - El programador asigna prioridades a cada tarea.
    - La hebra en background principal (main) tras inicializar el sistema desaparece.
  - Las hebras para comunicarse entre sí utilizan variables globales compartidas.
    - Es necesario detectar y resolver el acceso a secciones críticas.
  - El kernel de RTOS:
    - Gestiona la comunicación y sincronización entre las hebras (RTI y tareas).
    - Decide, típicamente por prioridades, qué tarea (i.e. qué función) debe ejecutarse en cada momento.
    - Conmuta entre tareas (conmutando entre contextos según el caso).
    - Según el caso, gestiona las copias de los contextos (registros y pila) de cada tarea.
    - Puede o no expropiar una tarea y pasar a ejecutar otra más prioritaria.
- Desventajas:
  - El RTOS supone una sobrecarga al sistema.

J.M. Mendías
2022

Arquitecturas básicas

RTOS (ii)

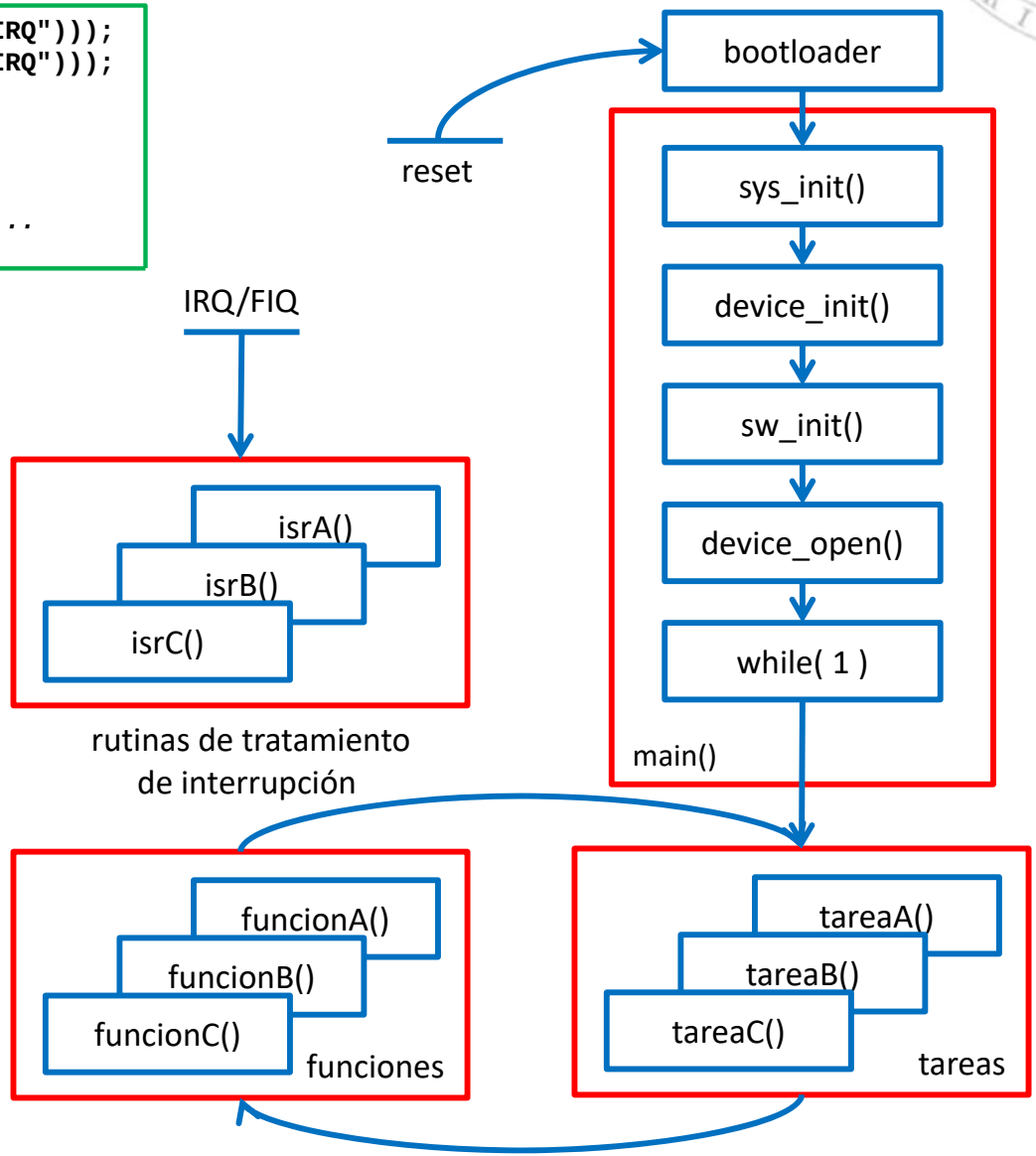
- RTOS cooperativo (non-preemptive multitasking)
  - Las tareas ceden voluntariamente el control al kernel para que conmute de tarea.
    - Implícitamente cuando terminan o explícitamente mediante una llamada al sistema.
    - El programador debe asegurar que las tareas cedan regularmente el control al kernel (i.e. en todo bucle de espera activa, en porciones de código largas, etc).
  - Cuando una tarea cede el control:
    - El planificador pasa a ejecutar la tarea preparada de mayor prioridad.
    - Si no existe dicha tarea, continúa la ejecución de la que cedió el control.
  - Si ocurre un evento:
    - La RTI lee/escribe el dato del dispositivo y pide al kernel que lo encole/desencole.
    - El kernel pasa a preparada la tarea que estaba a la espera del dato.
    - Dicha tarea no se ejecutará hasta que la tarea actualmente en ejecución ceda el control y no existan otras tareas preparadas con mayor prioridad.
- RTOS expropiativo (preemptive multitasking)
  - Las excepciones disparan la conmutación de tareas
    - Estos cambios son transparentes al programador.
  - Si ocurre un evento:
    - La tarea en ejecución es interrumpida y el kernel guarda su contexto.
    - La RTI lee/escribe el dato del dispositivo y pide al kernel que lo encole/desencole.
    - El kernel pasa a preparada la tarea que estaba a la espera del dato.
    - El planificador conmuta a la tarea preparada más prioritaria restaurando su contexto.

J.M. Mendías
2022

Arquitecturas básicas

RTOS (iv)

```
void ISRA( void ) __attributte__ ((interrupt ("IRQ")));
void ISRB( void ) __attributte__ ((interrupt ("IRQ")));
...
void main( void )
{
  ...crea tareas...
  ...inicia RTOS (de esta función no se vuelve)...
}
void ISRA( void )
{
  ...atiende al dispositivo A...
  signal( A );
}
void ISRB( void )
{
  ...atiende al dispositivo B...
  signal( B );
}
...
void tareaA( void )
{
  while( 1 )
  {
    wait( A );
    funcionA();
  }
}
void tareaB( void )
{
  while( 1 )
  {
    wait( B );
    funcionB();
  }
}
```



Programación con interrupciones

el problema de los datos compartidos (i)

- Las hebras se comunican entre sí usando variables globales compartidas
  - El acceso a estas variables es problemático si una hebra (i.e. RTI) expropia a otra.

INCORRECTO

```
volatile uint16 temp[2];
...
void leerTemperatura_isr( void )
{
  ...borra flag de int. pendiente...
  temp[0] = ...leer del HW...
  temp[1] = ...leer del HW...
}

void main( void )
{
  uint16 t0, t1;
  while( 1 )
  {
    t0 = temp[0];
    t1 = temp[1];
    if( t0 != t1 )
      ...suena la alarma...
  }
}
```

La RTI lee la temperatura medida por 2 sensores. Es llamada periódicamente (i.e. cada pocos ms, o cada vez que cambia la lectura de un sensor, etc.)

El programa hace sonar una alarma cuando detecta que las temperaturas medidas son distintas

***Interrupción**: La RTI lee 2 valores iguales (i.e 7 y 7)*

***Interrupción**: La RTI lee 2 valores iguales pero distintos a la última lectura (i.e 8 y 8)*

***Suena la alarma**, aun siendo ambas temperaturas iguales en todo momento (pero temp0 = 7 y temp1 = 8)*

Programación con interrupciones

el problema de los datos compartidos (ii)

- El problema siempre aparece en alto nivel aunque se utilicen en el código de la aplicación directamente las variables globales
  - Ya que una sentencia en alto nivel implica múltiples instrucciones en ensamblador

INCORRECTO

```
volatile uint16 temp[2];
...
void leerTemperatura_isr( void )
{
  ...borra flag de int. pendiente...
  temp[0] = ...leer del HW...
  temp[1] = ...leer del HW...
}

void main( void )
{
  while( 1 )
  {
    if( temp[0] != temp[1] )
       ...suena la alarma...
  }

}
```

```
  ...
bucle_while:
  ldr r0, =temp[0]      <-- Interrupción
  ldr r0, [r0]
  ldr r1, =temp[1]      <-- Interrupción
  ldr r1, [r1]
  cmp r0, r1
  beq bucle_while
suena_alarma:           <-- Suena la alarma
  ...
```

Programación con interrupciones

el problema de los datos compartidos (iv)

- Es un bug muy difícil de depurar porque se manifiesta ocasionalmente
  - Existe siempre que hay acceso concurrente a variables globales compartidas.

INCORRECTO

```
volatile uint8 seg, min, hor;
void actualiza_tiempo( void ) __attributte__ ((interrupt ("IRQ")));
...
void actualiza_tiempo( void )
{
  if( ++seg >= 60 )
  {
    seg = 0;
    if( ++min >= 60 )
    {
      min = 0;
      if( ++hor >= 24 )
        hor = 0;
    }
  }
}

int segundos_transcurridos( void )
{
  return ((((hor*60)+min)*60)+seg);
}
```

La RTI es llamada periódicamente cada segundo y actualiza la hora del sistema.

Esta función calcula los segundos transcurridos desde las 00:00:00

Si se interrumpe durante el cálculo el error puede llegar a ser hasta de 1 h (i.e. la hora pasa de 03:59:59 a 04:00:00).

Programación con interrupciones

deshabilitación temporal de interrupciones (i)

- La solución pasa por hacer un acceso atómico a las datos compartidos
  - Deshabilitando temporalmente las interrupciones cuando se accede a los datos

CORRECTO

```
volatile uint16 temp[2];
...
void leerTemperatura_isr( void )
{
  ...borra flag de int. pendiente...
  temp[0] = ...leer del HW...
  temp[1] = ...leer del HW...
}

void main( void )
{
  uint16 t0, t1;
  while( 1 )
  {
     INT_DISABLE;
     t0 = temp[0];
     t1 = temp[1];
     INT_ENABLE;
     if( t0 != t1 )
       ...suena la alarma...
  }
}
```

CPSR.FIQ = CPSR.IRQ = 1

CPSR.FIQ = CPSR.IRQ = 0

Aunque se produzcan interrupciones no se servirán y, por tanto, las variables globales conservarán su valor durante este intervalo .

Una vez habilitadas, las interrupciones pendientes se servirán y las variables globales podrán cambiar pero no se alterará el valor de las copias locales hasta la siguiente lectura.

Programación con interrupciones

deshabilitación temporal de interrupciones (ii)

- La deshabilitación temporal de interrupciones tiene sus riesgos:
  - Hay que asegurar que tras toda deshabilitación sigue una habilitación.
  - La implementación de las primitivas debe permitir la anidación arbitraria de pares habilitación/deshabilitación.
  - El tiempo que permanecen las interrupciones deshabilitadas debe ser mínimo.
  - Además, debe programarse en ensamblador por requerir el acceso al CPSR.

INCORRECTO

```
int segundos_transcurridos( void )
{
  INT_DISABLE;
  return ((((hor*60)+min)*60)+seg);
  INT_ENABLE;
}
```

Nunca se ejecuta (retorna antes): luego deja de actualizarse el reloj

CORRECTO

```
int segundos_transcurridos( void )
{
  int aux;
  INT_DISABLE;
  aux =((((hor*60)+min)*60)+seg);
  INT_ENABLE;
  return aux;
}
```

Programación con interrupciones

deshabilitación temporal de interrupciones (iii)

INCORRECTO

```
#define INT_DISABLE asm volatile (          \
  "mrs r0, cpsr              \n"            \
  "orr r0, r0, #0b11000000   \n"            \
  "msr cpsr_c, r0              " : : : "r0" )

#define INT_ENABLE asm volatile (           \
  "mrs r0, cpsr              \n"            \
  "and r0, r0, #0b00111111   \n"            \
  "msr cpsr_c, r0              " : : : "r0" )
...
void main( void )
{
  uint16 t0, t1, aux;
  while( 1 )
  {
     INT_DISABLE;
     aux = segundos_transcurridos();
     t0 = temp[0];
     t1 = temp[1];
     INT_ENABLE;
     if( t0 != t1 )
       ...suena la alarma...
  }
}
```

Esta implementación habilita incondicionalmente las interrupciones.

Deshabilita interrupciones

- *Deshabilita (sin efecto) las interrupciones*
- *Procesa*
- *Habilita las interrupciones*

*Se ejecuta con las interrupciones habilitadas: **error***

No tiene efecto

Programación con interrupciones

deshabilitación temporal de interrupciones (iv)

CORRECTO

```
#define INT_DISABLE asm volatile (                \
  "mrs   r0, cpsr                \n"              \
  "stmfd sp!, {r0}               \n"              \
  "orr   r0, r0, #0b11000000     \n"              \
  "msr   cpsr_c, r0                " : : : "r0" )

#define INT_ENABLE asm volatile (                 \
  "ldmfd sp!, {r0}               \n"              \
  "msr   cpsr_c, r0                " : : : "r0" )
...
void main( void )
{
  uint16 t0, t1, aux;
  while( 1 )
  {
     INT_DISABLE;
     aux = segundos_transcurridos();
     t0 = temp[0];
     t1 = temp[1];
     INT_ENABLE;
     if( t0 != t1 )
       ...suena la alarma...
  }
}
```

apila CPSR

deshabilita IRQ y FIQ

restaura CPSR al valor anterior a la llamada INT_ENABLE

Deshabilita interrupciones

- *Deshabilita (sin efecto) las interrupciones*
- *Procesa*
- *Restaura el CPSR: deja deshabilitadas las int.*

Se ejecuta con las interrupciones dehabilitadas

Habilita interrupciones

Programación con interrupciones

deshabilitación temporal de interrupciones (v)

- En el caso de la arquitectura ARM7 existe un problema adicional
  - Cuando la interrupción llega durante la modificación de cpsr

```
...
orr   r0, r0, #0b11000000
msr   cpsr_c, r0
```

  - En ese caso, tras modificar el CPSR (i.e interrupciones deshabilitadas) se saltará a la RTI copiando el CPSR sobre el SPSR_irq
  - Si la RTI a su vez modifica el SPSR_irq (activando los flags de interrupción), al volver de la RTI (y restaurarse el CPSR desde el SPSR_irq) las interrupciones se habilitan
  - Solución: evitar modificar el SPSR durante las RTI, o modificar la macro para que compruebe que el cambio de CPSR se ha hecho efectivo.

```
#define INT_DISABLE asm volatile (                \
  "mrs   r0, cpsr                \n"              \
  "stmfd sp!, {r0}               \n"              \
  "orr   r0, r0, #0b11000000     \n"              \
  "msr   cpsr_c, r0              \n"              \
  "mrs   r0, cpsr                \n"              \
  "and   r0, r0, #0b11000000     \n"              \
  "cmp   r0, #0b11000000         \n"              \
  "addne sp, sp, #4              \n"              \
  "subne pc, pc, #44                 " : : : "r0" )
```

salta 8 instrucciones hacia atrás teniendo en cuenta el efecto del pipeline del ARTM7TDI

Programación con interrupciones

spin locks (i)

- Desactivar temporalmente las interrupciones es efectivo, pero:
  - Penaliza la latencia de todas las interrupciones actualicen o no las variables compartidas que protegen.
  - Para evitar esto, se utilizan spin-locks (semáforos binarios con espera activa) .
- Un spin-lock tiene 2 primitivas: lock y unlock.
  - Lock, usando una única instrucción máquina (tipo swap, test and set, etc.) primero lee el semáforo y después lo cierra (pone a 1).
    - Si un chequeo posterior detecta que estaba cerrado vuelve a intentarlo hasta conseguirlo.
    - Una única instrucción máquina es atómica (no puede ser interrumpida)

```
#define LOCK_MUTEX( mutex_p ) asm volatile (          \
  "mov   r0, #1          \n"                          \
  "swpb  r0, r0, [%0]    \n"                          \
  "cmp   r0, #1          \n"                          \
  "subeq pc, pc, #20       " : : "r" (mutex_p) : "r0" )
```

salta 2 instrucciones hacia atrás teniendo en cuenta el efecto del pipeline del ARTM7TDI

  - Unlock, abre el semáforo (pone a 0)

```
#define UNLOCK_MUTEX( mutex_p ) asm volatile (        \
  "mov   r0, #0          \n"                          \
  "strb  r0, [%0]          " : : "r" (mutex_p) : "r0" )
```

Programación con interrupciones

spin locks (ii)

```
volatile boolean mutex = 0;
volatile uint16 temp[2];
...
void leerTemperatura_isr( void )
{
  ...borra flag de int. pendiente...
  if( !mutex )
  {
    temp[0] = ...leer del HW...
    temp[1] = ...leer del HW...
  }
}

void main( void )
{
  int t0, t1;
  while( 1 )
  {
     LOCK_MUTEX( &mutex );
     t0 = temp[0];
     t1 = temp[1];
     UNLOCK_MUTEX( &mutex );
     if( t0 != t1 )
       ...suena la alarma...
  }
}
```

*nunca **while** en una ISR: bloquearía el sistema*

no es necesario usar LOCK_MUTEX: todo el código de la RTI es atómico si no se permite la anidación de interrupciones

Si main está leyendo la temperatura la RTI no la actualiza y se pierde una lectura

espera hasta que pueda bloquear el semáforo

Podrá interrumpirse, pero la RTI no modificará las variables mientras se esté ejecutando esta sección crítica

desbloquea el semáforo

Programación con interrupciones

el problema de los datos compartidos (v)

- No olvidar declarar como volatile todos los datos compartidos
  - para indicar al compilador que acceda a memoria cada vez que se referencien.

```
void tick_isr( void ) __attributte__ ((interrupt ("IRQ")));
...
void tick_isr( void )
{
  ticks++;
}

void main( void )
{
  int bar
  ...
  bar += ticks;
  ...
  while( ticks != 100 );
  ...
}
```

```
static int ticks;
```

```
  ...
  ldr r0, =ticks
  ldr r0, [r0]
  ldr r1, =bar
  ldr r1, [r1]
  add r1, r1, r0
  ...
bucle_while:
  cmp r0, #100
  bne bucle_while
  ...
```

```
static volatile int ticks;
```

```
  ...
  ldr r0, =ticks
  ldr r0, [r0]
  ldr r1, =bar
  ldr r1, [r1]
  add r1, r1, r0
  ...
bucle_while:
  ldr r0, =ticks
  ldr r0, [r0]
  cmp r0, #100
  bne bucle_while
  ...
```

como ticks no se modifica en main, sin volatile puede que el compilador optimice los accesos a memoria y nunca salga del bucle...

Programación con interrupciones

anidamiento (i)

- En el ARM7TDMI, cuando un periférico interrumpe...
  - por la línea FIQ, el HW enmascara las interrupciones FIQ e IRQ.
  - por la línea IRQ, el HW enmascara las interrupciones IRQ.
  - si la RTI no deshace el enmascaramiento (modificando explícitamente los bits F/I del CPSR) la línea FIQ/IRQ permanece deshabilitada durante toda su ejecución.
- Por ello, por defecto:
  - La RTI de una FIQ no puede interrumpirse (no hay anidación).
  - La RTI de una IRQ solo puede ser interrumpida por una FIQ (un nivel de anidación).
    - Puede generar un problema adicional de datos compartidos entre RTI-FIQ y RTI-IRQ.
- Si dos periféricos interrumpen a la vez, se sirven secuencialmente según la prioridad establecida por HW /SW:
  - Si uno usa la línea FIQ y otro la IRQ: se sirve primero al que interrumpa por FIQ
  - Si ambos usan la línea FIQ: será la RTI (común) la que dirima el orden.
  - Si ambos usan la línea IRQ y el controlador tiene deshabilitada la vectorización: será la RTI (común) la que dirima el orden.
  - Si ambos usan la línea IRQ y el controlador tiene habilitada la vectorización: será la prioridad programada en el controlador la que dirima el orden.

J.M. Mendías
2022

Programación con interrupciones

anidamiento (ii)

- Un servicio secuencial por orden de llegada de interrupciones:
  - No supone un problema si las RTIs son muy pequeñas (buena práctica).
  - Pero si son grandes, puede ser necesario permitir el anidamiento para reducir la latencia de interrupciones de alta prioridad
    - Aunque un anidamiento descontrolado puede requerir un tamaño de pila impredecible.

- Para permitir el anidamiento, el programador debe habilitar la línea IRQ/FIQ dentro del cuerpo de la RTI, pero antes debe:
  - Salvar LR_irq y SPSR_irq (son sobrescritos cada vez que se produce una interrupción)
    - LR_irq lo salva el prólogo de la rutina de tratamiento
    - pero SPSR_irq lo debe salvar el programador explícitamente
  - Enmascarar interrupciones de igual o menor prioridad (evita reentradas cíclicas)
- Adicionalmente, si la RTI llama a otra función, el programador debe previamente:
  - Cambiar a un modo de ejecución distinto de IRQ/FIQ (típicamente SYS)
  - Salvar el LR del modo al que ha cambiado (típicamente LR_sys)
    - En caso contrario, si se produjera una interrupción en el intervalo que va desde el salto a la función y el apilado de LR_irq, la dirección de retorno (desde la función invocada) sería sobrescrita con la dirección de retorno de la nueva RTI (que usa también LR_irq).

Programación con interrupciones

ejemplo de anidamiento de IRQ (i)

- Una RTI con posibilitad de anidamiento en el ARM7TDMI supone:

... *prólogo*

11. Borrado del flag de interrupción pendiente (I_ISPC/F_ISPC)
12. Almacenamiento del registro máscara
13. Enmascarado las fuentes de interrupción de menor o igual prioridad (INTMSK)
14. Almacenamiento en pila de SPSR_irq
15. Conmutación a modo de ejecución SYS
16. Habilitación de interrupciones
17. Almacenamiento en pila de LR_sys

macro habilitación

18. Procesamiento
19. Restauración de LR_sys
20. Deshabilitación de interrupciones
21. Restauración del estado del procesador SPSR_irq

macro deshabilitación

22. Restauración del registro de máscara

... *epílogo*

rutina de servicio

Programación con interrupciones

ejemplo de anidamiento de IRQ (ii)

- El acceso a SPSR y CPSR necesariamente debe hacerse en ensamblador.

```
#define IRQ_NESTING_ENABLE asm volatile (  \
  "mrs   lr, spsr              \n“    \      } apila SPSR_irq usando LR_irq como reg. auxiliar
  "stmfd sp!, {lr}             \n"    \      } (previamente apilado por el prólogo de la RTI)
  "msr   cpsr_c, #0b00011111   \n"    \      ···· habilita IRQ y cambia a modo SYS
  "stmfd sp!, {lr}"                   )  ····· apila el LR_sys

#define IRQ_NESTING_DISABLE asm volatile ( \
  "ldmfd sp!, {lr}             \n"    \   ····· restaura el LR  sys
  "msr   cpsr_c, #0b10010010   \n"    \      ····· deshabilita IRQ y retorna a modo IRQ
  "ldmfd sp!, {lr}             \n"    \
  "msr   spsr, lr                "    )  } restaura SPSR_irq usando LR_irq como reg. aux.

void foo( void ) __attributte__ ((interrupt ("IRQ")))
{
  ...
  I_ISPC = ...;  ····· borra el flag de interrupción pendiente que corresponda.
  bar = INTMSK;  ····· almacena la máscara actual de interrupciones.
  INTMSK |= ...; ····· enmascara las interrupciones que correspondan.
  IRQ_NESTING_ENABLE;
  ...
  IRQ_NESTING_DISABLE;
  INTMSK = bar;  ····· restaura la máscara de interrupciones inicial.
}
```

Programación con interrupciones

latencia y tiempo de respuesta

- Las interrupciones son el método preferido para la respuesta rápida a eventos.

- La latencia de una interrupción depende de:
  1. El tiempo máximo en que están deshabilitadas.
  2. El tiempo que tardan en ejecutarse cualquier RTI de mayor prioridad (si la anidación está permitida) .
  3. El tiempo que tarda el procesador en detectar la interrupción y saltar a la RTI.
  4. El tiempo que la RTI tarda en guardar el contexto y realizar la acción que se considera respuesta a la interrupción.

- Conclusiones:
  - El código de la RTI debe ser eficiente para reducir el 4.
  - El código de la RTI debe ser pequeño para reducir el 2.
  - Las secciones críticas deben ser cortas para reducir 1.

Aplicaciones multihebra

secciones críticas y código reentrante

- La garantía de corrección de una aplicación multihebra es:
  - Toda porción de código que acceda a una sección crítica sea atómico.
  - Todo su código sea reentrante.
- Sección crítica: porción de código en la cual se accede a un recurso compartido (variable global o dispositivo de E/S) entre 2 o más hebras.
  - Para asegurar la coherencia de datos, el acceso debe ser mutuamente exclusivo.
  - Cuando una hebra está ejecutando una sección crítica no debe ser expropiada por otra que acceda al mismo recurso.
- Código atómico: aquel cuya ejecución no puede ser interrumpida por otro código que modifique los datos que está usando.
  - Puede ser interrumpido por otro que modifique otros datos.
  - Un código puede hacerse atómico deshabilitando interrupciones o arbitrando un mecanismo de acceso mutuamente exclusivo a sus datos compartidos.
- Código reentrante: aquel que puede ser ejecutado concurrentemente por más de una hebra.
  - Debe almacenar las variables locales en pila.
    - Las funciones en C son reentrantes (el paso de parámetros y las variables locales se ubican en pila)
    - Las funciones en ensamblador, dependerá de cómo se programen.
  - Debe acceder atómicamente a las variables globales.

Aplicaciones multihebra

secciones críticas: categorías

- Las secciones críticas pueden clasificarse en 3 categorías:
  - Read-Modify-Write:
    - Se copia localmente (variable, registro del procesador) el valor de una variable global.
    - Se modifica localmente dicho valor (la global todavía no ha sido modificada).
    - Se actualiza la variable global con el valor local modificado.
  - Write followed by Read:
    - Se escribe una variable global (siendo la única copia de una información importante).
    - Se lee dicha variable global, esperando que el dato original siga allí.
  - Nonatomic Multistep Write:
    - Se escribe parte del nuevo valor que debe tomar una variable global.
    - Seguidamente se escribe el resto del nuevo valor en la variable global.
- Los dispositivos de E/S, a estos efectos, pueden considerarse variables globales

```
int16 data;
...
void foo( void )
{
  data = data + 100;
}
```

```
int16 data;
...
int foo( int x, int y )
{
  data = x + y;
  return data
}
```

```
int16 data[2];
...
int foo( int x, int y )
{
  data[0] = x;
  data[1] = y;
}
```

J.M. Mendías
2022

Aplicaciones multihebra

secciones críticas: escenarios

- El acceso no atómico a una sección crítica por 2 o más hebras puede producirse en 3 escenarios:
  - Sistemas sin interrupciones anidadas (hebra vs. ISR):
    - Una hebra entra en una sección crítica.
    - Otra hebra la interrumpe y ejecuta al completo la sección crítica.
    - El control retorna a la primera y sale de la sección crítica.
  - Sistemas con interrupciones anidadas (ISR vs. ISR):
    - Una hebra entra en una sección crítica.
    - Otra hebra la interrumpe y entra en su sección crítica.
    - Una tercera hebra interrumpe y ejecuta al completo la sección crítica
    - Las hebras finalizan en orden inverso a su interrupción.
  - RTOS expropiativo (hebra vs. hebra):
    - Una hebra entra en una sección crítica
    - Otra hebra la interrumpe y entra en su sección crítica.
    - La hebra es expropiada y el control cedido a la primera.
    - La primera sale de la sección crítica.
    - El control retorna a la segunda y sale de la sección crítica.

Aplicaciones multihebra

secciones críticas: protección

- Arquitectura super-loop
  - No hay secciones críticas porque solo hay una hebra.
- Arquitectura foreground-background
  - La hebra background debe proteger todas sus secciones críticas.
    - Si lo hace mediante inhabilitación de interrupciones, las hebras foreground no tienen que hacer chequeos: solo entrarán en las sección crítica cuando la hebra background salga.
    - Si lo hace mediante spin-lock, la hebra foreground deberá evitar entrar en la sección crítica en caso de que este siendo accedida por la hebra background.
  - Si se permite anidamiento de interrupciones, todas las hebras expropiables deberán proteger sus secciones críticas.
- RTOS cooperativo
  - El programador tiene control sobre cuando se realizan los cambios de contexto.
  - Nunca deberá cambiar de contexto en mitad de una sección crítica.
- RTOS expropiativo
  - El programador no tiene control sobre cuando se realizan los cambios de contexto.
  - Todas las hebras protegen homogéneamente sus secciones críticas.

J.M. Mendías 2022

Aplicaciones multihebra

sincronización y comunicación: flags

- Un variable booleana compartida que permite sincronizar 2 hebras
  - Una hebra activa el flag.
  - La otra hebra chequea (y eventualmente espera) la activación del flag. Cuando encuentra el flag activado, lo desactiva.

```
volatile boolean flag = 0;
...
void main( void )
{
  while( 1 )
  {
    ...
    if( flag )
    {
      flag = 0;
      ... procesa ...
    }
  }
}
void isr( void )
{
  ...
  flag = 1;
}
```

```
while( 1 )
{
  ...
  while( !flag );
  flag = 0;
  ... procesa ...
}
```

MAIN chequea

ISR señaliza

```
volatile boolean flag = 0;
...
void main( void )
{
  while( 1 )
  {
    ...
    flag = 1;
  }
}
void isr( void )
{
  ...
  if( flag );
  {
    flag = 0;
    ...procesa...
  }
}
```

MAIN señaliza

ISR chequea

nunca while *en una ISR: bloquearía el sistema*

Aplicaciones multihebra

sincronización y comunicación: mailboxes

- Un flag con una variable de datos (mensaje) asociada
  - Una hebra escribe el dato y activa el flag.
  - La otra hebra chequea (y eventualmente espera) la activación del flag. Cuando encuentra el flag activado, lee el dato y desactiva el flag.

```
volatile boolean flag = 0;
volatile mail_t mail;
...
void main( void )
{
  while( 1 )
  {
    ...
    if( flag )
    {
      ... lee mail ...
      flag = 0;
    }
  }
}
void isr( void )
{
  ...
  mail = ...
  flag = 1;
}
```

MAIN chequea y lee

ISR escribe y señaliza

```
volatile boolean flag = 0;
volatile mail_t mail;
...
void main( void )
{
  while( 1 )
  {
    ...
    mail = ...
    flag = 1;
  }
}
void isr( void )
{
  ...
  if( flag );
  {
    ... lee mail ...
    flag = 0;
  }
}
```

MAIN escribe y señaliza

ISR chequea y lee

J.M. Mendías
2022

Aplicaciones multihebra

sincronización y comunicación: barreras de memoria (i)

- En ejecución, el orden de acceso a variables sin dependencia de datos no tiene por qué respetar el indicado en el código fuente.
  - Los procesadores modernos ejecutan las instrucciones fuera de orden.
  - Los compiladores modernos reorganizan los accesos a memoria.

```
volatile boolean flag = 0;
mail_t mail;
...
void main( void )
{
  while( 1 )
  {
    ...
    mail = ...
    flag = 1;
  }
}
void isr( void )
{
  ...
  if( flag );
  {
    ... lee mail ...
    flag = 0;
  }
}
```

Podría pensarse que no es necesario declararla como como volatile

Estas 2 variables no tienen una dependencia explícita: el compilador puede generar código que active el flag antes de escribir el mail.

La ISR leerá un valor inválido

INCORRECTO

J.M. Mendías
2022

Aplicaciones multihebra

sincronización y comunicación: barreras de memoria (ii)

- Una barrera de memoria fuerza que todos los accesos a variables previos a la barrera estén finalizados antes de comenzar los accesos a las variables después de la barrera.
  - En procesadores con ejecución fuera de orden se realiza mediante una instrucción.
  - En compiladores optimizantes se realiza mediante una directiva.
- El core ARM7TDMI ejecuta en orden, la directiva de GCC es:

```
#define MEMBAR asm volatile ( "" : : : "memory" )
```

```
volatile boolean flag;
mail_t mail;
...
void main( void )
{
  while( 1 )
  {
    ...
    mail = ...
    flag = 1;
  }
}
```

INCORRECTO

```
volatile boolean flag;
volatile mail_t mail;
...
void main( void )
{
  while( 1 )
  {
    ...
    mail = ...
    flag = 1;
  }
}
```

CORRECTO

```
volatile boolean flag;
mail_t mail;
...
void main( void )
{
  while( 1 )
  {
    ...
    mail = ...
    MEMBAR;
    flag = 1;
  }
}
```

CORRECTO

J.M. Mendías 2022

Aplicaciones multihebra

sincronización y comunicación: FIFO (i)

- Cola de datos compartida escrita/leída por hebras diferentes
  - La hebra productora encola los datos
  - La hebra consumidora desencola los datos

```
volatile fifo_t fifo;
...
void main( void )
{
  while( 1 )
  {
    ...
    if( !fifo_empty( &fifo ) )
    {
      fifo_dequeue( &fifo, &data );
      ... procesa data...
    }
}

void isr( void )
{
  ...
  data = ...
  if( !fifo_full( &fifo ) )
    fifo_enqueue( &fifo, data );
}
```

MAIN consume

ISR produce

```
volatile fifo_t fifo;
...
void main( void )
{
  while( 1 )
  {
    ...
    data = ...
    if( !fifo_full( &fifo ) )
      fifo_enqueue( &fifo, data );
}

void isr( void )
{
  ...
  if( !fifo_empty( &fifo ) )
  {
    fifo_dequeue( &fifo, &data );
    ... procesa data ...
  }
}
```

MAIN produce

ISR consume

J.M. Mendías
2022

Aplicaciones multihebra

sincronización y comunicación: FIFO (ii)

- Una FIFO se implementa como un buffer circular:

```
typedef struct fifo {
  data_t data[MAXSIZE];
  uint16 head;
  uint16 tail;
  uint16 size;
} fifo_t;

void fifo_init( fifo_t *pfifo )
{
  pfifo->head = 0;
  pfifo->tail = 0;
  pfifo->size = 0;
}

boolean fifo_is_empty( fifo_t *pfifo )
{
  return (pfifo->size == 0);
}

boolean fifo_is_full( fifo_t *pfifo )
{
  return (pfifo->size == MAXSIZE-1);
}
```

```
void fifo_enqueue( fifo_t *pfifo, data_t data )
{
  pfifo->data[pfifo->tail++] = data;
  if( pfifo->tail == MAXSIZE )
    pfifo->tail = 0;
  INT_DISABLE;
  pfifo->size++;
  INT_ENABLE;
}

void fifo_dequeue( fifo_t *pfifo, char *data )
{
  *data = pfifo->buffer[pfifo->head++];
  if( pfifo->head == MAXSIZE )
    pfifo->head = 0;
  INT_DISABLE;
  pfifo->size--;
  INT_ENABLE;
}
```

secciones críticas

Esta implementación es **válida solo** para **1 productor y 1 consumidor**, para un mayor número de productores y/o consumidores es también necesario proteger los accesos a head y tail

Aplicaciones multihebra

sincronización y comunicación: FIFO (iii)

- Los dispositivos de entrada interrumpen cuando tienen un nuevo dato disponible
  - Si no se reciben datos, no hay interrupciones
  - Cada vez que el dispositivo interrumpe, la ISR (el productor) lee el dato y lo encola.

- Los dispositivos de salida interrumpen cuando están preparados para aceptar un nuevo dato.
  - Si no hay datos que enviar, interrumpen constantemente
    - En ausencia de datos, las interrupciones del dispositivo (o el propio dispositivo) deben estar deshabilitadas.
  - El productor, cada vez que encola un dato habilita las interrupciones del dispositivo.
    - El dispositivo interrumpirá inmediatamente y la ISR (el consumidor) desencola y envía el dato.
    - Mientras la cola no esté vacía, las interrupciones se encadenan provocando sucesivos envíos.
  - Si tras un envío, la ISR detecta que la cola está vacía, deshabilita las interrupciones del dispositivo.

J.M. Mendias
2022

Aplicaciones multihebra

sincronización y comunicación: FIFO (iv)

MAIN produce

ISR consume

```
volatile fifo_t fifo;
...
void main( void )
{
  while( 1 )
  {
    ...
    data = ...
    if( !fifo_is_full( &fifo ) )
    {
      fifo_enqueue( &fifo, data );
      ...arma el dispositivo...
    }
}

void isr( void )
{
  ...
  if( fifo_is_empty( &fifo ) )
    ...desarma el dispositivo...
  else
  {
    fifo_dequeue( &fifo, &data );
    ...escribe data en la salida...
  }
}
```

J.M. Mendias
2022

Aplicaciones multihebra

sincronización y comunicación: FIFO (v)

```
static volatile fifo_t fifoTX;

void uart0_putchar( char ch )
{
  while( fifo_is_full( &fifoTX ) );
  fifo_enqueue( &fifoTX, ch );

  INTMSK &= ~BIT_UTXD0;
}


void uart0_isrTX( void )
{
  char ch;

  if( fifo_is_empty( &fifoTX ) )
    INTMSK |= BIT_UTXD0;
  else
  {
   fifo_dequeue( &fifoTX, &ch );
   UTXH0 = ch;
  }

  I_ISPC = BIT_UTXD0;
}
```

ENVIO

```
static volatile fifo_t fifoRX;

char uart0_getchar( void )
{
  char ch;

  while( fifo_is_empty( &fifoRX ) );
  fifo_dequeue( &fifoRX, &ch );
  return ch;
}

void uart0_isrRX( void )
{
  if( !fifo_is_full( &fifoRX ) )
    fifo_enqueue( &fifoRX, URXH0 );

  I_ISPC = BIT_URXD0;
}
```

RECEPCIÓN

Aritmética en punto fijo

representación de números reales

- Típicamente los microcontroladores no tienen soporte hardware para la aritmética de punto flotante.

- Para trabajar en C con números reales hay 2 alternativas:
  - Emular por software las operaciones de punto flotante
    - El programador usa los tipos `float`/`double`
    - El compilador enlaza las correspondientes funciones de la biblioteca aritmética.
    - Código simple, pero largo y lento.
  - Usar aritmética en punto fijo
    - El programador usa variantes del tipo `int` e implementa los operadores necesarios.
    - Código algo más complejo, pero compacto y rápido.

J.M. Mendías
2022

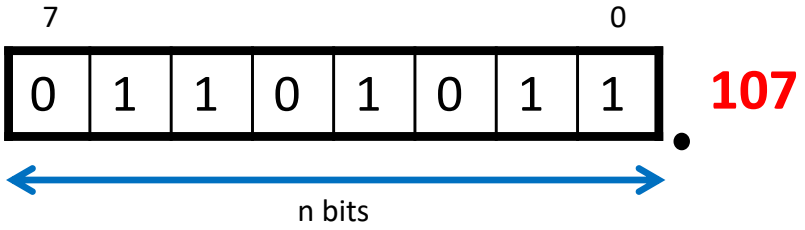
Aritmética en punto fijo

representación de datos (i)

- En punto fijo, la posición del punto en la cadena de bits es fija y queda implícita en el código.
  - En los números enteros se asume que el punto está situado a la derecha del LSB

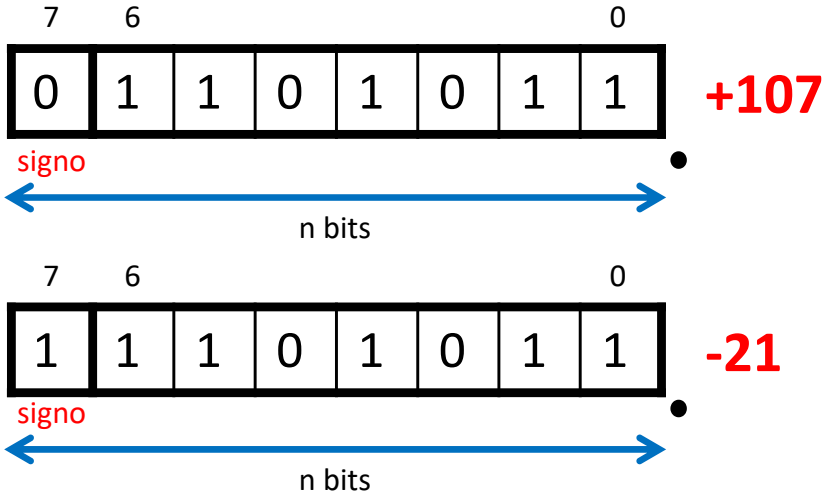
- Los números enteros sin signo se representan en binario

$$v(\underline{x}) = \sum_{i=0}^{n-1} 2^i \cdot x_i$$



- Los números enteros con signo se representan en C2

$$v(\underline{x}) = -2^{n-1} \cdot x_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot x_i$$



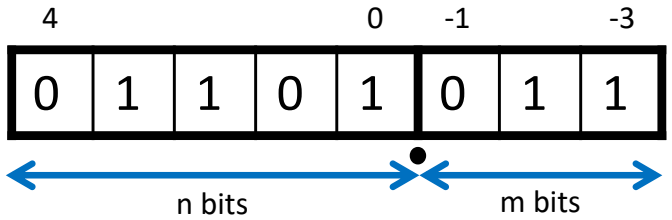
J.M. Mendías
2022

Aritmética en punto fijo

representación de datos (ii)

- Los números reales sin signo se representan en punto fijo en binario

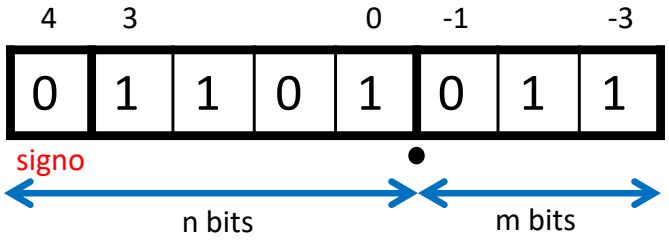
$$v(\underline{x}) = \sum_{i=0}^{n-1} 2^i \cdot x_i + \sum_{i=0}^{m-1} 2^{-i} \cdot x_{-i}$$



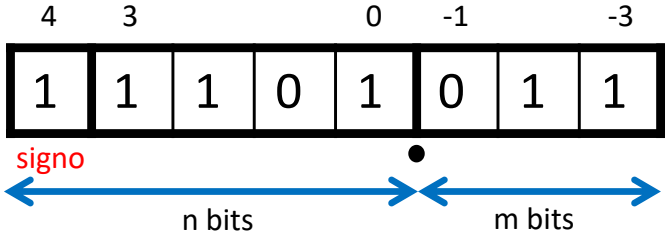
13.375

- Los números reales con signo se representan en punto fijo en C2

$$v(\underline{x}) = -2^{n-1} \cdot x_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot x_i + \sum_{i=0}^{m-1} 2^{-i} \cdot x_{-i}$$



+13.375



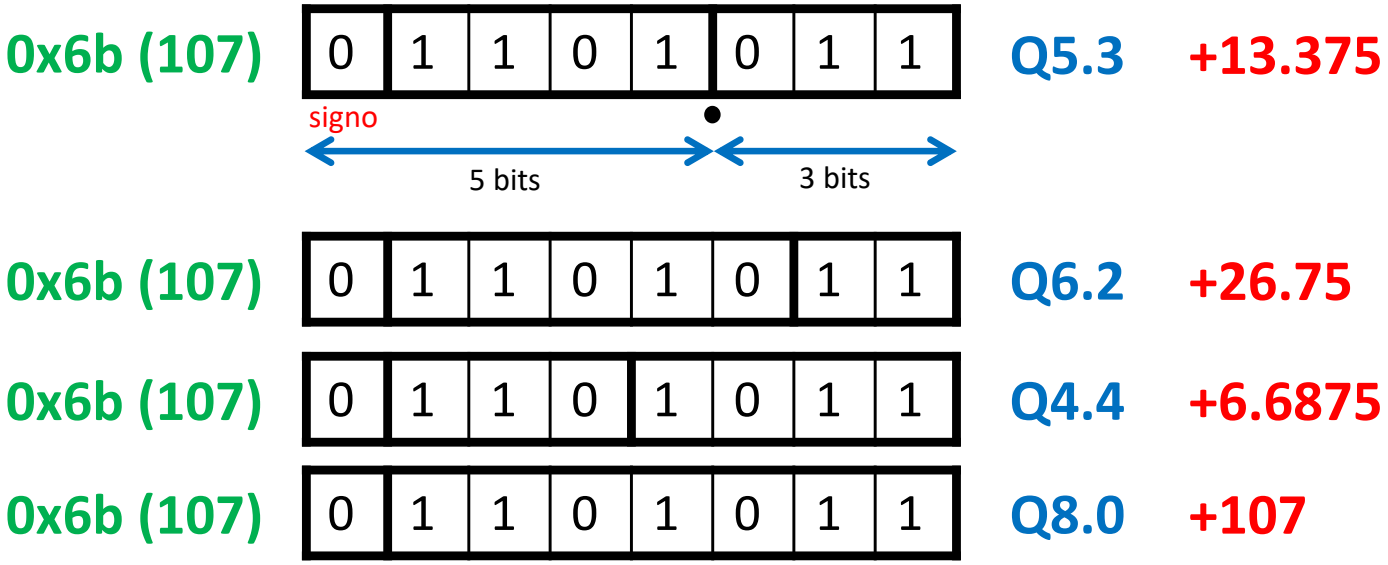
-2.625

J.M. Mendías 2022

Aritmética en punto fijo

notación Q

- Para indicar la representación concreta de punto fijo (típicamente con signo) utilizada en un tipo de dato, se usa la notación Qn.m, donde:
  - n es el número de bits enteros (excluyendo, o no, el bit de signo según convenio)
  - m es el número de bits decimales



- En ocasiones se usa únicamente Qm
  - si la anchura de la representación se sobrentiende (8/16/32 bits)

Aritmética en punto fijo

rango vs. resolución

- El rango representable en punto fijo Qn.m es $[-2^{n-1}, 2^{n-1}-2^{-m}]$
  - Por ejemplo, el rango representable en Q5.3 es $[-2^4, 2^4-2^{-3}] = [-16, +15{,}875]$
- La resolución en punto fijo Qn.m es 2^{-m} y el error máximo de representación $\pm 2^{-(m+1)}$
  - Por ejemplo, en Q5.3 la resolución es $2^{-3} = 0.125$ y el error $\pm 2^{-4} = \pm 0.0625$
- Así, a tamaño de palabra fijo:
  - Si m es muy pequeño, aumenta el error al representar números con decimales
  - Si m es muy grande, aumenta el riesgo de overflow

representación	rango	resolución	error
Q5.3	[-16, +15.875]	0.125	± 0.0625
Q6.2	[-32, +31.75]	0.25	± 0.125
Q4.4	[-8, +7.9375]	0,0625	± 0.03125
Q8.0	[-128, +127]	1	± 0.5
Q1.7	[-1, +0.9921875]	0,0078125	± 0.00390625

Aritmética en punto fijo

suma y resta (i)

- Para sumar/restar 2 números con la misma representación Qn.m
  - Al estar los puntos alineados, puede utilizarse la aritmética entera
  - Sin embargo, en general, el resultado requiere ser representado en Qn+1.m

	0	0	0	0	1	1	1	0	Q5.3	+1.75
+	0	1	1	0	1	0	1	1	Q5.3	+13.375
	0	1	1	1	1	0	0	1	Q5.3	+15.125

	0	1	1	0	0	1	0	0	Q5.3	+12.5
+	0	1	1	0	1	0	1	1	Q5.3	+13.375
CORRECTO	0	1	1	0	0	1	1	1	Q6.3	+25.875
INCORRECTO	1	1	0	0	1	1	1	1	Q5.3	-6.125

J.M. Mendías
2022

Aritmética en punto fijo

suma y resta (ii)

- Cuando la representación del resultado de la suma/resta debe ser la misma que la de los operandos, existen 2 alternativas:
  - Wrap: descartar siempre el bit mas significativo del resultado

	0	1	1	0	0	1	0	0	Q5.3 +12.5
+	0	1	1	0	1	0	1	1	Q5.3 +13.375
	~~0~~	1	1	0	0	1	1	1	Q5.3 -6.125

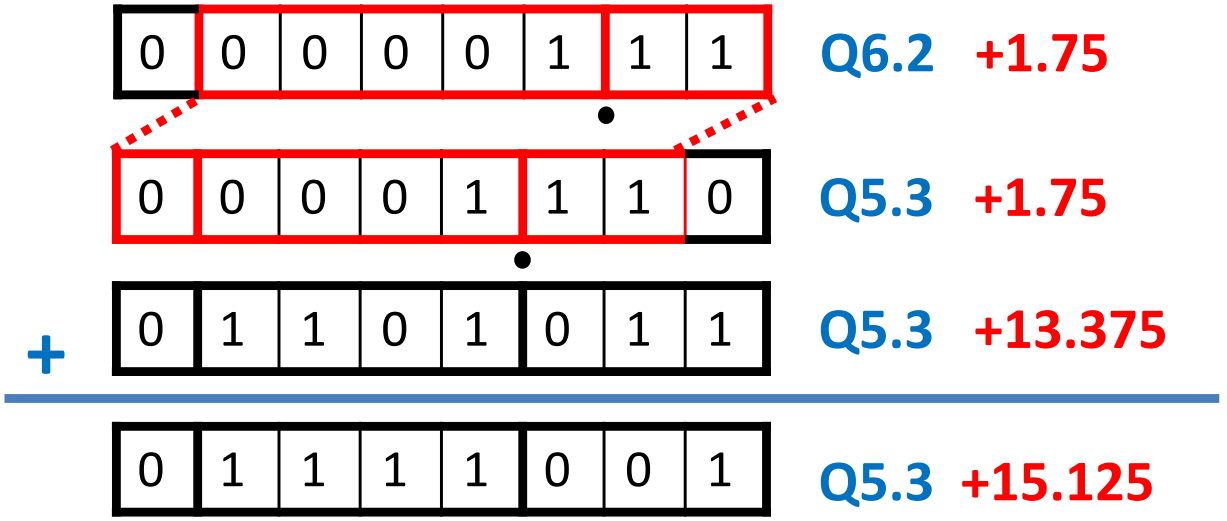
  - Saturar: en caso de producirse overflow/underflow devolver el máximo/mínimo valor representable.

	0	1	1	0	0	1	0	0	Q5.3 +12.5
+	0	1	1	0	1	0	1	1	Q5.3 +13.375
	0	1	1	1	1	1	1	1	Q5.3 +15.875

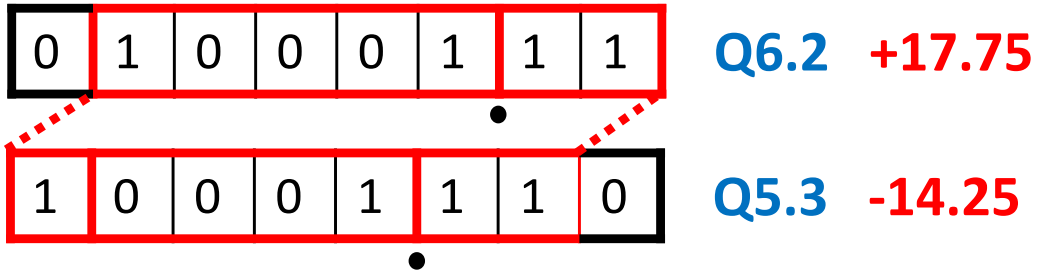
Aritmética en punto fijo

suma y resta (iii)

- Si los operandos a sumar/restar tienen representaciones distintas
  - Previamente deben alinearse los puntos mediante desplazamiento (reescalado)
  - Estos desplazamientos pueden provocar overflow o pérdida de resolución



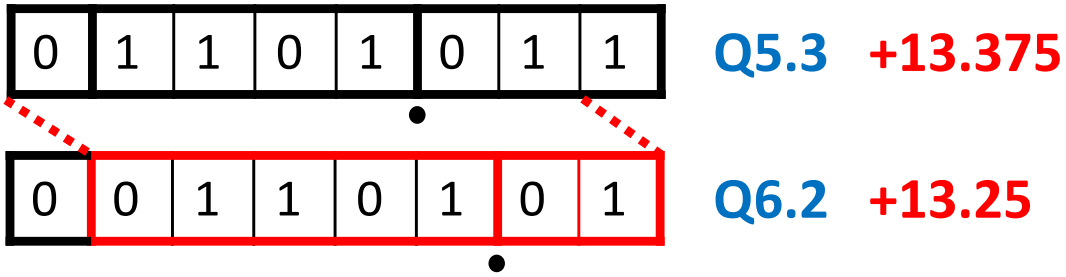
INCORRECTO



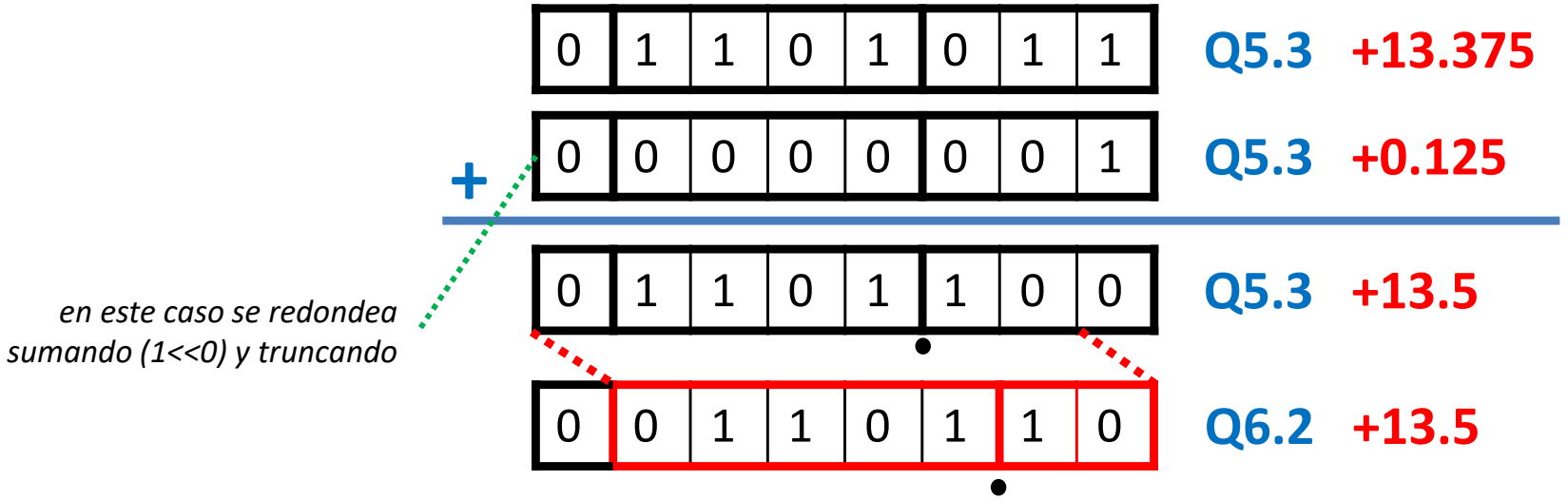
Aritmética en punto fijo

suma y resta (iv)

- Cuando el reescalado de un dato supone una pérdida de resolución, existen 2 alternativas:
  - Truncar: descartar siempre los bits menos significativos



  - Redondear: Aproximar al número representable más cercano

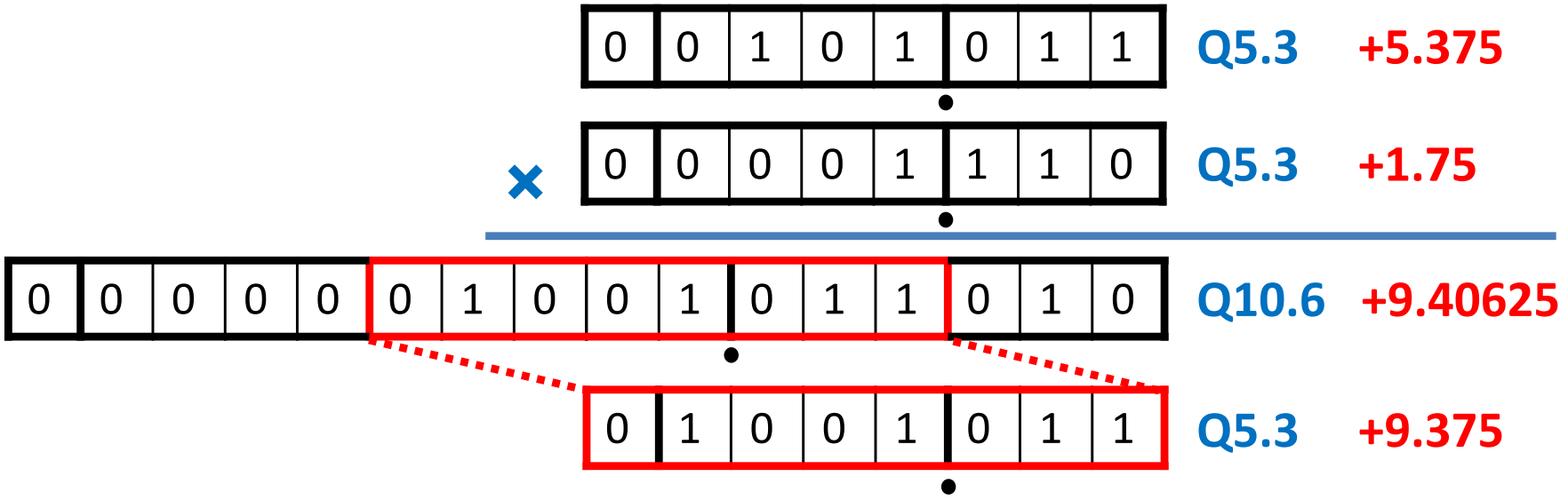


J.M. Mendías
2022

Aritmética en punto fijo

multiplicación y división (i)

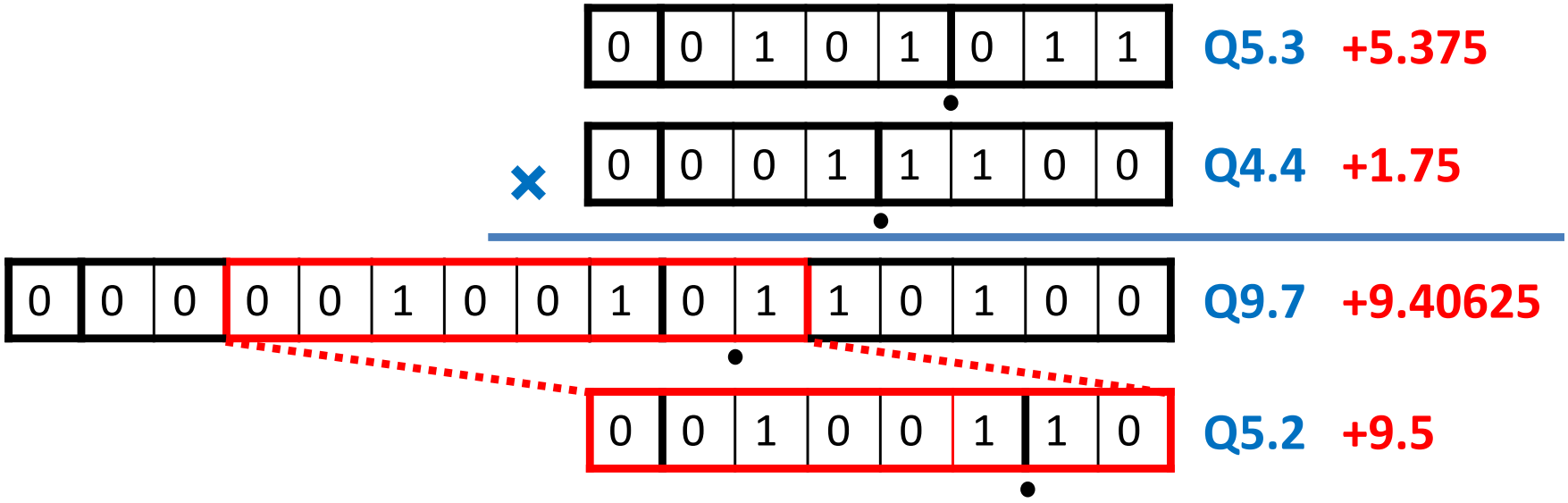
- Para multiplicar/dividir 2 números con la misma representación Qn.m
  - Se utiliza aritmética entera pero deben realizarse correcciones de escala
    - Multiplicación: $Z = (X \cdot Y) \div 2^m \Rightarrow Z = (X \cdot Y) >> m$
    - División: $Z = (X \cdot 2^m) \div Y \Rightarrow Z = (X << m) / Y$
  - Puede producirse overflow y perdida de resolución
    - Se compensan con las mismas técnicas aplicadas con la suma



Aritmética en punto fijo

multiplicación y división (ii)

- Si los operandos tienen representaciones distintas
  - La corrección de escala se ajusta según el número de bits decimales de cada uno.
  - Por ejemplo, para multiplicar 2 números en formato $Qn_1.m_1$ y $Qn_2.m_2$ de manera que el resultado quede en formato $Qn_3.m_3$
    - $Z = (X \cdot Y) >> m_1+m_2-m_3$



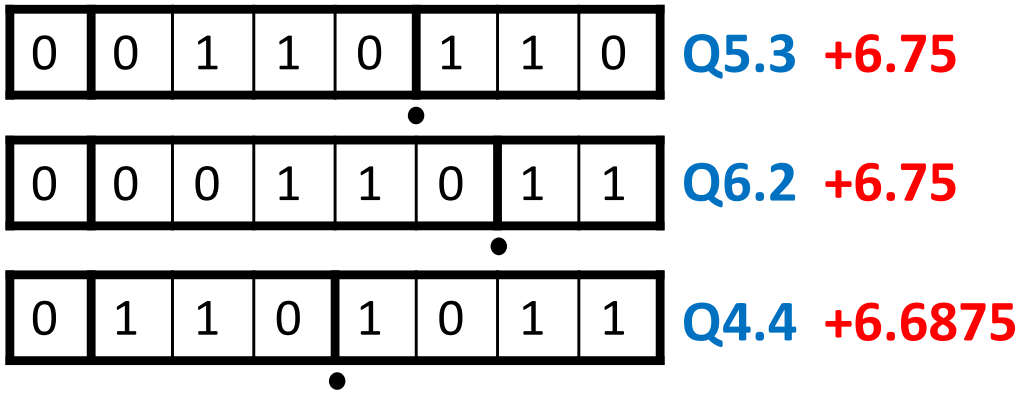
J.M. Mendías 2022

Aritmética en punto fijo

programación en C (i)

- C no soporta nativamente el punto fijo, por ello es necesario:
  - Declarar las variables y constantes de tipo entero.
    - El tipo entero elegido deberá ser los suficientemente ancho para contener los datos.
    - La posición del punto quedará implícita y podrá ser diferente para cada dato.
  - Usar los operadores enteros
    - Teniendo en cuenta los factores de escala a usar en cada operación aritmética.
    - Redondeando / truncando / saturando los resultados.
    - Gestionando los potenciales overflows.
  - Convertir manualmente los literales reales en sus correspondientes enteros.
  - Reglas de conversión de literales (real ↔ punto fijo Qn.m) :
    - $literal_{int} = redondeo(literal_{real} \cdot 2^m)$
    - $literal_{real} = literal_{int} \div 2^m$

literal real	repr.	literal entero
6.7	Q5.3	$6.7 \cdot 2^3 = 53.6 \approx 54$
	Q6.2	$6.7 \cdot 2^2 = 26.8 \approx 27$
	Q4.4	$6.7 \cdot 2^4 = 107.2 \approx 107$



J.M. Mendías
2022

Aritmética en punto fijo

programación en C (i)


```
int8 x, y, z;
```
operandos y resultado en representación Q5.3

```
z = x + y;
```
suma con wrapping (con riesgo de overflow)

```
int8 x, y;
int16 z;

z = (int16)x + (int16)y;
```
operandos en representación Q5.3
resultado en representación Q6.3
suma (sin riesgo de overflow)

```
int8 x, y, z;
int16 aux;

aux = (int16)x + (int16)y;
if( aux > INT8_MAX )
  z = INT8_MAX;
else if( aux < INT8_MIN )
  z = INT8_MIN;
else
  z = temp;
```
datos en representación Q5.3
resultado auxiliar en representación Q6.3
suma (sin riesgo de overflow)
satura el resultado en caso de overflow

J.M. Mendías
2022

Aritmética en punto fijo

programación en C (ii)

```
#define QM 3 ····· numero de bits decimales de la representación Q5.3

int8 x, y, z; ····· datos en representación Q5.3
int16 aux;  ····· resultado auxiliar en representación Q10.6

aux = (int16)x * (int16)y;  ····· multiplica (sin riesgo de overflow)
z = aux >> QM; ····· corrige la escala del resultado y trunca (con riesgo de overflow)
```

```
int8 x, y, z; ····· datos en representación Q5.3
int16 aux;····· resultado auxiliar en representación Q10.6

aux = (int16)x * (int16)y;  ····· multiplicación (sin riesgo de overflow)
aux = aux + (1 << QM-1); ····· redondea
aux = aux >> QM; ····· corrige la escala del resultado (con riesgo de overflow)
if( aux > MAX_INT8 )
  z = MAX_INT8;
else if( aux < MIN_INT8 )
  z = MIN_INT8;
else
  z = aux;
```

satura el resultado en caso de overflow

```
int8 x, y, z; ····· datos en representación Q5.3
int16 aux;····· dividendo auxiliar en representación Q10.6

aux = (int16)x << QM; ····· escala el dividendo
aux = aux + (y >> 1); ····· redondea
z = aux / y;····· divide y trunca
```

J.M. Mendías 2022

Aritmética en punto fijo

programación en C (iii)

- Realizar una función que calcule en punto fijo Q4.12 el polinomio:

$$z = ax^2 + bx + c = (ax + b)x + c$$
$$donde:\ a = 0.6054, b = 2.3473, c = -5.6855$$

```
#define QM 12 ····························· número de bits decimales de la representación Q4.12

int16 poly( int16 x )
{
  const int16 a = 2480; ··················· 0.6054 en Q4.12: 0.6054·2^12 = 2479.7 ≈ 2468
  const int16 b = 9615; ··················· 2.3473 en Q4.12: 2.3473·2^12 = 9614.5 ≈ 9615
  const int16 c = -23288; ················· -5.6855 en Q4.12: -5.6855·2^12 = -23287.8 ≈ -23288

  int32 aux; ····························· resultado auxiliar en representación Q8.24

  aux = ((a*(int32)x) >> QM) + b;
  aux = ((aux*x) >> QM) + c;              } calcula el resultado
  return aux; ····························· trunca
}
```

- Por ejemplo, para x = 1.023 el polinomio vale z= -2.6505
  - x = 1.023 ≈ 1.02294921875 = $(4190)_{Q4.12}$
  - poly(4190) = -10859
  - z = $(-10869)_{Q4.12}$ = -2,653564453125 ≈ -2.6505

} cálculos en punto fijo Q4.12

J.M. Mendías
2022

Aritmética en punto fijo

programación en C (iv)

- Para facilitar la legibilidad pueden declararse tipos y macros.

```
#define FADD(a,b) ((a)+(b))
#define FSUB(a,b) ((a)-(b))
#define FMUL(a,b,q) (((a)*(b))>>(q))
#define FDIV(a,b,q) (((a)<<(q))/(b))
```
} operaciones básicas en punto fijo sin redondeo ni saturación

```
#define TOFIX(t,d,q) ((t)((d)*(double)(1ULL<<(q))))
```
····· utilidad de conversión desde punto flotante

```
typedef int32 fix32;
typedef int16 fix16;
```
} declaración de los tipos en punto fijo

```
#define QM 12
```
····· número de bits decimales de la representación en punto fijo

```
fix16 poly( fix16 x )
{
  const fix16 a = TOFIX( fix16, 0.6054, QM );
  const fix16 b = TOFIX( fix16, 2.3473, QM );
  const fix16 c = TOFIX( fix16, -5.6855, QM );
```
} los argumentos deben ser constantes para que el compilador calcule el valor entero sin enlazar las funciones de punto flotante

```
  fix32 aux;
```
····· resultado auxiliar en representación Q8.24

```
  aux = FMUL( a, (fix32)x, QM );
  aux = FADD( aux, b );
  aux = FMUL( aux, x, QM );
  aux = FADD( aux, c );
```
} calcula el valor como (a·x+b)·x+c

```
  return aux;
```
····· trunca

```
}
```

Sistemas guiados por tiempo

introducción

- Existe un gran número de sistemas empotrados deben soportar la llamada periódica a una función (o conjunto de ellas).
  - La velocidad del vehículo debe medirse cada 0,5 s.
  - Un display debe refrescarse 40 veces/s.
  - La vibración del motor debe muestrearse 1000 veces/s
  - Un supervisor debe leer un conjunto distribuido de sensores cada segundo.

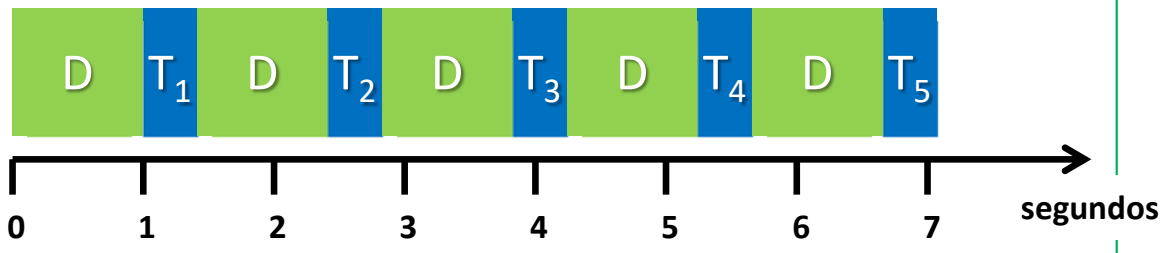
- Por otro lado, es muy común que un sistema empotrado realice indefinidamente una secuencia finita de acciones con cierta temporización y según la ocurrencia de eventos externos.
  - El usuario selecciona en un dial un elemento
  - A continuación, un motor se activa durante 1 s para mover un engranaje.
  - 5 s después, se abre una válvula hasta la activación de un sensor.
  - Finalmente se esperan 30 s y se reinicia el proceso.

Tareas periódicas

super-loop: primera aproximación

INCORRECTO

```
void task_init( void );
void task( void );
void delay( uint16 ms );

void main( void )
{
  sys_init();
  task_init();

  while( 1 )
  {
     delay_ms( 1000 );
     task();
  };
}
```



- La tarea se ejecuta 1s después del fin de su anterior ejecución:
  - NO se realiza 1 vez/s:
    - El tiempo de ejecución de propia la tarea se añade al retardo.
  - Podría conseguirse periodicidad si *delay* = 1s – *duración de la tarea*, pero:
    - ¿que pasa si la tarea no tiene una duración fija?
    - ¿que pasa si se producen interrupciones?
- Además la CPU está el 100% del tiempo activa

Tareas periódicas

foreground-background: segunda aproximación (i)

```
...
void timer0_isr( void ) __attributte__ ((interrupt ("IRQ")));

void main( void )
{
  sys_init();
  task_init();

  timer0_open_tick( timer0_isr, 1 );
  while( 1 )
  {
    sleep();
    task();
  }
}

void timer0_isr( void )
{
  I_ISPC = BIT_TIMER0;
}

void sleep( void )
{
  CLKCON |= (1 << 2);
}
```

1. Se programa el timer0 para generar int. periódicas.
2. La CPU se suspende.
3. Cada interrupción del timer0 despierta a la CPU.
4. La CPU ejecuta la tarea.
5. El bucle *while* repite el ciclo de suspensión-ejecución.

la RTI sólo borra el bit de interrupción pendiente

Pone a la CPU en estado IDLE: la CPU se para y consume un 60% menos de energía. Opcionalmente podrían apagarse el resto de módulos excepto el PWMTIMER (que es el que genera la int.)

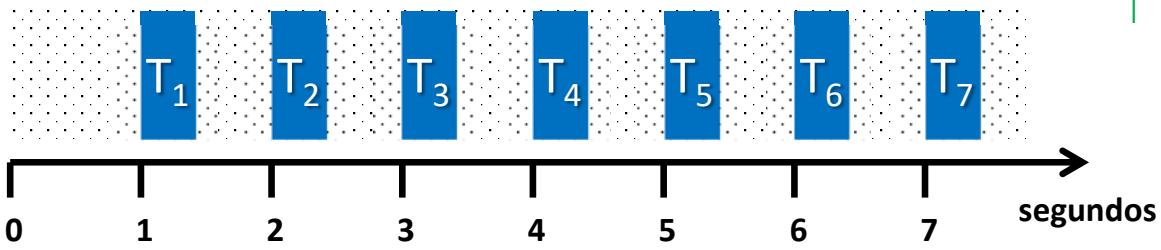
Tareas periódicas

foreground-background: segunda aproximación (ii)

```
void main( void )
{
  sys_init();
  task_init();

  timer0_open_tick( timer0_isr, 1 );
  while( 1 )
  {
    sleep();
    task();
  }
}
```

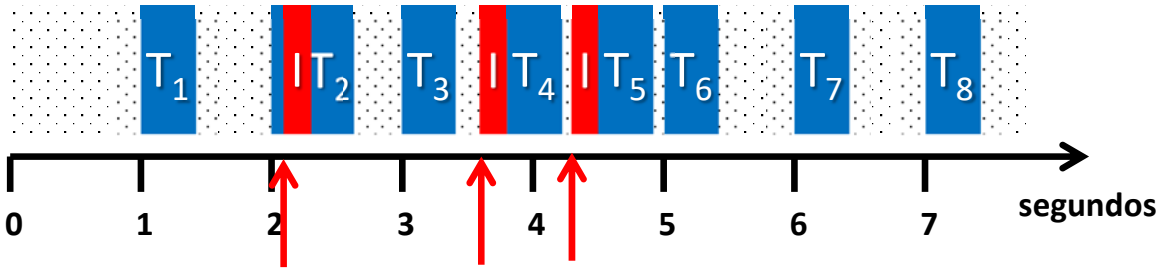
una única fuente de interrupción (timer0)

CORRECTO



múltiples fuentes de interrupción (timer0 y otras)

INCORRECTO



- La tarea se ejecuta 1s después del comienzo de la anterior ejecución
  - Lo hace hasta completarse pero con eventuales interrupciones.
  - Se realiza exactamente 1 vez/s (siempre y cuando su duración sea inferior a 1s y no haya otras fuentes de interrupción activas).
- La CPU está activa sólo durante la ejecución de la tarea.

J.M. Mendias
2022

Tareas periódicas

foreground-background: tercera aproximación (i)

```
void timer0_isr( void ) __attributte__ ((interrupt ("IRQ")));

void main( void )
{
  sys_init();
  task_init();

  timer0_open_tick( timer0_isr, 1 );
  while( 1 )
    sleep();
}

void timer0_isr( void )
{
  I_ISPC = BIT_TIMER0;
  task();
}
```

La tarea se ejecuta **dentro** de la RTI

1. Se programa el timer0 para generar int. periódicas.
2. La tarea se ubica dentro de la RTI del timer0.
3. La CPU se suspende.
4. Cada interrupción del timer0 despierta a la CPU.
5. La CPU ejecuta la tarea.
6. Al volver de la RTI al bucle *while*, se repite el ciclo de suspensión-ejecución.

- La tarea se ejecuta 1s después del comienzo de la anterior ejecución
  - De manera ininterrumpida hasta completarse (en la RTI las int. están desabilitadas)
  - Se realiza exactamente 1 vez/s (siempre y cuando su duración sea inferior a 1s y no haya otras fuentes de interrupción activas).
  - Si la tarea es pesada la latencia de servicio a otras interrupciones puede ser alta.
  - Si la tarea es compleja puede desbordar la pila de IRQ.

Tareas periódicas

foreground-background: tercera aproximación (ii)

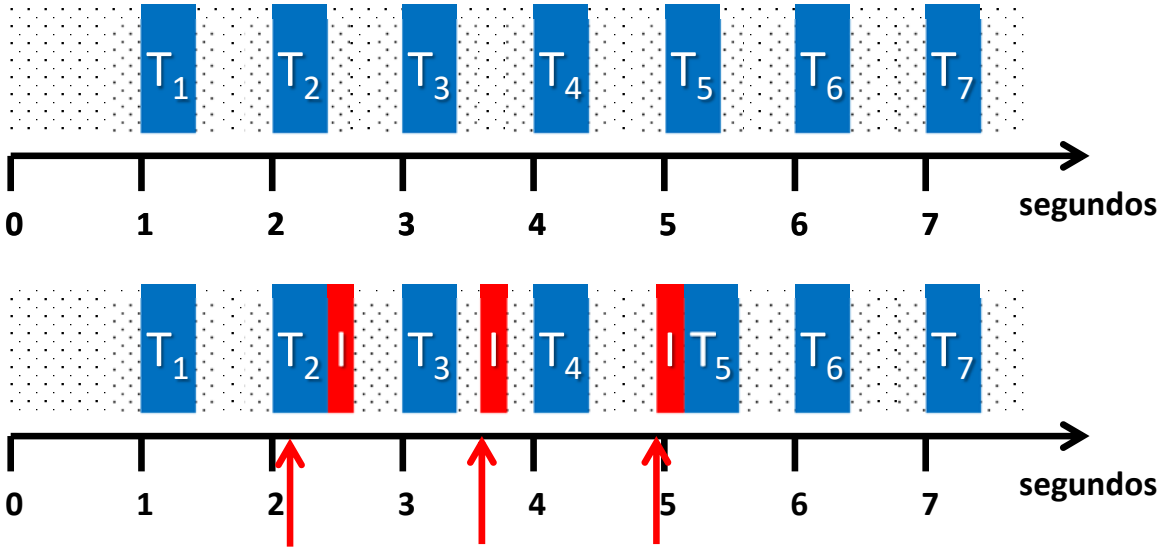
```
void main( void )
{
  sys_init();
  task_init();

  timer0_open_tick( timer0_isr, 1 );
  while( 1 )
    sleep();
}

void timer0_isr( void )
{
  I_ISPC = BIT_TIMER0;
  task();
}
```

una única fuente de interrupción (timer0)

CORRECTO

múltiples fuentes de interrupción (timer0 y otras)

CORRECTO



Tareas periódicas

foreground-background: cuarta aproximación

```
void main( void )
{
  sys_init();
  task_init();

  flag = 0;
  timer0_open_tick( timer0_isr, 1 );
  while( 1 )
  {
    sleep();
    if( flag )
    {
      flag = 0;
      task();
    }
  }
}
```

```
void timer0_isr( void )
{
  I_ISPC = BIT_TIMER0;
  flag = 1;
}
```

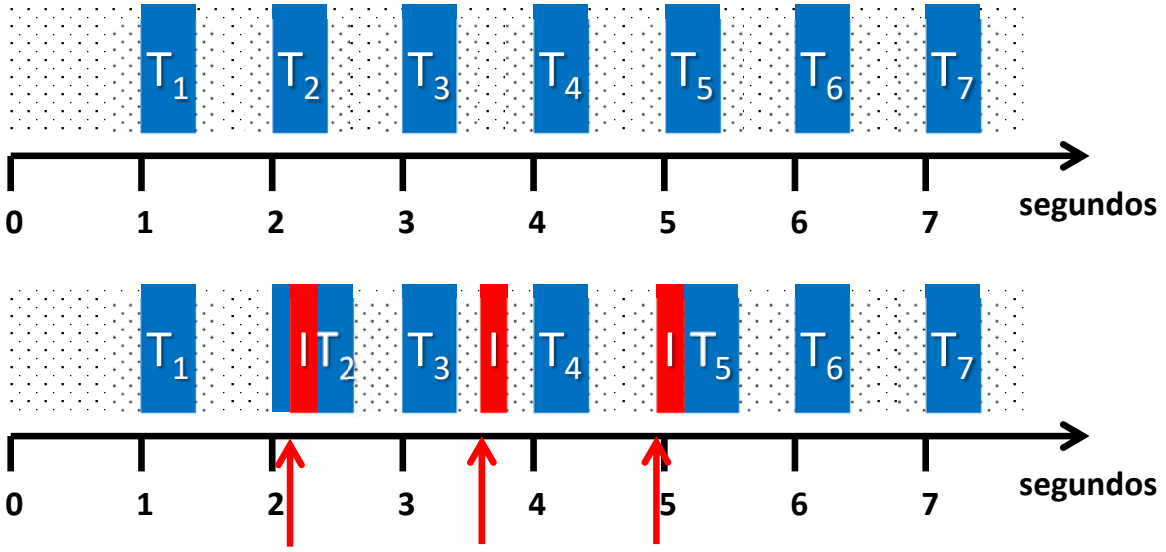
1. Se programa el timer0 para generar int. periódicas.
2. La RTI del timer0 activa un flag;
3. La CPU se suspende.
4. Cada interrupción del timer0 despierta a la CPU.
5. La CPU ejecuta la tarea si el flag está activado.
6. El bucle *while* repite el ciclo de suspensión-ejecución.

una única fuente de interrupción (timer0)

CORRECTO

múltiples fuentes de interrupción (timer0 y otras)

CORRECTO



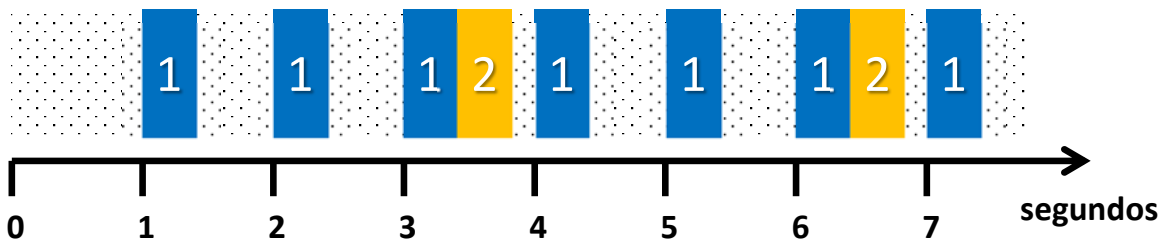
Tareas periódicas

foreground-background: quinta aproximación

- Si hay dos o más tareas con diferentes periodos:
  - Puede no ser posible dedicar un timer distinto para disparar cada una de ellas.
  - En su lugar, los disparos se realizan relativos ticks generados por un único timer.

```
#define TICKSxSEC  (100)

void main( void )
{
  sys_init();
  task_init();

 flagTask1 = 0;
 flagTask2 = 0;
 timer0_open_tick( timer0_isr, TICKSxSEC );
  while( 1 )
  {
    sleep();
    if( flagTask1 )
      { flagTask1 = 0; task1(); }
    if( flagTask2 )
      { flagTask2 = 0; task2(); }
  }
}
```

```
void timer0_isr( void )
{
  static uint16 cont100ticks = 100;
  static uint16 cont300ticks = 300;

  if( !(--cont100ticks) )
    { cont100ticks = 100; flagTask1 = 1; }

  if( !(--cont300ticks) )
    { cont300ticks = 300; flagTask2 = 1; }

  I_ISPC = BIT_TIMER0;
}
```



J.M. Mendías
2022

Tareas periódicas

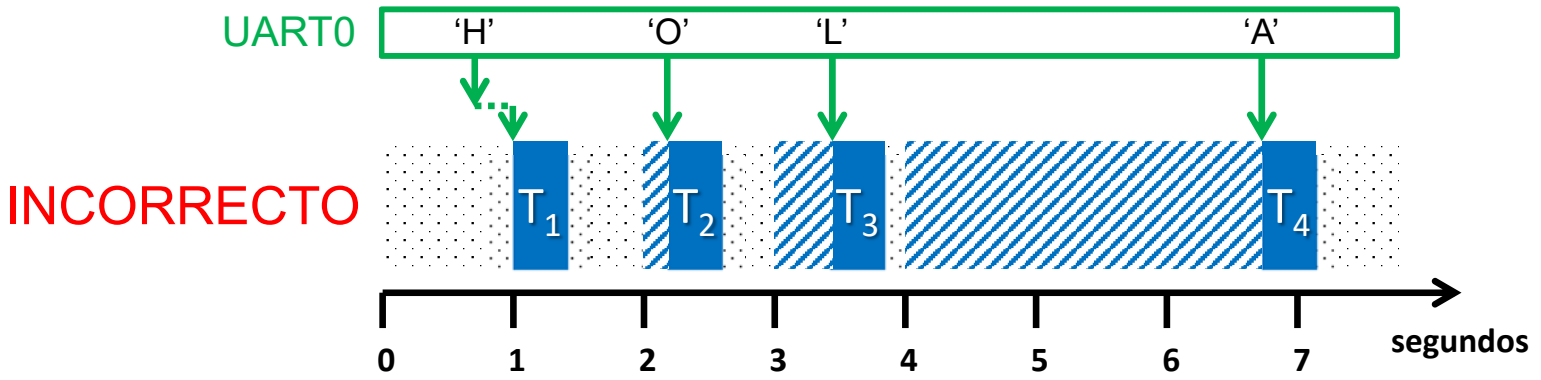
sobre pooling (i)

- El polling con espera activa no debe usarse en tareas periódicas
  - Si la transmisión de datos no es periódica o tiene periodo mayor que el de la tarea, puede alargar impredeciblemente su tiempo de ejecución y bloquear el sistema

```
void main( void )
{
  sys_init();
  task_init();

  timer0_open_tick( timer0_isr, 1 );
  while( 1 )
  {
    sleep();
    task();
  }
}
```

```
void task( void )
{
  ch = uart0_getchar();
  ...
}
```

```
char uart0_getchar( void )
{
  while( !( UFSTAT0 & 0xF ) );
  return URXH0;
}
```



J.M. Mendias
2022

Tareas periódicas

sobre pooling (ii)

- En tarea periódicas, alternativamente al polling convencional, se usa:
  - Polling con timeout: espera la llegada de un dato por un tiempo máximo
    - Evita que el sistema se bloquee aunque alarga la duración de la tarea.
  - Polling periódico: lee el dato (sin espera) solo si está disponible.
    - Evita que el sistema se bloquee y la duración de la tarea no se altera.
  - Polling ciego: lee el dato (sin espera) siempre
    - Solo aplicable con datos de disponibilidad permanente o con una tasa fija de transmisión igual al periodo de la tarea.

CON TIMEOUT

```
void task( void )
{
  ch = uart0_timeout_getchar( 300 );
  ...
}
```

```
char uart0_timeout_getchar( uint16 ms )
{
  timer3_start_timeout( 10*ms );
  while( !(UFSTAT0 & 0xf) && !timer3_timeout() );
  if( timer3_timeout() )
    return UART0_TIMEOUT;
  else
    return URXH0;
}
```

```
void task( void )
{
  if( UFSTAT0 & 0xF )
    ch = URXH0;
  ...
}
```

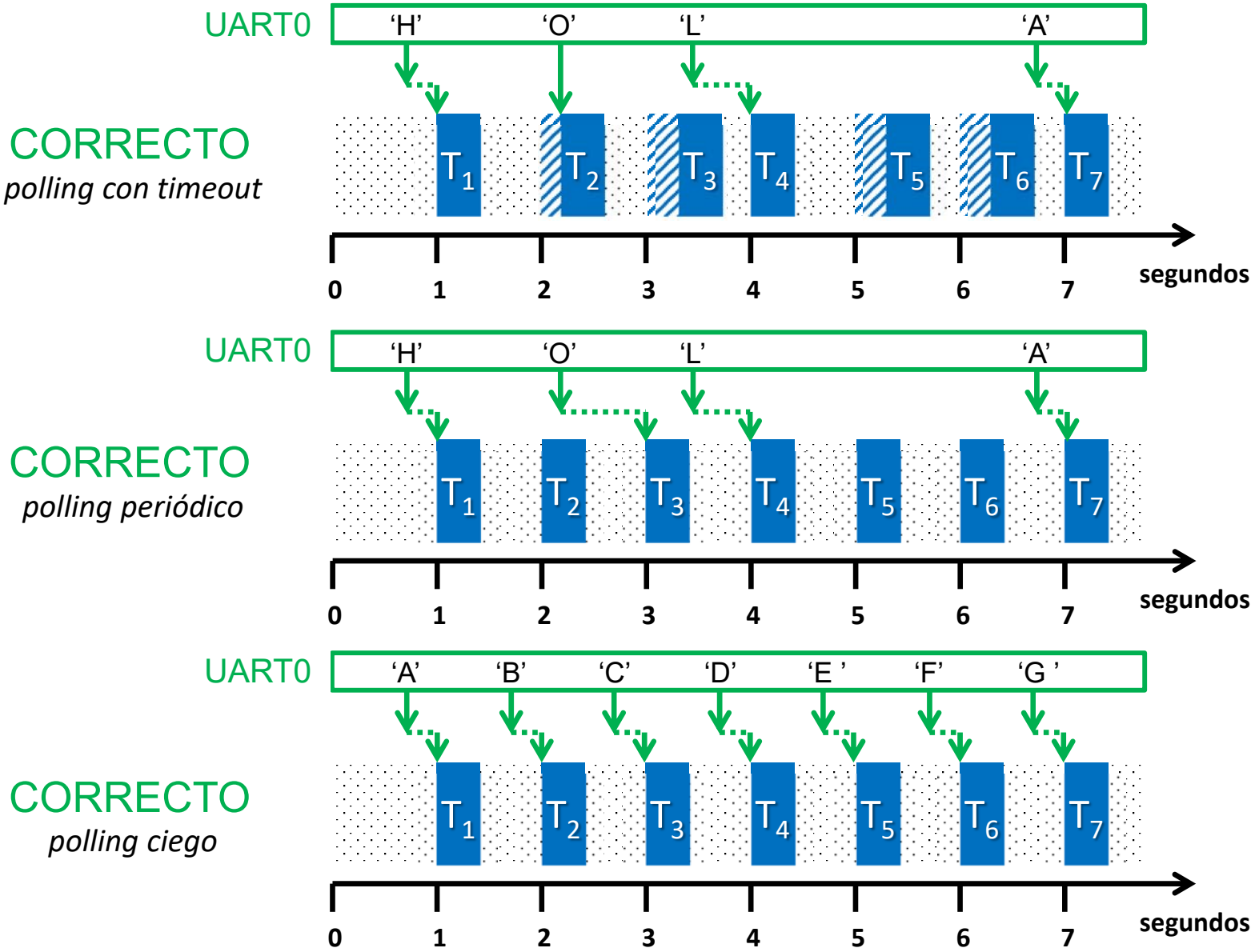
PERIODICO

```
void task( void )
{
  ch = URXH0;
  ...
}
```

CIEGO

Tareas periódicas

sobre pooling (iii)



J.M. Mendías
2022

Tareas periódicas

sobre pooling (iv)

- El pooling periódico es una alternativa al uso de interrupciones en sistemas dirigidos por tiempo.
  - Las tareas convencionales encolan/desencolan en FIFOs los datos a transmitir
  - Una tarea periódica adicional recibe/envía datos encolados

```
static volatile fifo_t fifoTX;
static volatile fifo_t fifoRX;

void uart0_putchar( char ch )
{
  while( fifo_is_full( &fifoTX ) );
  fifo_enqueue( &fifoTX, ch );
}

char uart0_getchar( void )
{
  char ch;

  while( fifo_is_empty( &fifoRX ) );
  fifo_dequeue( &fifoRX, &ch );
  return ch;
}
```

```
void spooler_task( void )
{

  if( !fifo_is_full( &fifoRX )
      && (UFSTAT0 & 0x0F) )
    fifo_enqueue( &fifoRX, URXH0 );

  if( !fifo_is_empty( &fifoTX )
      && !(UFSTAT0 & (1<<9) )
  {
    fifo_dequeue( &fifoTX, &ch );
    UTXH0 = ch;
  }

}
```

recepción

envío

J.M. Mendías
2022

Sistemas cyclic executive

introducción

- Simplificando, una tarea periódica puede caracterizarse por:
  - t_i: periodo de activación (cada cuanto tiempo debe ejecutarse)
  - c_i: tiempo máximo de cómputo
  - Se asume que toda tarea dispone de un plazo de ejecución equivalente a su periodo, es decir, que entre 2 activaciones puede ejecutarse en cualquier momento.

- Para ejecutar sin expropiación un conjunto arbitrario de tareas periódicas de manera que todas cumplan sus plazos existen 2 alternativas básicas:
  - Planificación cíclica (estática)
    - Las tareas se ejecutan siguiendo una planificación fija calculada off-line.
    - El sistema lleva la cuenta del tiempo y ejecuta cada tarea en el momento precalculado.
    - No existe planificador, solo un despachador cíclico (cycle executive)
  - Planificador cooperativo basado en prioridades (dinámica):
    - Las tareas se ejecutan siguiendo una planificación calculada en tiempo real.
    - El sistema dispone de un planificador basado en prioridades que dado un conjunto de tareas preparadas determina el momento en que pasan a ejecución.
    - La prioridad estática de cada tarea se asigna en función de sus características.
  - Asumen que el número de tareas es fijo y conocido así como los valores de t_i y c_i.

J.M. Mendías 2022

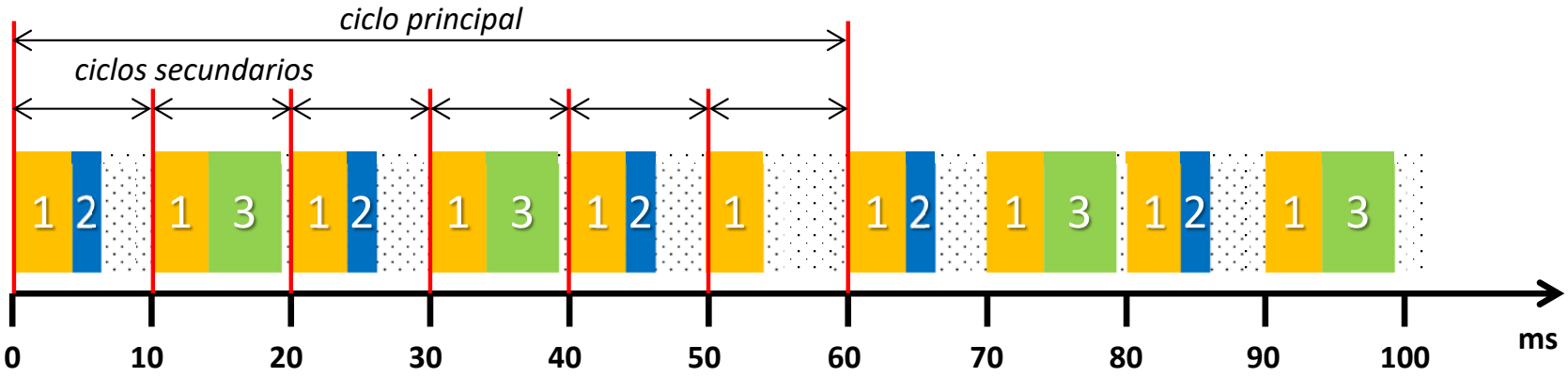
Sistemas cyclic executive

planificación cíclica

- Planificar un conjunto de tareas arbitrario es un problema NP-hard
  - Por ejemplo, una posible planificación de un sistema con 3 tareas:

tarea 1: $t_1 = 10$ ms, $c_2 = 4$ ms
tarea 2: $t_2 = 20$ ms, $c_2 = 2$ ms
tarea 3: $t_3 = 30$ ms, $c_3 = 5$ ms

T_p = mcm(10, 20, 30) = 60 ms
T_s = 10 ms = 60/6



- Las tareas se ejecutan siguiendo un ciclo principal que se repite:
  - El periodo del ciclo principal es T_p = mcm(t_i)
- El ciclo principal se subdivide en varios ciclos secundarios
  - En cada ciclo secundario se ejecutan en secuencia un grupo distinto de tareas.
  - El periodo del ciclo secundario, T_s, es un divisor entero del periodo del ciclo principal y es el que marca la frecuencia de interrupción del temporizador.

J.M. Mendias
2022

Sistemas cyclic executive

ejemplo de implementación (i)

```
...
void timer0_isr( void ) __attributte__ ((interrupt ("IRQ")));

void main( void )
{
  sys_init();
  timer0_open_ms( timer0_isr, 10, TIMER_INTERVAL );
  while( 1 )
  {
    task1(); task2();  ..... ciclo secundario
    sleep();
    task1(); task3();  ..... ciclo secundario
    sleep();
    task1(); task2();  ..... ciclo secundario
    sleep();
    task1(); task3();  ..... ciclo secundario
    sleep();
    task1(); task2();  ..... ciclo secundario
    sleep();
    task1();  ..... ciclo secundario
    sleep()
  }
}

void timer0_isr( void )
{
  I_ISPC = BIT_TIMER0;
}
```

ciclo principal

Sistemas cyclic executive

ejemplo de implementación (ii)

- Para generalizar la implementación:
  - Las tareas de cada ciclo secundario se encapsulan en un trabajo
  - Se crea un buffer circular de trabajos que se despachan ordenadamente

```
...
#define MAX_JOBS (6)
void (*pjobs[MAX_JOBS])(void) =
{
  jobA, jobB, jobA, jobB, jobA, jobC
};
...
void jobA( void )
{
  task1();
  task2();
}
void jobB( void )
{
  task1();
  task3();
}
void jobC( void )
{
  task1();
}
```

array estático de trabajos

aunque hay 6 ciclos secundarios solo hay 3 esquemas de ejecución distintos

```
void main( void )
{
  uint8 i = 0;

  sys_init();
  timer0_open_ms( timer0_isr, 10,
     TIMER_INTERVAL );
  while( 1 )
  {
    (*pjobs[i])();
    i = ( i==MAX_JOBS-1 ? 0 : i+1 );
    sleep();
  }
}

void timer0_isr( void )
{
  I_ISPC = BIT_TIMER0;
}
```

despacha trabajo

Planificador cooperativo

introducción

- Planificador cooperativo (sin expropiación) elemental:
  - En todo momento sólo existe una única tarea en ejecución.
  - Las tareas se ejecutan hasta completarse sin ser interrumpidas por otras.
  - Las tareas se ejecutan con periodicidad fija.
    - Cuando llega el momento de ejecutar una tarea se encola en una lista de tareas "preparadas".
    - Cuando la CPU está libre, ejecuta las tareas "preparadas" en un orden fijo preestablecido.
  - El temporizador que dispara la ejecución de las tareas es la única fuente de interrupción
  - Si es inevitable la existencia de otras fuentes de interrupción:
    - Las RTI deberán realizar funciones complementarias simples (señalización, buffering...).
    - Será necesario hacer un análisis temporal detallado del peor caso y desarrollar mecanismos de exclusión mutua para el acceso a recursos compartidos entre las tareas y las RTI.
    - Si el modelo se complica: plantearse el uso de un micro-kernel (expropiativo) de tiempo real.
  - Propiedades:
    - Simple: el planificador es parte de la aplicación y se escribe en C con poca sobrecarga.
    - La ejecución de tareas es predecible, segura y fiable.
    - Sólo se asigna memoria para la tarea en ejecución.
    - Las tareas hacen un uso cooperativo de los recursos (no existen secciones críticas)
    - Las tareas poco prioritarias y la atención a eventos externos pueden sufrir altos retrasos.

fuente: PONT, Michael J. *Patterns for triggered embedded systems.* Addisson-Wesley, 2001

J.M. Mendias
2022

Planificador cooperativo

bloque de control de tarea

- En este modelo, cada tarea está definida:
  - La función que realiza
  - La periodicidad con que se ejecuta (medida en "ticks" de reloj)
  - El tiempo transcurrido desde su última ejecución (medido en "ticks" de reloj)
- Las periodicidades de las tareas determinan la resolución del "tick"
  - Debe haber una escala de tiempo común = MCD de los periodos.
- Las tarea pueden estar en uno de los dos estados:
  - No preparada: el tiempo transcurrido desde su última ejecución es inferior a su periodo.
  - Preparada: ha transcurrido un periodo desde su última ejecución, por lo que está a la espera de tomar la CPU para ser ejecutada.

TCB – Task Control Block

```
typedef struct
{
  void (* pfunction) (void);   ..... Función realizada por la tarea
  uint32 period;               ..... Periodo de tiempo entre 2 ejecuciones consecutivas
  uint32 ticks;                ..... Tiempo transcurrido desde la última ejecución
  boolean ready;               ..... Estado de la tarea
} task_t;

#define MAX_TASKS   (10)
task_t tasks[MAX_TASKS];       ..... Lista de TCBs. El índice indica la prioridad de la tarea:
                                     prioridad( tasks[i] ) > prioridad( task[i+1] )
```

J.M. Mendias
2022

Planificador cooperativo

creación y destrucción de tareas

```
void scheduler_init( void )
{
  uint32 id;

  for( id=0; id<MAX_TASKS; id++ )
    delete_task( id );
};

void delete_task( uint32 id )
{
  tasks[id].pfunction = NULL;
  tasks[id].period = 0;
  tasks[id].ticks = 0;
  tasks[id].ready = FALSE;
}

uint32 create_task( void (*pfunction)(void), uint32 period )
{
  uint32 id;

  for( id=0; id<MAX_TASKS && tasks[id].pfunction; id++ );

  tasks[id].pfunction = pfunction;
  tasks[id].period = period;
  tasks[id].ticks = 0;
  tasks[id].ready = FALSE;

  return id;
}
```

Busca la primera entrada libre de la lista de TCBs

J.M. Mendías
2022

Planificador cooperativo

planificación y despacho

```
void scheduler( void )
{
  uint32 id;

  I_ISPC = BIT_TIMER0;

  for( i=0; i<MAX_TASKS; i++ )
    if( tasks[id].pfunction )
      if( ++tasks[id].ticks == tasks[id].period )
      {
        tasks[id].ticks = 0;
        tasks[id].ready = TRUE;
      };
};
```

Recorre el array de bloques de control de tarea, incrementando el contador de ticks de cada una y pasando a "preparadas" aquellas que deben ejecutarse

El planificador es la RTI por fin de cuenta del temporizador: se ejecuta cada vez que este interrumpe (cada vez que hay un tick)

Si el contador de ticks de la tarea equivale a su periodo de ejecución:
- inicializa a 0 su contador de ticks
- pasa la tarea a estado "preparado"

```
void dispacher( void )
{
  uint32 id;

  for( id=0; id<MAX_TASKS; id++ )
    if( tasks[id].ready == TRUE )
    {
      (*tasks[id].pfunction)();
      tasks[id].ready = FALSE;
    }
};
```

Recorre el array de bloques de control de tareas para ejecutar en orden todas las tareas preparadas.

Si la tarea está en estado "preparado":
- la ejecuta
- pasa la tarea a estado "no preparado"

Planificador cooperativo

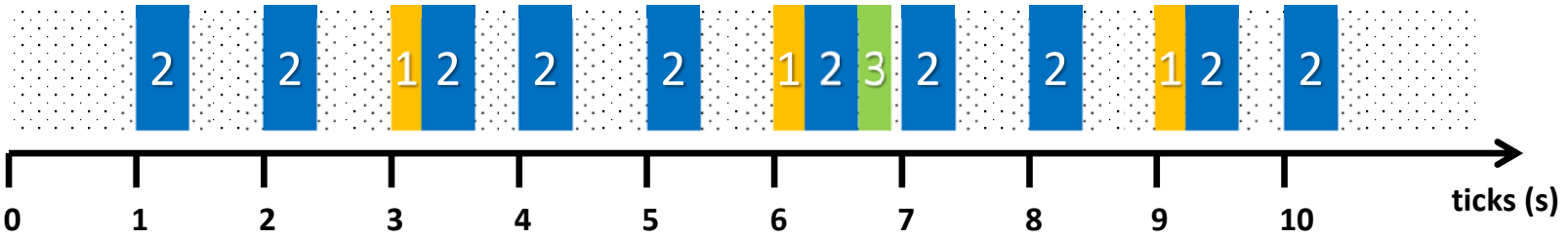
poniéndolo todo junto

```
...
void scheduler( void ) __attributte__ ((interrupt ("IRQ")));
void timer0_open_tick( void (*isr)(), unit16 tps );

void main( void )
{
  sys_init();

  scheduler_init();
  create_task( task1, 3 );
  create_task( task2, 1 );
  create_task( task3, 6 );

  task1_init();
  task2_init();
  task3_init();

  timer0_open_tick( scheduler, TICKSxSEC );
  while( 1 )
  {
    sleep();
    dispacher();
  }
}
```

La prioridad de las tareas es: task1 > task2 > task3.

El planificador se instala como RTI del timer0

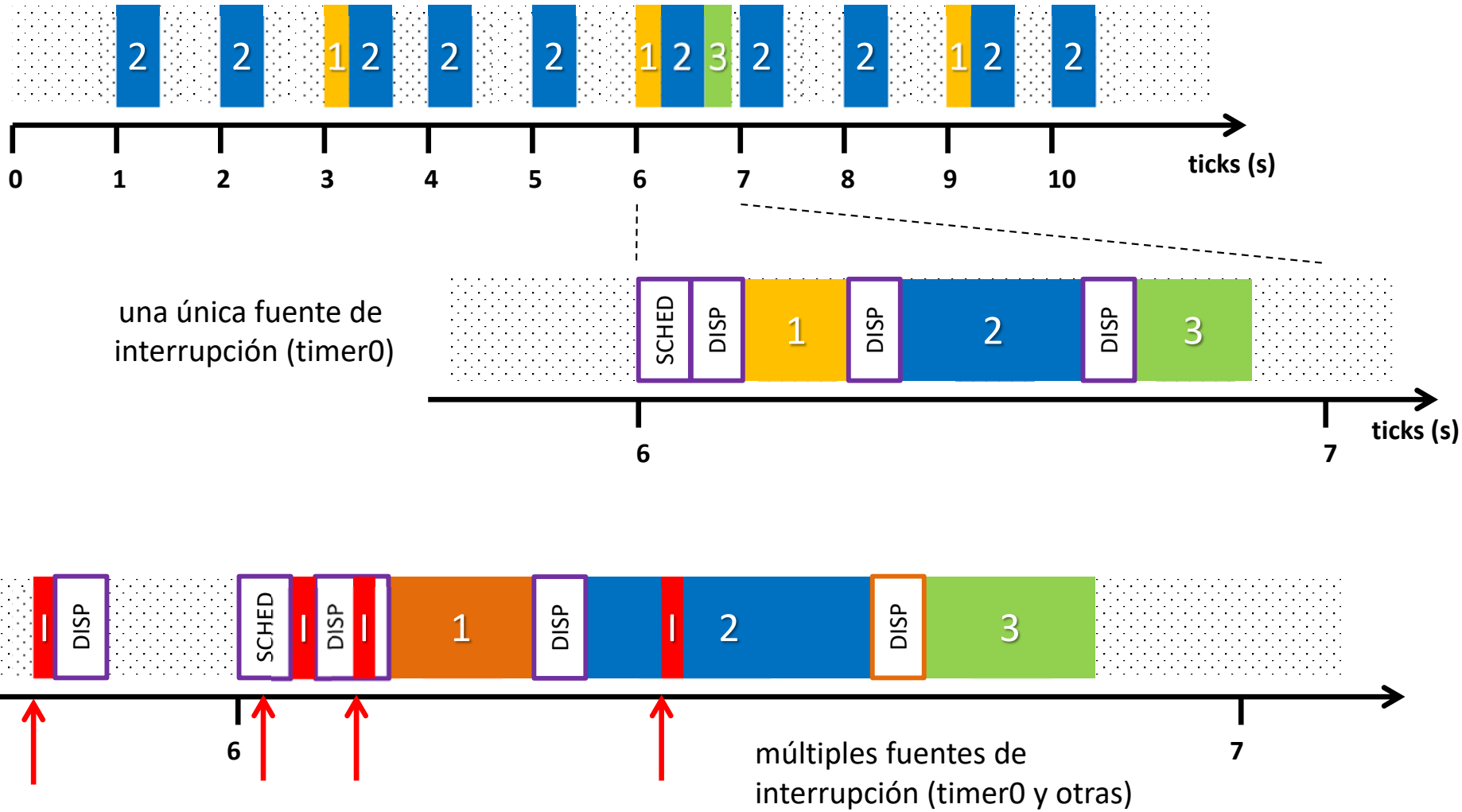
Las tareas se ejecutan fuera de la RTI



J.M. Mendías
2022

Planificador cooperativo

poniéndolo todo junto



Planificador cooperativo

variaciones (i)

- El modelo de tarea puede hacerse más completo incluyendo:
  - El número de veces máximo que debe ejecutarse cada tarea:

```
typedef struct
{
  void (* pfunction) (void);
  uint32 period;
  uint32 ticks;
  uint32 times;
  boolean ready;
} task_t;
```

```
void dispacher( void )
{
  ...
  if( tasks[id].ready == TRUE )
  {
    (*tasks[id].pfunction)();
    tasks[id].ready = FALSE;
    if( --task[id] == 0 )
      delete_task( id );
  }
};
```

  - La latencia (offset) de la primera iniciación:

```
uint32 create_task( void (*pfunction)(void), uint32 period, uint32 offset )
{
  ...
  tasks[id].ticks = period - offset;
  ...
}
```

Planificador cooperativo

variaciones (ii)

- La inicialización de la tarea durante su creación:

```
uint32 create_task( ... )
{
  ...
  tasks[id].pfunction = pfunction;
  tasks[id].period = period;
  tasks[id].ticks = 0;
  tasks[id].ready = FALSE;
  (*pfunction)();
  ...
}
```

```
void task( void )
{
  static boolean init = TRUE;

  if( init )
  {
     task_init();
     init = FALSE;
  }
  else
     ... el cuerpo de la tarea ...
}
```

- Solventar problemas puntuales de underrun:

```
void scheduler( void )
{
  ...
  if( ++tasks[id].ticks == tasks[id].period )
  {
    tasks[id].ticks = 0;
    tasks[id].ready++;
  };
};
```

```
void dispacher( void )
{
  ...
  if( tasks[id].ready )
  {
    (*tasks[id].pfunction)();
    tasks[id].ready--;
  }
};
```

J.M. Mendías
2022

Planificador cooperativo

variaciones (iii)

- La instrumentación de las tareas (medida de tiempos de ejecución max/min) :

```
typedef struct
{
  void (* pfunction) (void);
  uint32 period;
  uint32 ticks;
  boolean ready;
  uint32 WCET;
  uint32 BCET;
} task_t;
```

```
void dispacher( void )
{
  uint32 id;
  uint32 time;

  for( id=0; id<MAX_TASKS; id++ )
    if( tasks[id].ready == TRUE )
    {
      timer3_start();
      (*tasks[id].pfunction)();
      time = timer3_stop();
      if( time < tasks[id].BCET )
        tasks[id].BCET = time;
      if( time > tasks[id].WCET )
        tasks[id].WCET = time;
      tasks[id].ready = FALSE;
    }
};
```

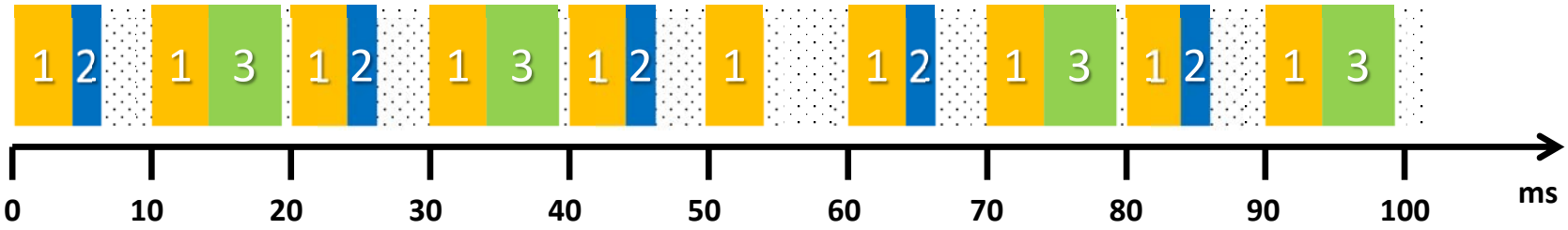
J.M. Mendías 2022

Planificador cooperativo

comparativa

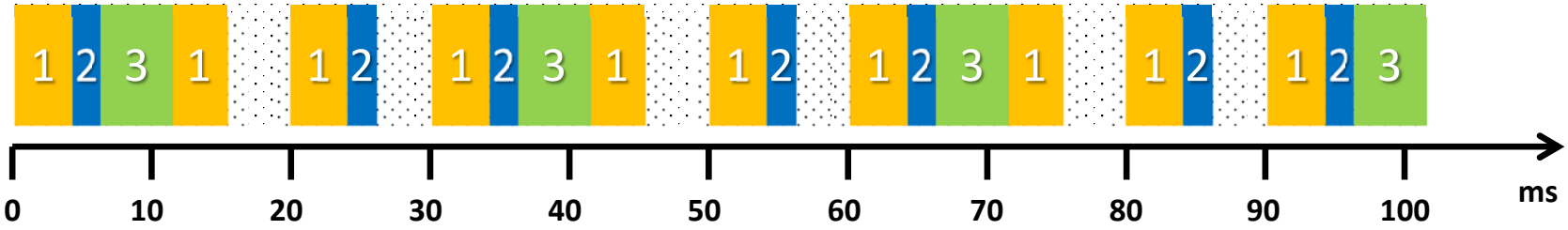
- Planificación estática:

tarea 1: $t_1 = 10$ *ms*, $c_2 = 4$ *ms*
tarea 2: $t_2 = 20$ *ms*, $c_2 = 2$ *ms*
tarea 3: $t_3 = 30$ *ms*, $c_3 = 5$ *ms*



- Planificación dinámica:

Resolución del tick: **10 ms**
La prioridad de las tareas es: **tarea1** > **tarea2** > **tarea3**



Planificador híbrido

introducción

- Planificador híbrido (con una única tarea expropiativa) elemental:
  - Soporta cualquier número de tareas cooperativas, pero sólo una expropiativa.
  - En todo momento sólo existen un máximo de 2 tareas en ejecución.
  - Las tareas cooperativas se ejecutan hasta completarse pudiendo ser interrumpidas únicamente por la tarea expropiativa.
  - Las tareas se ejecutan con periodicidad fija.
    - Cuando llega el momento de ejecutar una tarea expropiativa se ejecuta inmediatamente
    - Cuando llega el momento de ejecutar una tarea cooperativa se encola en una lista de tareas "preparadas".
    - Cuando la CPU está libre, ejecuta las tareas "preparadas" en un orden fijo preestablecido.
  - El temporizador que dispara la ejecución de las tareas es la única fuente de interrupción
  - Propiedades:
    - Simple: el planificador es parte de la aplicación y se escribe en C con poca sobrecarga.
    - La ejecución de tareas es predecible, segura y fiable.
    - Sólo se asigna memoria como máximo para las dos tareas en ejecución.
    - La compartición de recursos entre las tareas cooperativas y la expropiativa tendrá que ser mutuamente exclusiva.
    - La respuesta a eventos externos puede ser rápida.

fuente: PONT, Michael J. *Patterns for triggered embedded systems.* Addisson-Wesley, 2001

Planificador híbrido

implementación

```
#define COOPERATIVE_TASK   (0)
#define PREEMPTIVE_TASK    (1)
```

Tipos de tarea

TCB – Task Control Block

```
typedef struct
{
  void (* pfunction) (void);
  uint32 period;
  uint32 ticks;
  boolean ready;
  uint8 type;
} task_t;
```

Tipo de la tarea: **sólo una** puede ser expropiativa

```
void delete_task( uint32 id )
{
  ...
  tasks[id].type = 0;
  ...
}

uint32 create_task( void (*pfunction)(void), uint32 period, uint8 type )
{
  ...
  tasks[id].type = type;
  ...
}
```

Planificador híbrido

implementación

```
void scheduler( void )
{
  uint32 id;

  I_ISPC = BIT_TIMER0;

  for( i=0; i<MAX_TASKS; i++ )
    if( tasks[id].pfunction )
      if( ++tasks[id].ticks == tasks[id].period )
      {
        tasks[id].ticks = 0;
        if( tasks[id].type == COOPERATIVE_TASK )
          tasks[id].ready = TRUE;
        else
          (*tasks[id].pfunction)();
      };
};
```

Si el contador de ticks de la tarea equivale a su periodo de ejecución:

- inicializa a 0 su contador de ticks
- si la tarea es cooperativa: la pasa a estado "preparado"
- si la tarea es expropiativa: la ejecuta

- ¿Cómo y donde se ejecutan las tareas?
  - Las cooperativas son ejecutadas en orden por el dispacher como parte del programa principal.
  - La expropiativa es ejecutada por el scheduler dentro de la RTI del temporizador.

J.M. Mendías
2022

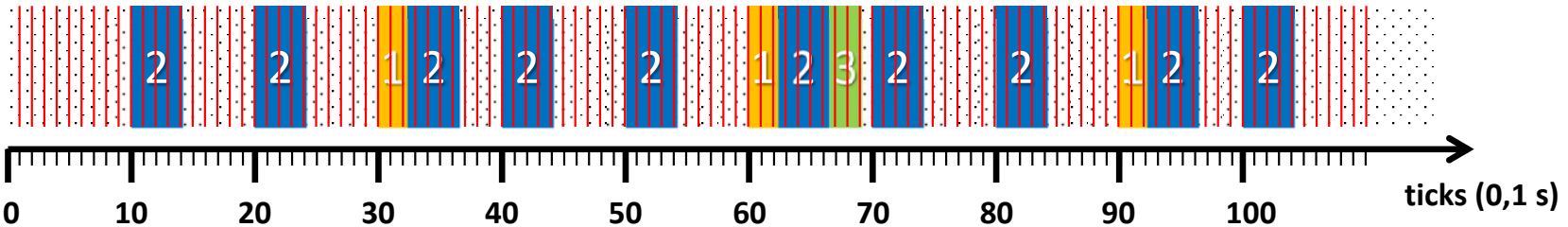
Planificador híbrido

poniéndolo todo junto

```
...
void scheduler( void ) __attributte__ ((interrupt ("IRQ")));
void timer_open( void (*isr)(), unit16 ms );

void main( void )
{
  sys_init();

  scheduler_init();
  create_task( ptask, 1, PREEMPTIVE_TASK );
  create_task( task1, 30, COOPERATIVE_TASK );
  create_task( task2, 10, COOPERATIVE_TASK );
  create_task( task3, 60, COOPERATIVE_TASK );

  ptask_init();
  task1_init();
  task2_init();
  task3_init();

  timer0_open_tick( scheduler, 10 );
  while( 1 )
  {
    sleep();
    dispacher();
  }
}
```

La tarea expropiativa tiene una frecuencia de ejecución mayor

La prioridad de las tareas cooperativas es: task1 > task2 > task3.



Sistemas multiestado

introducción

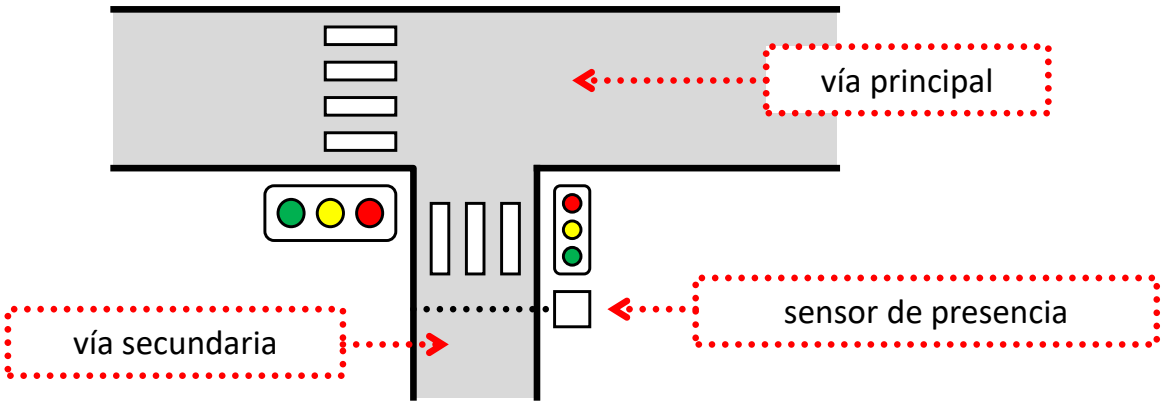
- Es muy común que un sistema empotrado realice indefinidamente una secuencia finita de acciones con cierta temporización y según la ocurrencia de eventos externos.
  - Estos sistemas se modelan mediante 1 o varias FSM cooperativas.

- Sistema multiestado dirigido por tiempos:
  - La FSM cambia de estado únicamente por el paso del tiempo
    - Que puede definirse en valor absoluto o en valor relativo a un tick periódico.
- Sistema multiestado dirigido por tiempos y eventos externos:
  - La FSM cambia de estado tanto por el paso del tiempo como la ocurrencia de eventos externos o de lectura de valores de entradas.
- Sistema multiestado dirigido por eventos externos:
  - No muy común, ya que por seguridad estos sistemas siempre tienen alguna dependencia con el tiempo en forma de timeouts.

Sistemas multiestado

dirigidos por tiempo y eventos: ejemplo

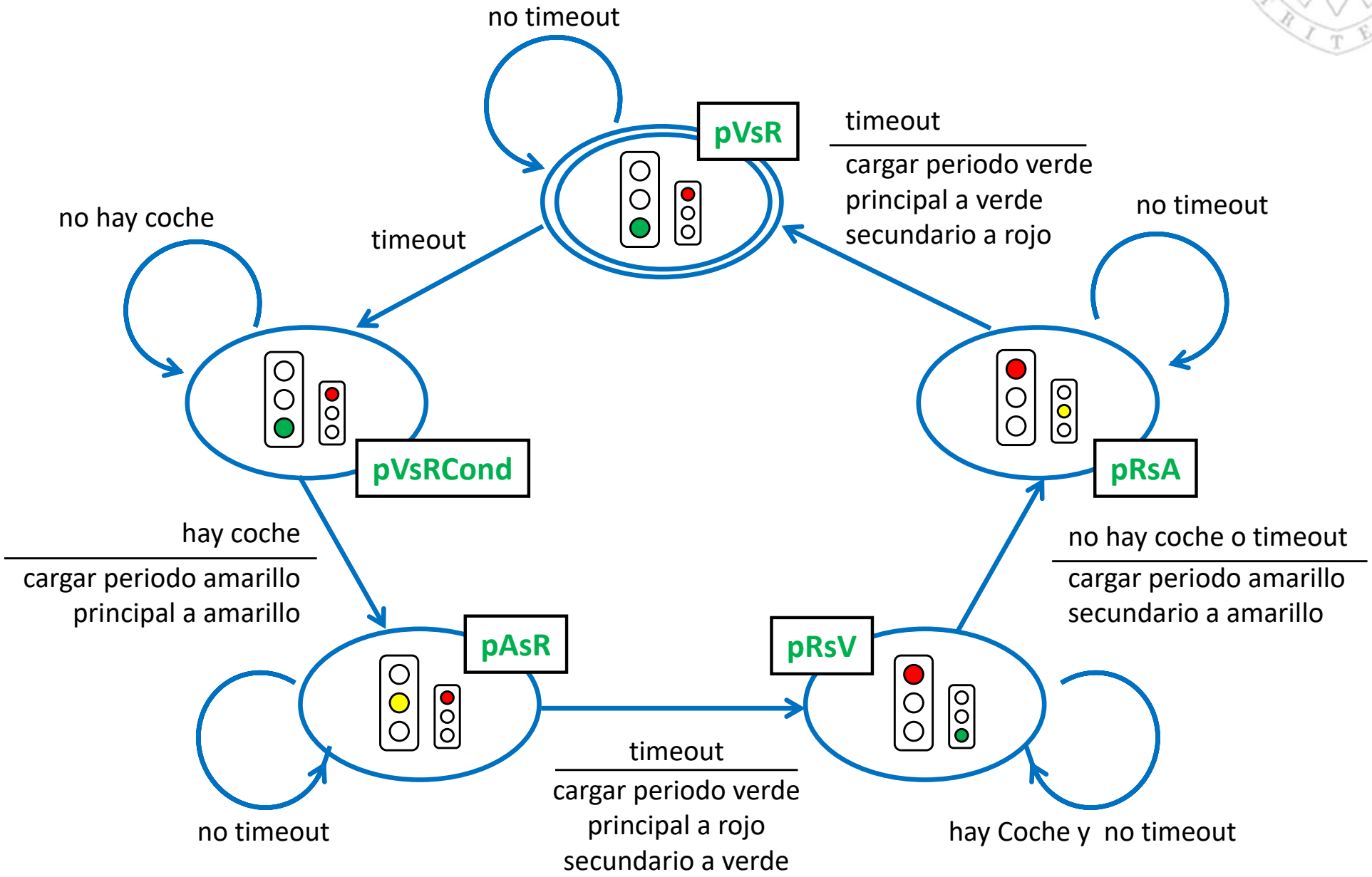
- Se desea desarrollar un sistema que controle los semáforos que hay en el cruce entre una carretera principal y otra secundaria:
  - El semáforo principal estará verde como mínimo un periodo verde y continuará en verde hasta que no se detecten coches en la vía secundaria.
  - Si hubiera coches en la vía secundaria, sólo si el semáforo de coches ha estado en verde durante un periodo verde completo, el semáforo principal pasará a amarillo durante un periodo amarillo, tras el cual se pondrá en rojo. Entonces el semáforo secundario pasará a verde.
  - El semáforo secundario estará verde como máximo un periodo verde tras el cual pasará a amarillo durante un periodo amarillo, y finalizado éste, pasará a rojo. Entonces el semáforo principal pasará a verde.
  - No obstante si la vía secundaria se quedara vacía antes de consumir el periodo verde, se disparará la transición de los semáforos.



J.M. Mendías
2022

Sistemas multiestado

dirigidos por tiempo y eventos: FSM



Sistemas multiestado

dirigidos por tiempo (absoluto) y eventos: por pooling (i)

```
#define PER_VERDE    (...)
#define PER_AMARILLO (...)
```
Duraciones absolutas del periodo de los semáforos (en unidades de tiempo)

```
typedef enum {pVsR, pVsRCond, pAsR, pRsV, pRsA} state_t;
```
Definición de estados de la FSM

```
state_t state;
```
Estado actual de la FSM

```
void semaforos_init( void );
void semaforos( void );

void main( void )
{
  sys_init();
  semaforos_init();

  while( 1 )
    semaforos();
}

void semaforos_init( void )
{
  timers_init();
  sensor_init();
  ligths_init();
  state := pVsR;
  light_on( PRINCIPAL, VERDE );
  light_on( SECUNDARIO, ROJO );
  timer3_start_timeout( PER_VERDE );
}
```

Sistemas multiestado

dirigidos por tiempo (absoluto) y eventos: por pooling (ii)

```
void semaforos( void )
{
  switch( state )
  {
    case pVsR :
      if( timer3_timeout() )
        state = pVsRCond;
      break;
    case pVsRCond :
      if( sensor_status() )
      {
        state = pAsR;
        light_off( PRINCIPAL, VERDE );
        light_on( PRINCIPAL, AMARILLO );
        timer3_start_timeout( PER_AMARILLO );
      }
      break;
    case pAsR :
      if( timer3_timeout() )
      {
        state = pRsV;
        light_off( PRINCIPAL, AMARILLO );
        light_on( PRINCIPAL, ROJO );
        light_off( SECUNDARIO, ROJO );
        light_on( SECUNDARIO, VERDE );
        timer3_start_timeout( PER_VERDE );
      }
      break;
    ...
    ...
    case pRsV :
      if( !sensor_status() || timer3_timeout() )
      {
        state = pRsA;
        light_off( SECUNDARIO, VERDE );
        light_on( SECUNDARIO, AMARILLO );
        timer3_start_timeout( PER_AMARILLO );
      }
      break;
    case pRsA :
      if( timer3_timeout() )
      {
        state = pVsR;
        light_off( PRINCIPAL, ROJO );
        light_on( PRINCIPAL, VERDE );
        light_off( SECUNDARIO, AMARILLO );
        light_on( SECUNDARIO, ROJO );
        timer3_start_timeout( PER_VERDE );
      }
      break;
  };
}
```

J.M. Mendías
2022

Sistemas multiestado

dirigidos por tiempo (absoluto) y eventos: por pooling (iii)

```
void semaforos( void )
{
  switch( state ) {
    case pVsR :
      if( check_timeout() )
        state = pVsRCond;
      break;
    case pVsRCond :
      if( check_sensor() ) {
        state = pAsR;
        pAsR_actions(); }
      break;
    case pAsR :
      if( check_timeout() ) {
        state = pRsV;
        pRsV_actions(); }
      break;
    case pRsV :
      if( check_both() ) {
        state = pRsA;
        pRsA_actions(); }
      break;
    case pRsA :
      if( check_timeout() ) {
        state = pVsR;
        pVsR_actions(); }
      break;
  };
}
```

```
boolean check_timeout( void )
{
  return timer3_timeout();
};

boolean check_sensor( void )
{
  return sensor_status();
};
...
```

```
void pAsR_actions( void )
{
  light_off( PRINCIPAL, VERDE );
  light_on( PRINCIPAL, AMARILLO );
  timer3_start_timeout( PER_AMARILLO );
}

void pRsV_actions( void )
{
  light_off( PRINCIPAL, AMARILLO );
  light_on( PRINCIPAL, ROJO );
  light_off( SECUNDARIO, ROJO );
  light_on( SECUNDARIO, VERDE );
  timer3_start_timeout( PER_VERDE );
}
...
```

Sistemas multiestado

dirigidos por tiempo (absoluto) y eventos: por interrupción

```
volatile boolean flag_timeout = FALSE;
...
void isr_timer3( void ) __attribute__ ((interrupt ("IRQ")));
...

void isr_timer3( void )
{
  flag_timeout = TRUE;  ..... Activa flag
  I_ISPC       = BIT_TIMER3;
}
...
```

```
boolean check_timeout( void )
{
  return flag_timeout;  ..... Devuelve el valor del flag
};
...
```

```
void pAsR_actions( void )
{
  light_off( PRINCIPAL, VERDE );
  light_on( PRINCIPAL, AMARILLO );
  flag_timeout = FALSE; ..... Desactiva flag
  timer3_open( isr_timer3, PER_AMARILLO, TIMER_ONE_SHOT );
}
...
```

Sistemas multiestado

dirigidos por tiempo (absoluto) y eventos: generalización (i)

```
typedef struct {
  uint16  src;                  ..... Estado inicial de la transición
  boolean (*pfChk)(void);       ..... Función a satisfacer para efectuar el cambio de estado
  uint16  dst;                  ..... Estado final de la transición
  void    (*pfAct)(void);       ..... Función a realizar tras el cambio de estado
} fsmt_t;

typedef struct {
  uint16 state;                 ..... Estado actual de la FSM
  fsmt_t *ptt;                  ..... Tabla de transiciones de la FSM
  uint16 size;                  ..... Tamaño de la tabla de transiciones
  void   (*init)(void);         ..... Función de inicialización de la FSM
} fsm_t;
```

Independiente de la FSM concreta

```
void fsm_init( fsm_t *fsm )
{
  fsm->init();
}
void fsm_run( fsm_t *fsm )
{
  uint16 i;
  fsmt_t *t;

  for( i=0, t=fsm->ptt; i<fsm->size; t++, i++ )  ..... Recorre la tabla de transiciones
    if ( fsm->state == t->src )                  ..... Si el estado actual es igual al estado inicial de la transición
      if( t->pfChk() ) {                         ..... y se satisface la función que habilita el cambio de estado
        fsm->state = t->dst;                     ..... Cambia al estado final de la transición
        if( t->pfAct )
          t->pfAct();                            ..... y ejecuta la correspondiente función en caso de estar definida
      }
}
```

Independiente de la FSM concreta

Sistemas multiestado

dirigidos por tiempo (absoluto) y eventos: generalización (ii)

```
#define PER_VERDE    (...)
#define PER_AMARILLO (...)

typedef enum {pVsR, pVsRCond, pAsR, pRsV, pRsA} state_t;

boolean check_timeout( void );
...
void pAsR_actions( void );
...

fsmt_t semaforos_tt[] = {
  { pVsR,     check_timeout, pVsRCond, NULL          },
  { pVsRCond, check_sensor,  pAsR,     pAsR_actions },
  { pAsR,     check_timeout, pRsV,     pRsV_actions },
  { pRsV,     check_both,    pRsA,     pRsA_actions },
  { pRsA,     check_timeout, pVsR,     pVsR_actions }
};

fsm_t semaforos = { pVsR, semaforos_tt, 5, semaforos_init };

void main( void )
{
  sys_init();
  fsm_init( &semaforos );

  while( 1 )
    fsm_run( &semaforos );
}
```

Tabla de transición del sistema

FSM

Específico de la FSM concreta

Sistemas multiestado

dirigidos por tiempo (relativo) y eventos: generalización (i)

```
...
typedef struct {
  uint16 state;
  uint16 *pticks;          ..... Tabla de duraciones (en ticks) de cada estado de la FSM
  fsmt_t *ptt;
  uint16 size;
  void   (*init)(void);
} fsm_t;

volatile uint16 ticks;     ..... Contador de ticks que quedan para una transición por tiempo
```

```
void fsm_run( fsm_t *fsm )
{
  uint16 i;
  fsmt_t *t;

  if( !ticks )  ..... Chequea si a transcurrido la duración
    for( i=0, t=fsm->ptt; i<fsm->size; t++, i++ )  ..... Recorre la tabla de transiciones
      if( fsm->state == t->src )
        if( !t->pfChk ) {  ..... Chequea si el cambio de estado es por tiempo
          fsm->state = t->dst;
          ticks = fsm->ticks[state];  ..... Actualiza el contador de ticks
          if( t->pfAct )
            t->pfAct();
        } else if( t->pfChk() ) {  ..... Chequea si el cambio de estado es por evento
          fsm->state = t->dst;
          ticks = fsm->ticks[state];
          if( t->pfAct )
            t->pfAct();
        }
}
```

J.M. Mendías
2022

Sistemas multiestado

dirigidos por tiempo (relativo) y eventos: generalización (ii)

```
...
void timer0_isr( void ) __attributte__ ((interrupt ("IRQ")));

uint16 semamoros_ticks = { PER_VERDE, 0, PER_AMARILLO, PER_VERDE, PER_AMARILLO }

fsmt_t semaforos_tt[] = {
  { pVsR,     NULL,         pVsRCond, NULL         },
  { pVsRCond, check_sensor, pAsR,     pAsR_actions },
  { pAsR,     NULL,         pRsV,     pRsV_actions },
  { pRsV,     check_sensor, pRsA,     pRsA_actions },
  { pRsA,     NULL,         pVsR,     pVsR_actions }
};

fsm_t semaforos = { pVsR, semaforos_ticks, semaforos_tt, 5, semaforos_init };

void main( void )
{
  sys_init();
  fsm_init( &semaforos );
  timer0_open_tick( timer0_isr, 1 );
  while( 1 )
 {
    sleep();
    fsm_run( &semaforos );
  }
}

void timer0_isr( void )
{
  I_ISPC = BIT_TIMER0;
  ticks = ticks ? ticks-1 : 0;
}
```

Planificador cooperativo

de múltiples tareas multiestado dirigidas por tiempo

```
typedef struct
{
  void (* pfsm) (uint32 *, uint32 *);  ..... FSM
  uint32 state;  ..... Estado actual de la FSM
  uint32 period;  ..... Duración del estado actual (en ticks)
  uint32 ticks;  ..... Tiempo transcurrido en el estado actual (en ticks)
  boolean ready;
} task_t;

#define MAX_TASKS   (10)
task_t tasks[MAX_TASKS];
```

```
void delete_task( uint32 id )
{
  ...
  tasks[id].state = 0;
  ...
}

uint32 create_task( void (*pfsm)(uint32 *, uint32 *), uint32 state, uint32 period )
{
  ...
  tasks[id].state = state;
  ...
}
```

J.M. Mendías
2022

Planificador cooperativo

de múltiples tareas multiestado dirigidas por tiempo

```
void dispacher( void )
{
  uint32 id;

  for( id=0; id<MAX_TASKS; id++ )
    if( tasks[id].ready == TRUE )
    {
      (*tasks[id].pfsm)( &tasks[id].state, &tasks[id].period );
      tasks[id].ready = FALSE;
    }
};
```

```
void task_fsmN( uint32 *state, uint32 *period )
{
  switch( *state )
  {
    case state0:
      ... funciones ...
      *state = ... siguiente estado ...;
      *period = ... siguiente periodo ...;
      break;
    case stateN:
      ..
      break;
  };
}
```

Se realiza en el último tick del estado actual.

El concepto de actualización del periodo sería aplicable a tareas con periodicidad variable.

Recuperación de bloqueos

gestión del watchdog (i)

- Un watchdog es un temporizador (descendente) que resetea el sistema (o genera una interrupción) tras agotar un cierto timeout programable.
  - Para evitar el reseteo, el software debe periódicamente reiniciar la cuenta del watchdog (kicking the dog).
  - Cada vez que el watchdog se reinicia, comienza a contar un nuevo timeout.

```
static uint16 timeout;          ..... Almacena localmente el timeout (en número de intervalos de 100 µs)

void wd_on( uint16 dms )
{
  timeout = dms;
  WTCNT = dms;                  ..... Carga inicialmente en el timeout en el registro de cuenta
  WTCON |= (99 << 8) | (1 << 5) | (2 << 3) | (1);   { Fija la resolución a 100 µs
                                                      Habilita la cuenta
                                                      Activa la generación de reset tras timeout
}

void wd_kick( void )
{
  WTCNT = timeout;              ..... Reinicia la cuenta , recargando el timeout
}
```

t_{WD} = (PRESCALER + 1) / (64 MHz / DIVISOR) = (99 + 1) / (64 MHz / 64) = 100 µs

Recuperación de bloqueos

gestión del watchdog (ii)

- Permite la recuperación del sistema frente a bloqueos imprevistos:
  - Un sistema vivo, no tendrá problemas en reiniciar el watchdog antes de cada timeout.
  - Un sistema bloqueado, no podrá reiniciar el watchdog. Cuando el timeout se alcance, el watchdog resetea el sistema sacándolo del bloqueo.
  - Para asegurar que el watchdog funciona, el sistema durante su inicialización puede dejar pasar el timeout para forzar un watchdog reset.
- Tras un watchdog reset el sistema puede
  - Iniciarse normalmente
  - Dormirse guardando su estado para posterior depuración off-line
  - Arrancar algún tipo de diagnosis y/o corrección de errores.
- Para distinguir un power-on reset de un watchdog reset:
  - Puede usarse un flag en RAM, analizar el valor de alguno de los registros, etc.
- Las técnicas para reiniciar el watchdog dependen de la arquitectura SW.

J.M. Mendías
2022

Recuperación de bloqueos

gestión del watchdog (iii)

- Arquitectura super-loop
  - El watchdog se reinicia en cada iteración del bucle.
    - Si la ejecución de alguna iteración dura más del timeout, no podrá reiniciarse el watchdog a tiempo y el sistema se resetea.
    - El timeout debe ser algo superior a la máxima duración posible (sin bloqueo) de una iteración.
- Arquitectura foreground/background
  - La hebra en bakground reinicia el wathdog en cada iteración del bucle.
    - Si la ejecución de alguna hebra en foreground dura más del timeout, la hebra en background no podrá reiniciar el watchdog a tiempo y el sistema se resetea.
    - El timeout debe ser algo superior a la duración de la hebra de mayor retardo (o a la suma de las duraciones del peor caso de ejecución superpuesta de varias hebras).

```
#define TIMEOUT ...

void main( void )
{
  wd_on( TIMEOUT );
  while( 1 )
  {
    ...E/S y procesamiento...
    wd_kick();
  }
}
```

```
void isr1( void )
{
 ...E/S y/o procesamiento...
}

void isr2( void )
{
  ...E/S y/o procesamiento...
}
...
```

J.M. Mendías
2022

Recuperación de bloqueos

gestión del watchdog (iv)

- Sistemas multitarea cooperativos basados en tiempo
  - Una tarea periódica (monitor) reinicia el watchdog un poco antes del fin del timeout.
    - Si alguna tarea dura más del timeout, no devolverá el control al planificador y por consiguiente el monitor no podrá reiniciar el watchdog a tiempo y el sistema se resetea.
    - El timeout debe ser ligeramente superior a la duración de la tarea de mayor retardo.

```
#define TIMEOUT ...

void main( void )
{
  ...
  scheduler_init();
  create_task( wd_kick, ... );
  ...crea tareas...  ...

  timer0_open_tick( scheduler, 1 );
  wd_on( TIMEOUT );
  while( 1 )
  {
    sleep();
    dispacher();
  }
}
```

La tarea monitor tiene máxima prioridad para que llegado su instante de tiempo sea despachada la primera

J.M. Mendias
2022

Recuperación de bloqueos

gestión del watchdog (v)

- Sistemas multitarea expropiativos
  - Cada tarea dispone de un flag para indicar su estado.
    - Tareas repetitivas (repiten indefinidamente un mismo cálculo): al finalizar cada iteración ponen su flag a ALIVE.
    - Tareas de espera (que esperan un evento que puede que no suceda) antes de cada espera ponen su flag a SLEEP y después a ALIVE.
  - Una tarea periódica (monitor) reinicia el watchdog un poco antes del fin del timeout solo si todos los flags están ALIVE/SLEEP. Tras el reinicio, pone los flags a UNKNOWN.
    - Si alguna tarea dura más del timeout, no modificará a tiempo su flag, el monitor no reiniciará el watchdog y el sistema se resetea.
    - El timeout debe ser algo superior a la duración de la tarea repetitiva de mayor retardo.

- Un esquema análogo se puede aplicar para monitorizar la frecuencia de ejecución relativa de las tareas.
  - Cada tarea dispone de un contador del número de veces que se ejecuta.
  - El monitor reinicia el wathdog solo si el número de veces que se ha ejecutado cada tarea esta dentro de los límites previstos. Tras el reinicio, pone a 0 las cuentas.

J.M. Mendias
2022

Recuperación de bloqueos

gestión del watchdog (vi)

```
enum { UNKNOWN, SLEEP, ALIVE }
  task_status[NUM_TASKS];

void main( void )
{
  ...crea tareas...
  wd_on( TIMEOUT );
  ...inicia RTOS...
}
```

```
void Task1( void )
{
  while( 1 )
  {
    ... procesamiento ...
    task_status[1] = ALIVE;
  }
}

void Task2( void )
{
  while( 1 )
  {
    task_status[2] = SLEEP;
    ...espera por evento...
    task_status[2] = ALIVE;
    ...procesamiento...
  }
}
```

```
void MonitorTask( void )
{
  uint8 i;
  boolean error;

  while( 1 )
  {
    error = FALSE;
    for( i=0; i<NUM_TASKS; i++ )
      if( task_status[i] == UNKNOWN )
        error = TRUE;

    if( error )
      ...reporta error...
    else
      wd_kick();

    for( i=0; i<NUM_TASKS; i++ )
      task_status[i] = UNKNOWN;

    ...espera por cierto tiempo...
  }
}
```

J.M. Mendias
2022

Recuperación de bloqueos

gestión del watchdog (vii)

```
uint8 runs[NUM_TASKS];
const uint8 min_runs[NUM_TASKS];
const uint8 max_runs[NUM_TASKS];

void main( void )
{
  ...crea tareas...
  wd_on( TIMEOUT );
  ...inicia RTOS...
}
```

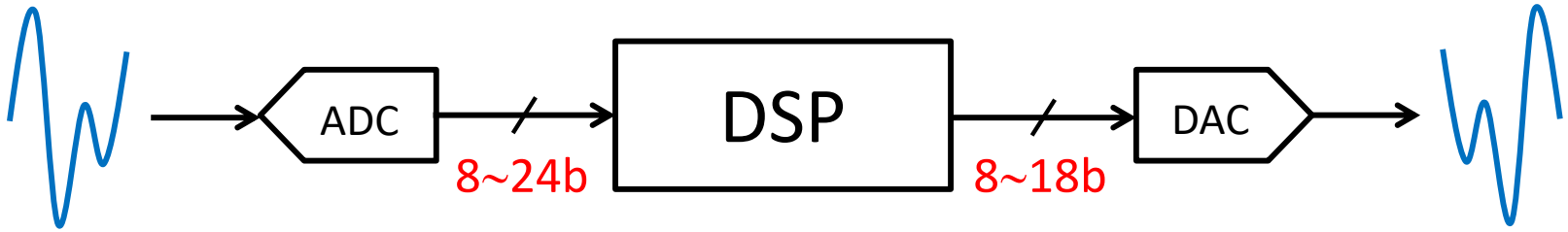
```
void Task1( void )
{
  while( 1 )
  {
    ... procesamiento ...
    runs[1]++;
  }
}

void Task2( void )
{
  while( 1 )
  {
    ... procesamiento ...
    runs[2]++;
  }
}
```

```
void MonitorTask( void )
{
  uint8 i;
  boolean error;

  while( 1 )
  {
    error = FALSE;
    for( i=0; i<NUM_TASKS; i++ )
      if( runs[i] < min_runs[i] ||
          runs[i] > max_runs[i] )
        error = TRUE;

    if( error )
      ...reporta error...
    else
      wd_kick();

    for( i=0; i<NUM_TASKS; i++ )
      runs[i] = 0;

    ...espera por cierto tiempo...
  }
}
```

Sistemas muestreados

introducción

- Sistemas que leen, procesan y escriben muestras discretas de señales analógicas.
  - Típicamente utilizan conversores analógico-digital (ADC) y/o digital-analógico (DAC).
  - Todo el procesamiento se realiza sobre las valores digitales muestreados.

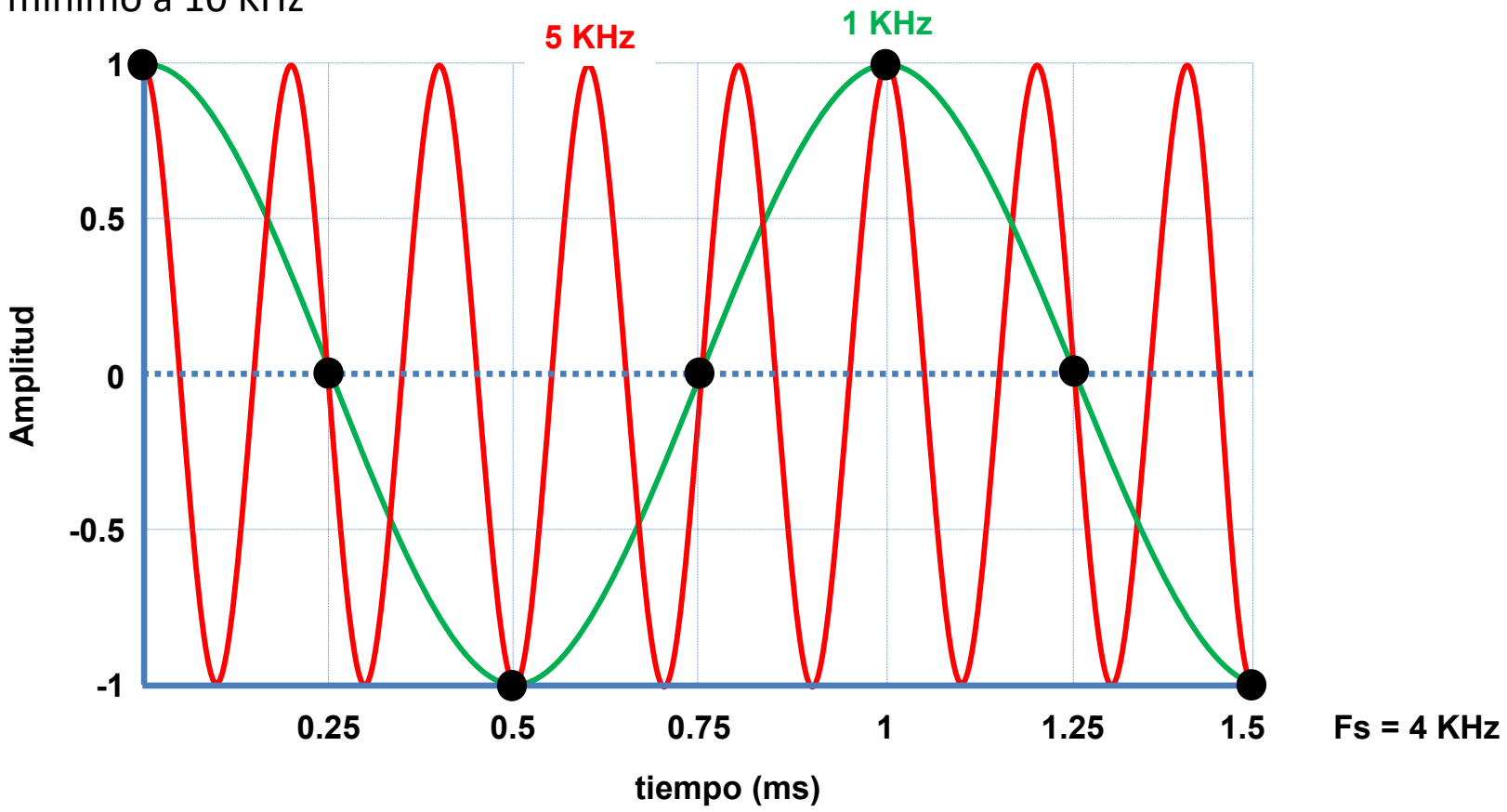


- Las muestras se toman esporádicamente o a una periodicidad fija:
  - El periodo/frecuencia de toma de muestras se llama periodo/frecuencia de muestreo.
    - Según el teorema de Nyquist la frecuencia de muestreo debe ser como mínimo el doble de la componente de frecuencia más alta que tenga una señal ($f_s > 2f$).
    - Para evitar aliasing en el camino de entrada existe algún tipo de filtro que elimina las frecuencias por encima de la mitad de la frecuencia de muestreo.
  - Por ejemplo, dado que el oído no puede detectar frecuencias superiores a 20 KHz (la voz humana entre 70-1000 Hz), los sistemas de audio muestreado:
    - Muestrean a frecuencias superiores a 40 KHz (típicamente 44.1KHz o 48 KHz)
    - A la entrada tienen un filtro paso baja con una frecuencia de corte de 20 KHz.

Sistemas muestreados

aliasing

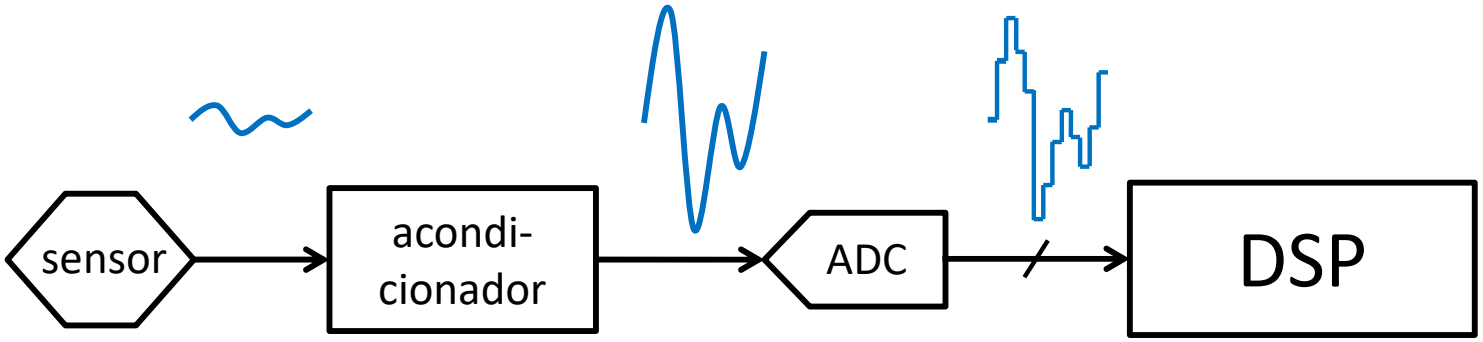
- Dos señales de 1 KHz y 5 KHz en fase muestreadas a 4 KHz son indistinguibles
  - Generan la secuencia de muestras (1, 0, -1, 0, 1, 0, -1 ...)
  - Si queremos distinguir hasta frecuencias de 5 KHz, el muestreo debe hacerse como mínimo a 10 KHz



Sistemas muestreados

sensores y acondicionadores (i)

- Típicamente se muestrea un voltaje analógico proveniente de un sensor cuya salida ha sido acondicionada.
  - Los sensores convierten una magnitud física (temperatura, luminosidad...) en una magnitud eléctrica (voltaje, resistencia, capacidad o intensidad)
    - Típicamente de baja amplitud y con cierto nivel de ruido.
    - Por ejemplo, la resistencia de un termistor NTC varía con la temperatura
  - El acondicionador convierte a voltaje, aísla, amplifica, filtra, lineariza y/o demodula la salida del sensor para que pueda ser convertida en digital
    - Por ejemplo, un termistor NTC se conecta a un divisor de tensión y a un amplificador.

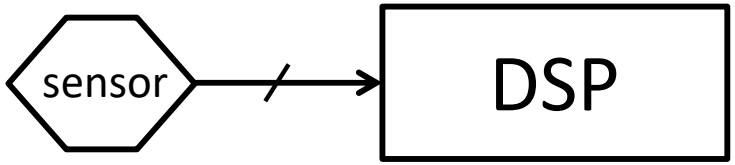


- En ocasiones, parte del acondicionamiento se realiza por SW en el DSP
  - Evaluando una ecuación característica, interpolando, accediendo a una tabla...

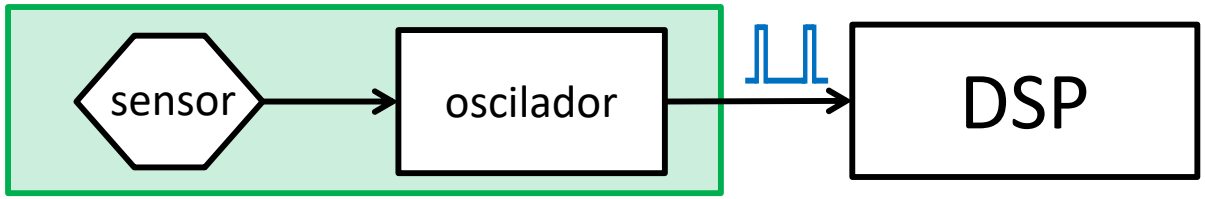
Sistemas muestreados

sensores y acondicionadores (ii)

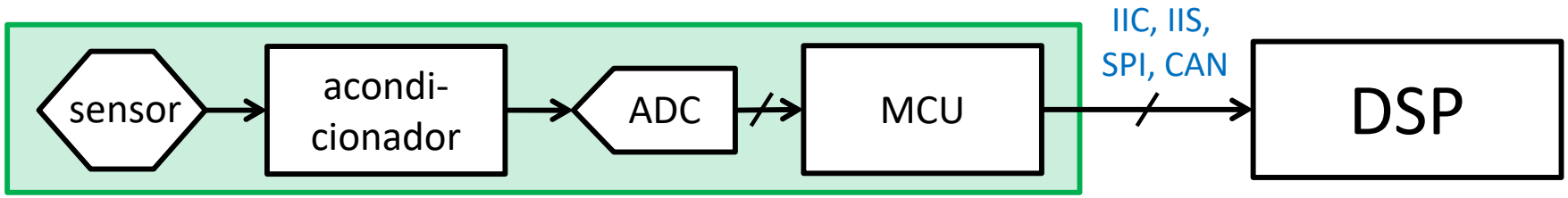
- Alternativamente existen otros mecanismos de captura de muestras:
  - Sensores digitales: generan directamente un código digital (codificadores posición)



  - Sensores "casi digitales": convierten la señal analógica en una señal con una modulación de alguno de sus parámetro temporales
    - Frecuencia, ciclo de trabajo o intervalo de tiempo entre pulsos



  - Sensores integrados: comunican las muestras usando un protocolo digital estándar.



Sistemas muestreados

conversores analógico-digital (i)

- Existe una gran variedad de conversores analógico-digitales que pueden clasificarse según distintos parámetros.

- Según el número de terminales analógicos, un ADC puede ser:
  - Single-ended: digitaliza el nivel de voltaje de un único terminal (V_{in}) respecto a tierra
  - Diferencial: digitaliza la diferencia de voltaje entre 2 terminales ($V_{in+} - V_{in-}$)
    - Completamente diferencial: ambos terminales aceptan señales arbitrarias.
    - Pseudo-diferencial: el terminal V_{in-} toma un voltaje fijo (típicamente GND o $V_{ref}/2$)

- Según el rango de voltaje de las señales analógicas, un ADC puede ser:
  - Unipolar: la señal de entrada toma valores comprendidos entre tierra y V_{ref}
  - Bipolar: la señal de entrada toma valores comprendidos entre $-V_{ref}$ y $+V_{ref}$

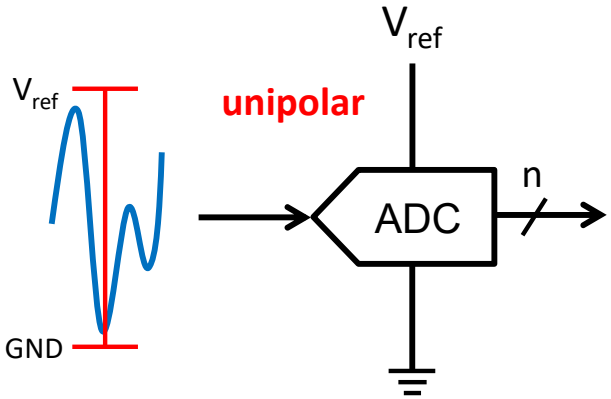
- La señal digital de n bits obtenida será, según el caso, un entero:
  - Sin signo: codificado en binario o binario inverso.
  - Con signo: codificado en C2, C2 inverso, biased (exceso 2^{n-1}) o biased inverso.

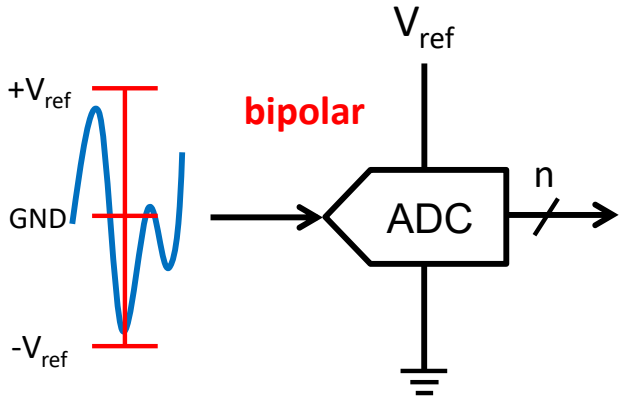
J.M. Mendías
2022

Sistemas muestreados

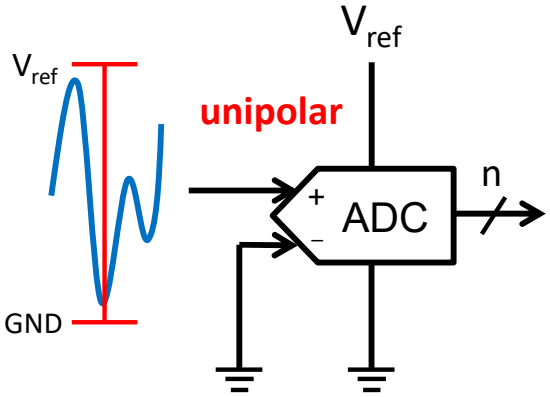
conversores analógico-digital (ii)

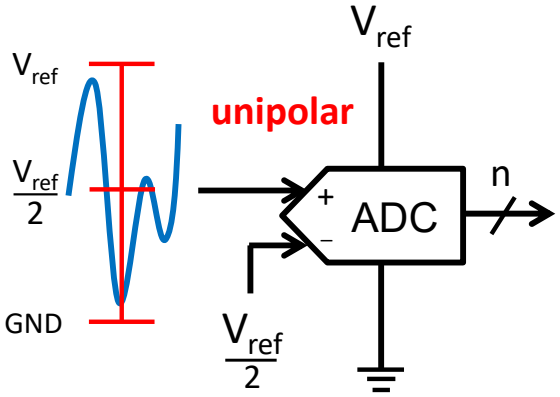
- Conversores single-ended:

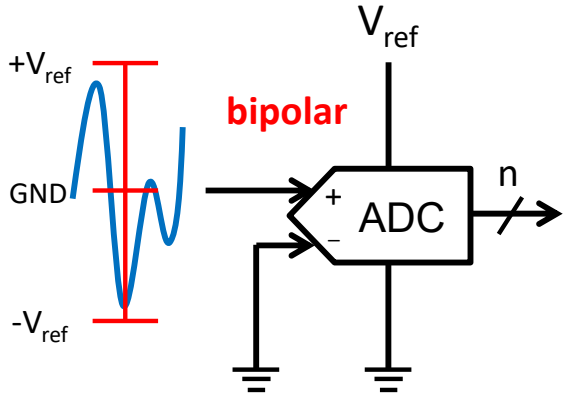




- Conversores pseudo-diferenciales:





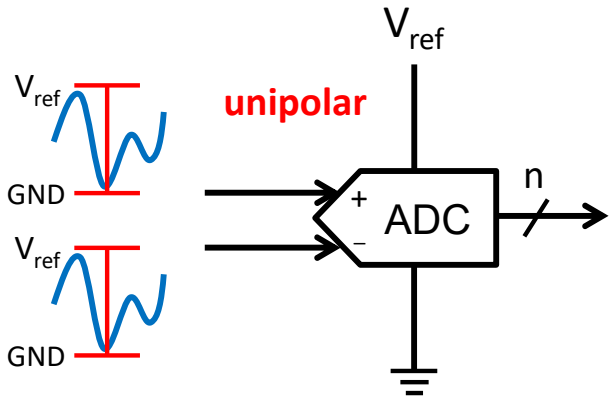


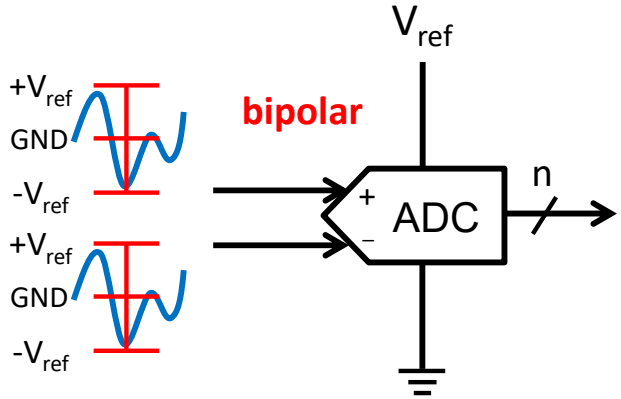
J.M. Mendías 2022

Sistemas muestreados

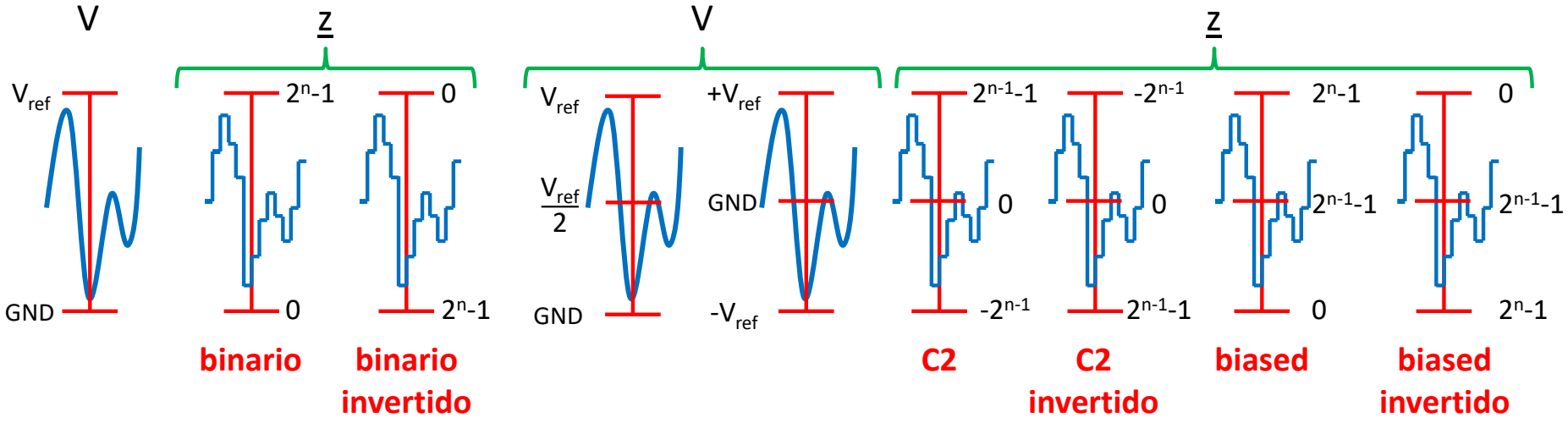
conversores analógico-digital (iii)

- Conversores completamente diferenciales:





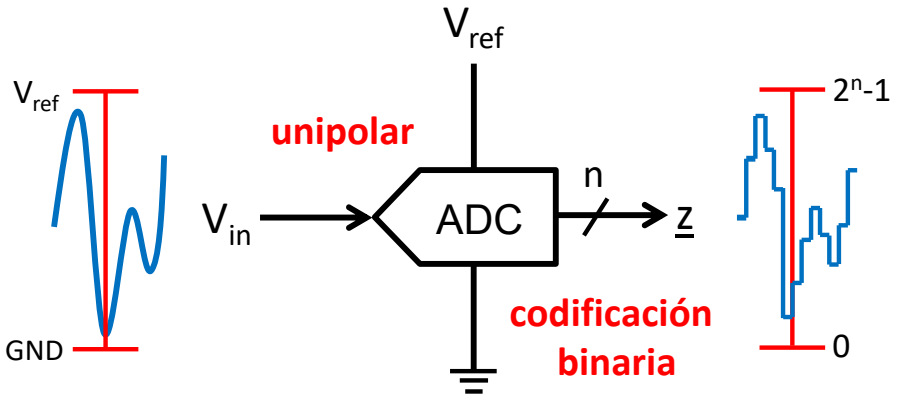
- La relación entre el voltaje (absoluto o diferencial) y el código generado:



Sistemas muestreados

conversores analógico-digital (iv)

- El código digital generado para un voltaje depende del tipo de ADC.
  - Se denomina resolución (o LSB) la cantidad de voltaje necesario para que un ADC pase de generar un código a generar el siguiente adyacente.
  - El error de medida introducido por el ADC (error de cuantización) es ±LSB/2



$$\underline{z} = \frac{2^n}{V_{ref}} \cdot V_{in} \qquad LSB = \frac{V_{ref}}{2^n}$$

$$V_{in} = \frac{V_{ref}}{2^n} \cdot \underline{z} \qquad FS = V_{ref} - LSB$$

- Por ejemplo: ADC unipolar de 10b con codificación binaria y V_{ref} = 3.3V
  - Calcular su resolución, el código generado para V_{in}=1.5V y el voltaje al que corresponde el código 120 (0x78)

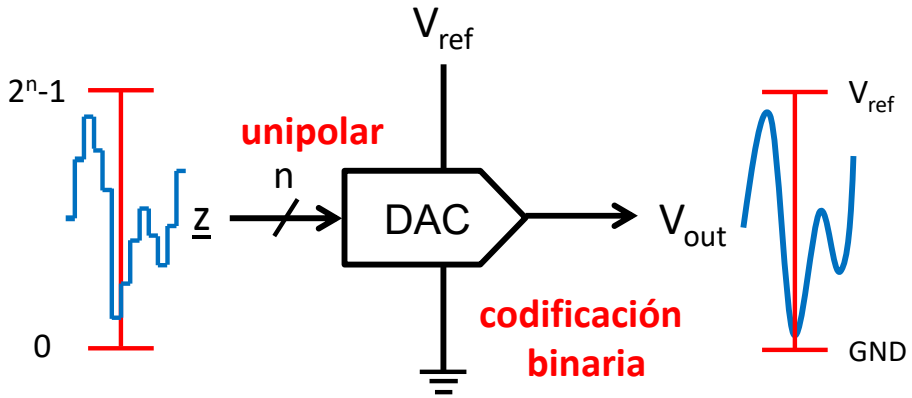
$$LSB = \frac{3.3V}{2^{10}} = 3.2\,mV \qquad \mathrm{z} = \frac{2^{10}}{3.3V} \cdot 1.5V = 465 \qquad V_{in} = \frac{3.3V}{2^{10}} \cdot 120 = 0.39\,V\ (\pm 1.6\,mV)$$

Sistemas muestreados

conversores digital-analógico

- Un DAC, a su vez, puede ser:
  - Unipolar: la señal de salida toma valores comprendidos entre tierra y V_{ref}
  - Bipolar: la señal de salida toma valores comprendidos entre $-V_{ref}$ y $+V_{ref}$

- La relación entre voltaje generado y código digital en un DAC es análoga a la existente en un ADC y depende del tipo de DAC



$$V_{out} = \frac{V_{ref}}{2^n} \cdot \underline{z} \qquad LSB = \frac{V_{ref}}{2^n}$$

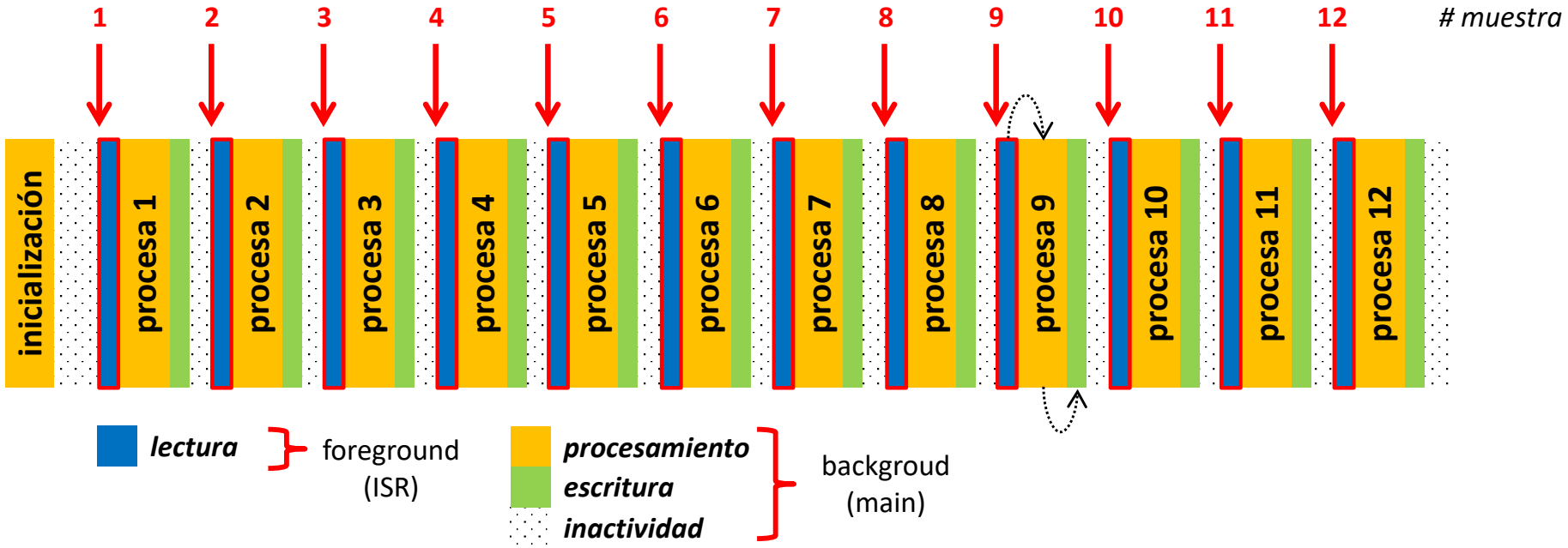
$$\underline{z} = \frac{2^n}{V_{ref}} \cdot V_{out}$$

J.M. Mendías
2022

Sistemas muestreados

procesamiento individual (i)

- Los algoritmos de DSP que procesan las muestras individualmente:
  - Almacenan la muestra en una variable atómica.
  - El tiempo de lectura+procesado+escritura de cada muestra debe ser inferior al periodo de muestreo.
  - La latencia del sistema equivale a 1 periodo de muestreo.



Sistemas muestreados

procesamiento individual (ii)

- Típicamente presentan una arquitectura background-foreground:
  - Ambas hebras se sincronizan y comunican mediante un mailbox.
  - Un temporizador (o el propio ADC/DAC) genera interrupciones periódicas.
  - La RTI (hebra foreground) lee las muestras en una variable global intermedia
    - la lectura del ADC arranca el proceso de captación de la siguiente muestra
  - Una hebra en background procesa la variable intermedia y escribe las muestras
    - la escritura del DAC arranca su proceso de conversión

```
volatile boolean flag;
volatile sample_t sample;

void main( void )
{
  ...inicializa dispositivos...

  flag = 0;
  while( 1 )
  {
    while( !flag );
    flag = 0;
    processSample( &sample );
    putSample( sample );
  }
}
```

```
void isr( void )
{
  ...borra flag de interrupción pendiente...

  sample = getSample();
  flag = 1;
}
```

si la frecuencia de muestreo está bien dimensionada, no es necesario esperar la disponibilidad del ADC/DAC al leer o escribir la muestra.

Sistemas muestreados

procesamiento individual (iii)

- Para evitar la espera activa de la hebra en background
  - El flag que sincroniza ambas hebras puede reemplazarse por una suspensión del procesador.
  - Cada vez que haya una nueva muestra el procesador se despierta para procesarla.

```
volatile sample_t sample;

void main( void )
{
  ...inicializa dispositivos...

  while( 1 )
  {
    sleep();
    processSample( &sample );
    putSample( sample );
  }
}
```

```
void isr( void )
{
  ...borra flag de interrupción pendiente...

  sample = getSample();
}
```

Sistemas muestreados

procesamiento individual (iv)

- Alternativamente, si el sistema se dedica en exclusividad a procesar datos de una única fuente:
  - El procesamiento y la escritura pueden incluirse en la RTI.
  - O incluso, puede usarse una arquitectura super-loop.

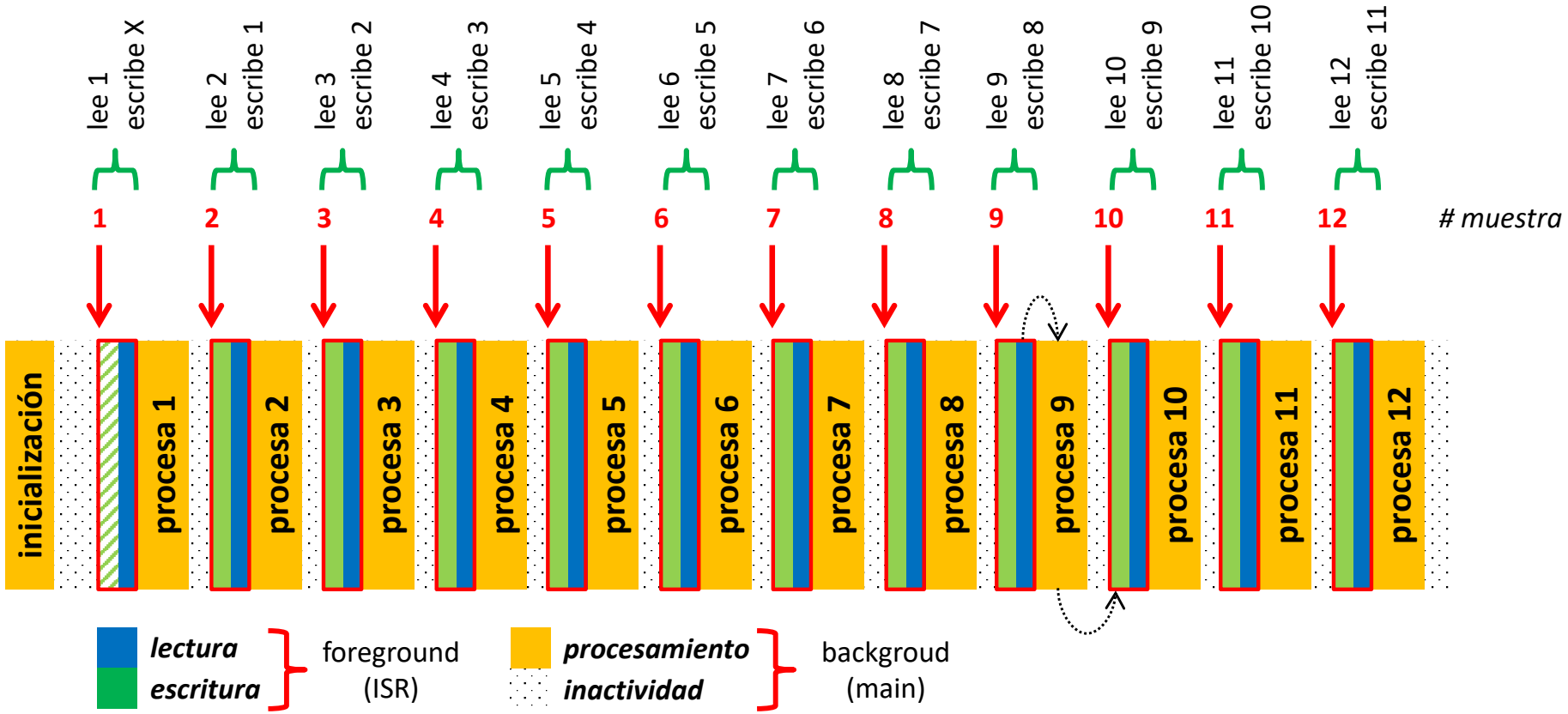
```
void main( void )
{

  ...inicializa dispositivos...
  while( 1 )
     sleep();

}

void isr( void )
{
  sample_t sample;

  ...borra flag de int. pendiente...

  sample = getSample();
  processSample( &sample );
  putSample( sample );
}
```

```
void main( void )
{
  sample_t sample;

  ...inicializa dispositivos...
  while( 1 )
  {
    while( ...no fin periodo de muestreo... );
    sample = getSample();
    processSample( &sample );
    putSample( sample );
  }
}
```

Sistemas muestreados

procesamiento individual (vi)

- La cadencia de lectura de muestras es exacta
  - Las lecturas están disparadas por la interrupción del temporizador
  - La cadencia de escritura de muestras, en cambio, depende la duración de procesado.
- Si se desea asegurar también una cadencia exacta en las escrituras
  - La escritura puede incluirse en la RTI (a costa de que la primera escritura sea falsa)



Sistemas muestreados

procesamiento individual (v)

- Ahora, el tiempo que transcurre entre 2 lecturas o 2 escrituras es constante y equivalente.

```
volatile boolean flag;
volatile sample_t sample;

void main( void )
{
  ...inicializa dispositivos...

  sample = 0;
  flag = 0;
  while( 1 )
  {
    while( !flag );
    flag = 0;
    processSample( &sample );
  }
}
```

```
void isr( void )
{
  ...borra flag de interrupción pendiente...

  putSample( sample );
  sample = getSample();
  flag = 1;
}
```

la primera muestra enviada DAC es falsa ya que no se deriva de ninguna muestra recibida

Sistemas muestreados

procesamiento múltiple

- Muchos sistemas (i.e filtros) para calcular cada muestra de salida necesitan usar las últimas muestras de entrada y/o salida.
  - Almacenan en un buffer las últimas muestras leídas/calculadas que necesiten.

- Filtro FIR (finite-impulse response):
  - la salida en cada instante depende del valor de las últimas N entradas

ecuación de diferencias

$$y_n = \sum_{k=0}^{N-1} a_k x_{n-k}$$

función de transferencia

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{k=0}^{N-1} a_k z^{-k}$$

- Filtro IIR (infinite-impulse response)
  - la salida en cada instante depende del valor de las últimas N entradas y M salidas.

ecuación de diferencias

$$y_n = \sum_{k=0}^{N-1} a_k x_{n-k} - \sum_{k=1}^{M-1} b_k y_{n-k}$$

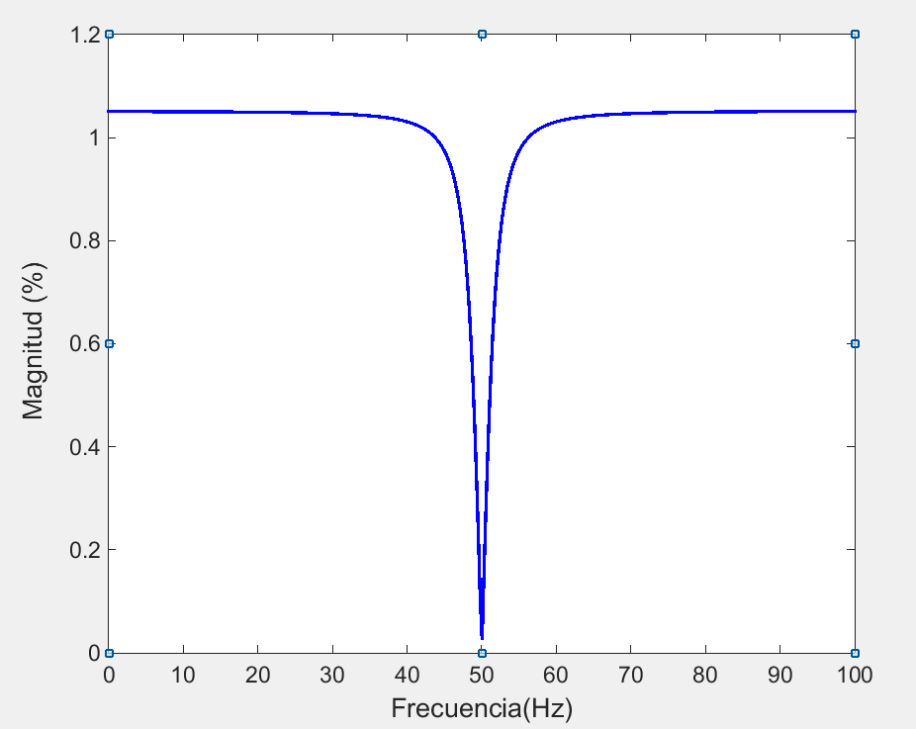
función de transferencia

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N-1} a_k z^{-k}}{1 + \sum_{k=0}^{M-1} b_k z^{-k}}$$

Sistemas muestreados

procesamiento múltiple: ejemplo (i)

- Un notch filter es un filtro que atenúa las frecuencias en un rango muy estrecho dado y deja inalteradas el resto de frecuencias.
  - Uno de sus usos es para eliminar las interferencias causadas por la corriente alterna que alimenta los equipos (60 Hz en EEUU y 50 Hz en el resto del mundo).



Sistemas muestreados

procesamiento múltiple: ejemplo (ii)

- Un notch filter puede implementarse como un filtro IIR de 2do. orden
  - Su función de transferencia con 2 ceros en f_{stop} y 2 polos situados a una distancia α de los polos es:

$$H(z) = \frac{1 - 2\cos(\theta)\, z^{-1} + z^{-2}}{1 - 2\alpha\cos(\theta)\, z^{-1} + \alpha^2 z^{-2}} \qquad donde\ \ \theta = 2\pi\frac{f_{stop}}{f_s}\ y\ 0 < \alpha < 1$$

  - Que se expresa como ecuación de diferencias:

$$y_n = x_n - 2\cos(\theta)\, x_{n-1} + x_{n-2} + 2\alpha\cos(\theta)\, y_{n-1} - \alpha^2 y_{n-2}$$

- Por ejemplo, supongamos que deseamos implementar un notch filter para eliminar el ruido de alimentación de un ECG:

$$f_{stop} = 50\ Hz \qquad f_s = 250\ Hz \qquad \alpha = 0.95 \qquad\qquad cos\left(2\pi\frac{50}{250}\right) = 0{,}309017$$

- La ecuación de diferencias queda:

$$y_n = x_n - 0.618034 \cdot x_{n-1} + x_{n-2} + 0.5871323 \cdot y_{n-1} - 0.9025 \cdot y_{n-2}$$

Sistemas muestreados

procesamiento múltiple: ejemplo (iii)

- La ecuación de diferencias puede expresarse con aritmética entera
  - Asumiendo que las señales x e y tienen la misma representación
  - Multiplicando y dividiendo cada constante por 2^m (en este caso m=15)

$$y_n = \frac{-20252 \cdot x_{n-1} + 19239 \cdot y_{n-1} - 29573 \cdot y_{n-2}}{32768} + x_n + x_{n-2}$$

```
void notch_filter_isr( void )
{
  static int16 x[3] = {0, 0, 0};   ····· array de muestras de entrada: x[i] = x_{n-i}
  static int16 y[3] = {0, 0, 0};   ····· array de muestras de salida: y[i] = y_{n-i}
  int32 acc;

  ...borra flag de interrupción pendiente...

  x[0] = getSample();   ····· recibe la nueva muestra de entrada y la almacena en el array

  acc =  -20252 * (int32)x[1];
  acc +=  19239 * (int32)y[1];
  acc += -29573 * (int32)y[2];
  acc >>= 15;                    } calcula la muestra de salida
  acc += x[0];
  acc += x[2];

  putSample( y[0] = acc );   ····· almacena en el array la nueva muestra de salida y la envía

  x[2] = x[1]; x[1] = x[0];
  y[2] = y[1]; y[1] = y[0];      } desplaza las muestras en los arrays para la siguiente iteración
}
```

Sistemas muestreados

procesamiento múltiple: ejemplo (iv)

- De manera más genérica en punto fijo Q1.15

```
#define QM 15
void notch_filter_isr( void )
{
  const int16 a[3] = { 32767, -20252, 32767 };  // 1, -0.618034, 1
  const int16 b[3] = { 0, 19239, -29573 };      // 0, 0.5871323, -0.9025
  static int16 x[3] = {0, 0, 0};
  static int16 y[3] = {0, 0, 0};

  int32 acc;

  ...borra flag de interrupción pendiente...

  x[0] = getSample();

  acc =  a[0] * (int32)x[0];
  acc += a[1] * (int32)x[1];
  acc += a[2] * (int32)x[2];
  acc += b[1] * (int32)y[1];
  acc += b[2] * (int32)y[2];

  putSample( y[0] = acc >> QM );

  x[2] = x[1]; x[1] = x[0];
  y[2] = y[1]; y[1] = y[0];
};
```

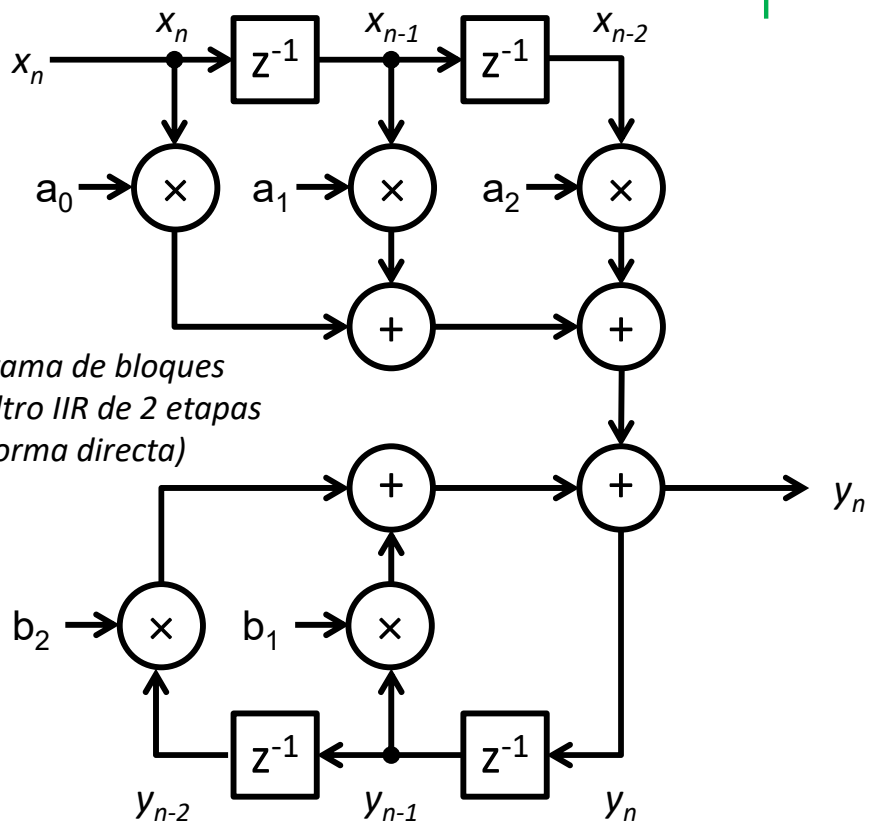


diagrama de bloques de un filtro IIR de 2 etapas (forma directa)

Sistemas muestreados

procesamiento múltiple: array

```
#define QM ...          factor de escala de los datos
#define N ...           número de etapas del filtro

const int16 a[N] = { ... }      coeficientes constantes del filtro (escalados)
int16 x[N] = {0, ..., 0};       array de muestras de entrada: x[i] = x_{n-i}

void FIR_filter_isr( void )
{
  uint8 i;
  int32 y;          debe ser lo suficiente ancho para evitar overflow durante el calculo
  ...borra flag de interrupción pendiente...

  x[0] = getSample();           almacena la nueva muestra

  y = 0;
  for( i=0; i<N; i++ )
    y += a[i] * (int32)x[i];    realiza la convolución (asegurando que no hay pérdida de precisión)

  putSample( y >> QM );         escribe el resultado corrigiendo el escalado

  for( i=N-1; i>0; i-- )
    x[i] = x[i-1];              desplaza las muestras en el array
}
```

	0	1	2
1	x_1	0	0
2	x_2	x_1	0
3	x_3	x_2	x_1
4	x_4	x_3	x_2

N = 3

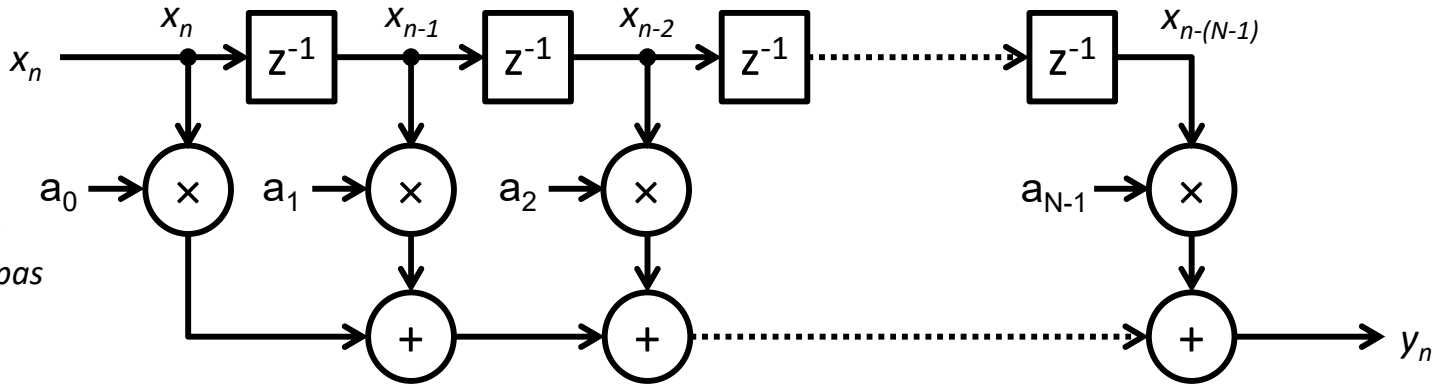


diagrama de bloques de un filtro FIR de N etapas (forma directa)

Sistemas muestreados

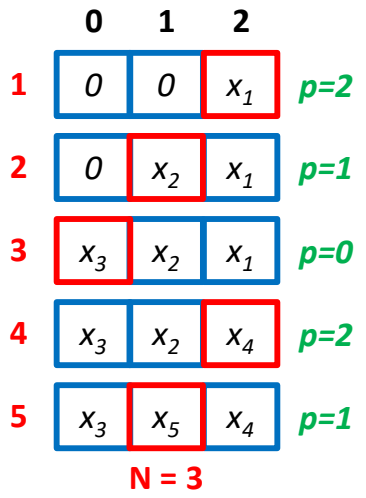
procesamiento múltiple: buffer circular simple

- Para evitar el movimiento de muestras se usa un buffer circular simple.
  - Por eficiencia, requiere dividir en 2 el bucle de convolución y tener doble índice.

```
#define QM ...
#define N ...

int16 a[N] = { ... };
int16 x[N] = {0, ..., 0};···· array de muestras de entrada: x[(p+i) mod N] = x_{n-i}
uint8 p = N-1;····················· pivote: índice de la muestra más reciente

void FIR_filter_isr( void )
{
  uint8 xi, ai; ····· índices diferentes para recorrer muestras y coeficientes
  int32 y;

  ...borra flag de int. pendiente...

  x[p] = getSample();

  y = 0;
  for( xi=p, ai=0; xi<N; xi++, ai++ )
    y += a[ai] * (int32)x[xi];
  for( xi=0; xi<p; xi++, ai++ )
    y += a[ai] * (int32)x[xi];

  putSample( y >> QM );

  p = ( p ? p-1 : N-1 ); ····· decrementa modularmente el pivote
}
```

realiza el cálculo para las posiciones [p..N-1] del array de muestras

realiza el cálculo para las posiciones [0..p-1] del array de muestras



Sistemas muestreados

procesamiento múltiple: buffer circular doble

- Para evitar el doble índice puede usarse un buffer circular doble.
  - Usa un índice único pero duplica la memoria ocupada por las muestras.

```
#define QM ...
#define N ...

int16 a[N] = { ... };
int16 x[2*N] = {0, ..., 0};  ···· el array tiene tamaño doble
uint8 p = N-1;

void FIR_filter_isr( void )
{
  int8 *xaux;
  uint8 i;  ····· índice común para recorrer muestras y coeficientes
  int32 y;

  ...borra flag de int. pendiente...

  x[p] = x[p+N] = getSample();  ····· las muestras se almacenan 2 veces

  y = 0;
  xaux = x + p;
  for( i=0; i<N; i++ )
    y += a[i] * (int32)xaux[i];   } realiza el cálculo

  putSample( y >> QM );

  p = ( p ? p-1 : N-1 );
}
```

	0	1	2	3	4	5	
1	0	0	x_1	0	0	x_1	p=2
2	0	x_2	x_1	0	x_2	x_1	p=1
3	x_3	x_2	x_1	x_3	x_2	x_1	p=0
4	x_3	x_2	x_4	x_3	x_2	x_4	p=2
5	x_3	x_5	x_4	x_5	x_3	x_4	p=1

N = 3

Sistemas muestreados

procesamiento múltiple: filtro digital (v)

- Si el rendimiento es clave, la actualización de muestras y la convolución se hacen en ensamblador

```
#define N ...

int8  a[N] = { ... };
int8  x[2*N] = {0, ..., 0
uint8 p = N-1;

extern int16 asmFIR( int16 sample, int16 *a, int16 *x, uint8 *p );

void FIR_filter_isr( void )
{
  ...borra flag de int. pendiente...

  putSample( asmFIR( getSample(), a, x, &p ) );
}
```

```
.equ QM, ...
.equ N, ...

.global asmFIR
.section .text

asmFIR :
  stmfd  sp!, {r4-r7}

  ldrb   r4, [r3]           // r4 = *p
  add    r6, r2, r4, lsl#1  // x[p] = sample
  strh   r0, [r6]
  add    r6, r6, #(2*N)     // x[p+N]= sample
  strh   r0, [r6]

  mov    r0, #0             // y = 0
  add    r2, r2, r4, lsl#1  // xaux = x + p
  mov    r5, #N-1           // i = N-1
```

```
loop :
  ldrsh  r6, [r1], #2       // r6 = *(a++);
  ldrsh  r7, [r2], #2       // r7 = *(xaux++);
  mla    r0, r6, r7, r0     // y = a*xaux + y
  subs   r5, r5, #1         // i--
  bne    loop               // if( !i ) goto loop

  cmp    r4, #0
  subne  r4, r4, #1         // if( p ) p = p-1
  moveq  r4, #N-1           // if( !p ) p = N-1
  strb   r4, [r3]

  mov    r0, r0, asr#QM     // y = y >> QM

  ldmfd  sp!, {r4-r7}
  bx     lr

.ltorg
.end
```

Sistemas muestreados

procesamiento múltiple: filtro digital (vi)

- La carga computacional de un filtro digital crece conforme el número de etapas aumenta.
  - Si el número de etapas es alta, puede que los cálculos no puedan realizarse durante el intervalo entre dos muestras.

- Empíricamente, el número máximo de etapas que podría tener un filtro FIR para ejecutarse en tiempo real en el S3C44BOX
  - fclk = 64 Mhz, fs = 16000 Hz, 16b datos
  - ciclos de reloj entre muestras = 64 MHz / 16 Kz = 4000

arquitectura SW (punto fijo)	C	ASM	C	ASM
	en SDRAM		en SRAM (scratchpad)	
buffer lineal	5	21	25	149
buffer circular simple	6	32	35	230
buffer circular doble	7	35	39	238

J.M. Mendías
2022

Sistemas muestreados

wavetables (i)

- En sistemas DSP es común necesitar generar señales periódicas de distintas frecuencias ($f < f_s/2$) que tengan una forma de onda dada
  - Por ejemplo, para modular en amplitud (AM) o en frecuencia (FM) otra señal
- Una sinusoide puede generarse algorítmicamente en tiempo real:
  - Pero es un método computacionalmente muy costoso

```
#include <math.h>

#define PI 3.14159265358979323846

#define FS 16000 ..... frecuencia de muestreo
#define F  1000  ..... frecuencia de la sinusoide
#define A  .... ..... amplitud de la sinusoide

const float radPerSample = 2*PI*F/FS; ..... radianes que recorre una sinusoide de frecuencia f
                                           entre muestra y muestra (t=1/fs)
void main( void )
{
  float x = 0;
  ...
  while( 1 ) {
    putSample( (int16)(A*sin(x)) );
    x += radPerSample;                  } realiza cálculos en punto flotante y usa funciones matemáticas
    if( x > 2*PI ) x -= 2*PI; ..... evita que x desborde, sabiendo que sin(2π + x) = sin( x )
  }
}
```

Sistemas muestreados

wavetables (ii)

- En tiempo real es mucho más eficiente indexar un buffer circular
  - Que contenga precalculada la sinusoide (o forma de onda) a generar

```
#include <math.h>

#define PI 3.14159265358979323846

#define FS    16000      // frecuencia de muestreo
#define F     1000       // frecuencia de la sinusoide
#define SIZE (FS/F)      // tamaño de la tabla
#define A     ...        // amplitud de la sinusoide (normalizada [0..1] y en formato Q1.15)

const float radPerSample = 2*PI*F/FS;

int16 waveTable[SIZE];   // almacena los valores de la sinusoide en formato Q1.15

void main( void )
{
  uint16 i, n=0;
  ...
  for( i=0; i<SIZE; i++ )
    waveTable[i] = TOFIX( fix16, sin(radPerSample*i), 15 );

  while( 1 ) {
    putSample( (A*(int32)waveTable[n]) >> 15 );
    n = n+1;
    if( n >= SIZE ) n = 0;
  }
}
```

la tabla puede rellenarse durante la inicialización o declararse ya inicializada

todos los cálculos son enteros

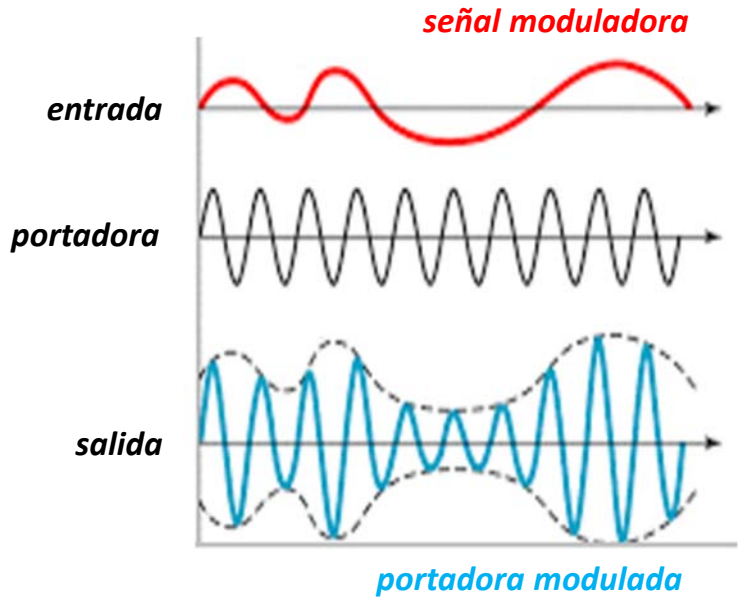
indexa circularmente la tabla

Sistemas muestreados

wavetables (iii)

- Para modular una señal portadora sinusoidal con una señal de entrada,
  - basta con multiplicar la entrada por elementos consecutivos de la waveTable

```
#include <math.h>

#define PI 3.14159265358979323846

#define FS    16000
#define F     1000
#define SIZE (FS/F)


const float radPerSample = 2*PI*F/FS;

int16 waveTable[SIZE];

void main( void )
{
  uint16 i, n=0;
  ...
  for( i=0; i<SIZE; i++ )
    waveTable[i] = TOFIX( fix16, sin(radPerSample*i), 15 );

  while( 1 ) {
    putSample(  (getSample()*(int32)waveTable[n]) >> 15 );
    n = n+1;
    if( n >= SIZE ) n = 0;
  }
}
```



J.M. Mendías
2022

Sistemas muestreados

wavetables (iv)

- Si necesitamos generar sinusoides de distintas frecuencias
  - No es necesario disponer de una tabla por frecuencia, basta tener una única tabla conteniendo la sinusoide de menor frecuencia.

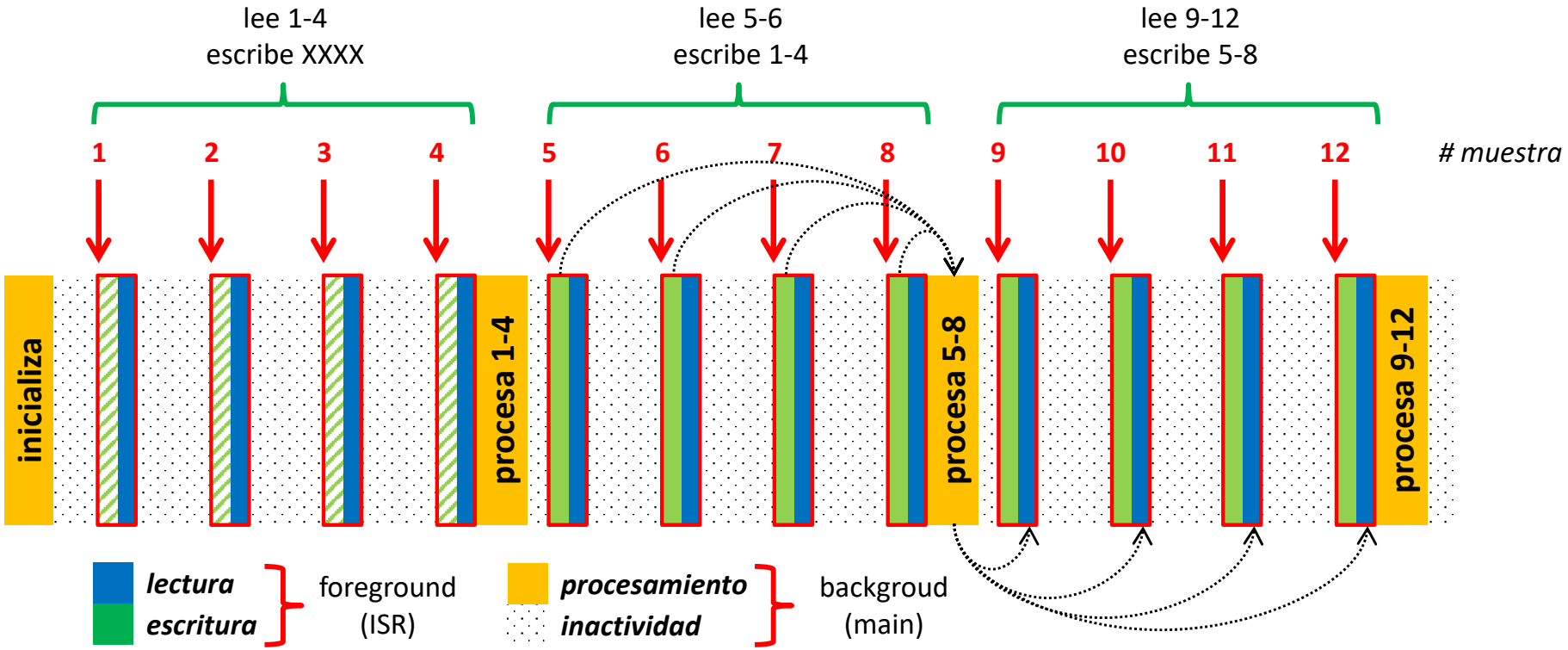
```
#include <math.h>

#define PI 3.14159265358979323846

#define FS    16000
#define FMIN  1 ····· frecuencia de la sinusoide mínima a generar
#define SIZE (FS/FMIN)
#define A     ...

const float radPerSample = 2*PI*FMIN/FS;

int16 waveTable[SIZE];

void main( void )
{
  uint16 f; ····· frecuencia variable de la sinusoide a generar en cada momento
  uint16 i, n=0;
  ...
  for( i=0; i<SIZE; i++ )
    waveTable[i] = TOFIX( fix16, sin(radPerSample*i), 15 );

  while( 1 ) {
    putSample( (A*(int32)waveTable[n]) >> 15 );
    n = n + f/FMIN; ····· el índice se incrementa dependiendo de la frecuencia a generar
    if( n >= SIZE ) n -= SIZE;
  }
}
```

J.M. Mendías
2022

Sistemas muestreados

procesamiento en bloques: buffer simple (i)

- Los algoritmos de DSP (i.e. FFT) que procesan por bloques de N muestras:
  - Almacenan las N muestras del bloque en un buffer.
  - El tiempo de lectura+escritura de la última muestra añadido al tiempo de procesado de cada bloque debe ser inferior al periodo de muestreo.
    - El tiempo disponible para procesar N muestras es el mismo que para procesar 1.
  - La latencia del sistema equivale a N periodos de muestreo.



J.M. Mendias
2022

Sistemas muestreados

procesamiento en bloques: buffer simple (ii)

- Se utiliza un buffer de para almacenar cada bloque a procesar
  - La RTI lee y escribe individualmente las muestras del buffer
    - Cuando lo completa lo señaliza activando un flag.
  - Una hebra en background procesa y sobreescribe el buffer completo.
    - Solo tras detectar la activación del flag.

```
volatile boolean flag;
volatile sample_t buffer[N];

void main( void )
{
  ...inicializa dispositivo...
  ...inicializa el buffer...
  flag = 0;
  while( 1 )
  {
    while( !flag );
    flag = 0;
    processBuffer( buffer );
  }
}
```

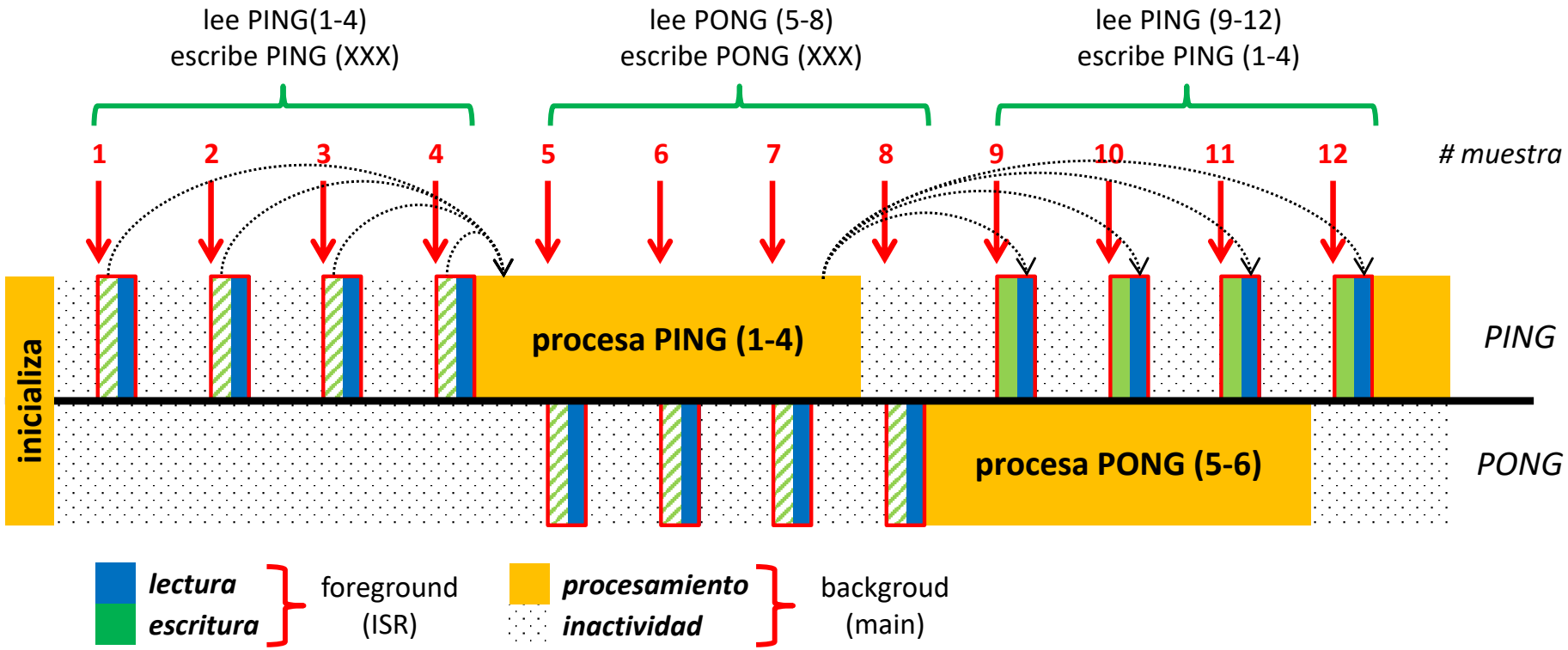
```
void isr( void )
{
  static uint16 i = 0;

  ...borra flag de int. pendiente...

  putSample( buffer[i] );
  buffer[i] = getSample();
  if( i < N-1 )
    i = i+1;
  else {
    i = 0;
    flag = 1;
  }
}
```

J.M. Mendías 2022

Sistemas muestreados

procesamiento en bloques: buffer doble (i)

- El tiempo disponible de procesado puede aumentarse un buffer doble.
  - Cada grupo de N muestras se almacena alternativamente en un buffer distinto.
  - Mientras que se lee/escribe de un buffer, se procesa el otro buffer
    - El tiempo disponible para procesar N muestras es N veces el disponible para procesar 1.
  - La latencia del sistema equivale a 2×N periodos de muestreo.



Sistemas muestreados

procesamiento en bloques: buffer doble (ii)

- Se utilizan alternativamente 2 bufferes para almacenar los bloques
  - La RTI lee y escribe individualmente las muestras en uno de los bufferes
    - Cuando lo completa lo señaliza activando un flag y pasa a utilizar el otro
  - Una hebra en background procesa y sobreescribe el buffer que no usa la RTI.
    - Solo tras detectar la activación del flag.

```
volatile boolean flag;
volatile boolean pingpong;
volatile sample_t buffer[2][N];

void main( void )
{
  ...inicializa dispositivo...
  ...inicializa el buffer...
  flag = 0;
  pingpong = 0;
  while( 1 )
  {
    while( !flag );
    flag = 0;
    if( pingpong == 0 )
      processBuffer( buffer[1] );
    else
      processBuffer( buffer[0] );
  }
}
```

```
void isr( void )
{
  static uint16 i = 0;

  ...borra flag de int. pendiente...

  putSample( buffer[pingpong][i] );
  buffer[pingpong][i] = getSample();
  if( i < N-1 )
    i = i+1;
  else {
    i = 0;
    pingpong = !pingpong;
    flag = 1;
  }
}
```

Sistemas de control

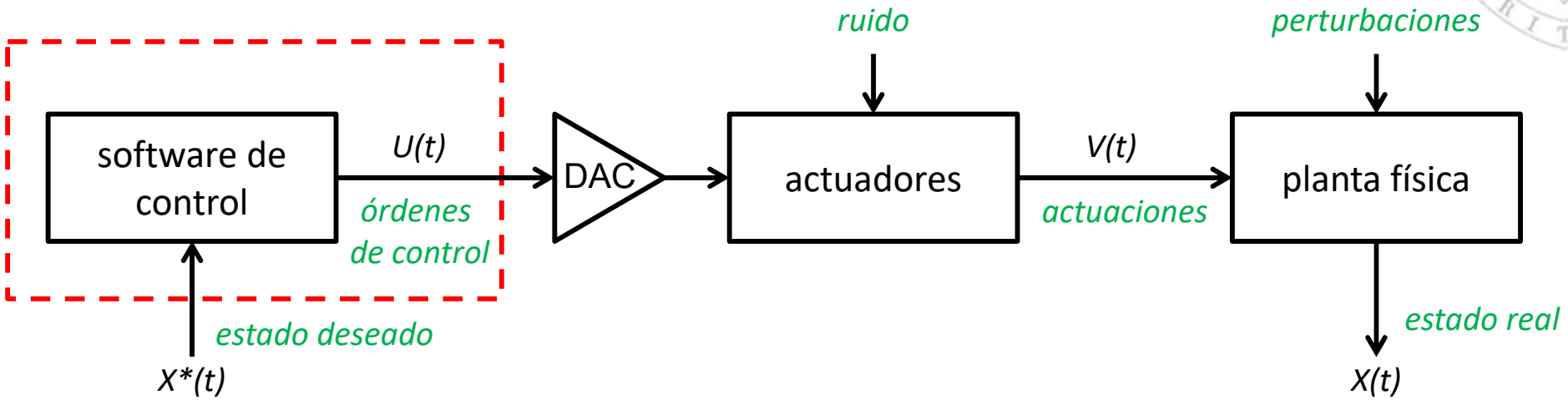
introducción

- Un sistema de control es una colección de dispositivos eléctricos y mecánicos conectados con el objetivo de regular el estado de una planta física.
  - Estado: conjunto de propiedades que se desea regular (i.e temperatura, posición, velocidad, presión...)
    - pueden medirse directamente o indirectamente
  - Entrada: el estado en el que se desea que esté el sistema.
  - Salida: el estado real en el que se encuentra el sistema.

- Según su arquitectura existen 2 tipos de sistemas de control
  - Lazo abierto: la salida del sistema no tiene efecto sobre las acciones de control
    - No están realimentados: el sistema desconoce si las acciones de control se traducen realmente en el cambio deseado del estado de la planta
  - Lazo cerrado: la salida del sistema tiene efecto directo sobre las acciones de control
    - Están realimentados: conocen el efecto de las acciones de control ya que disponen de una señal de error que en todo momento determina la diferencia entre el estado deseado y el estado real de la planta.

Sistemas de control

control en lazo abierto

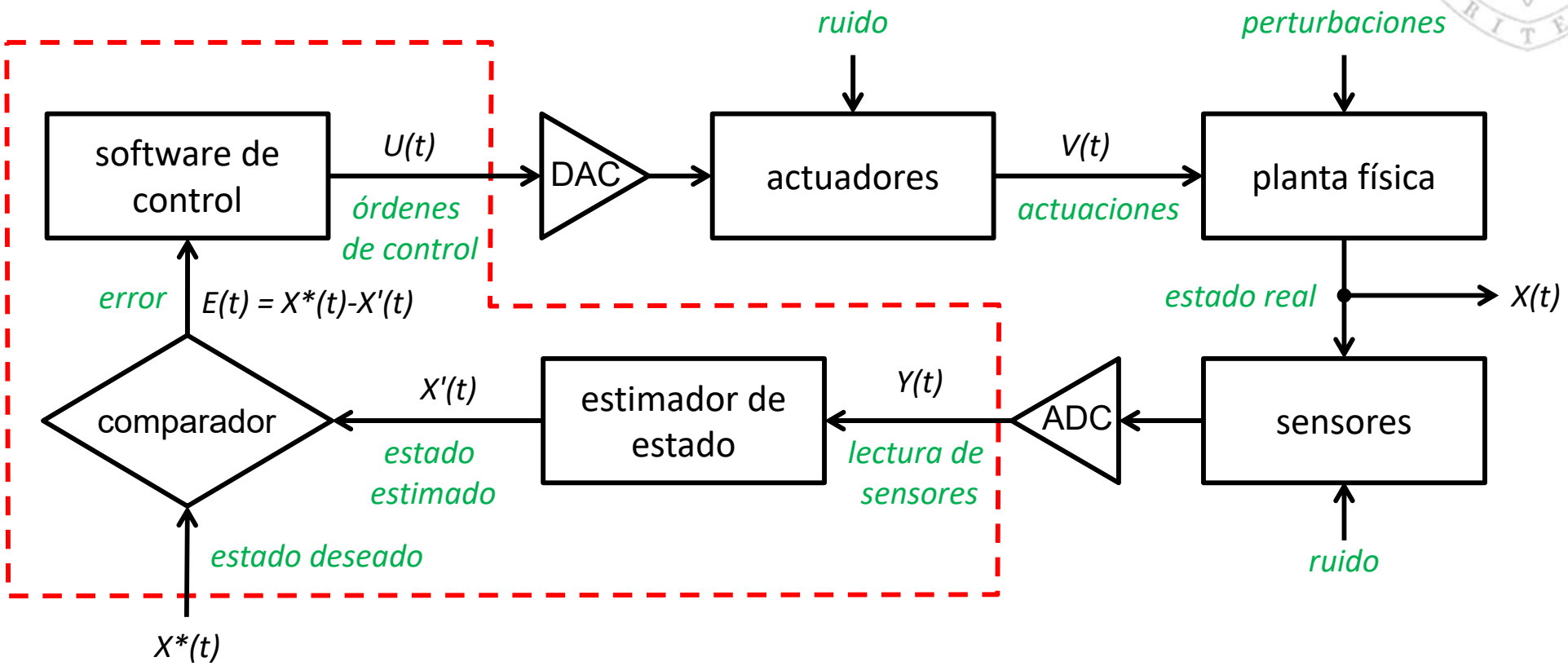


- El control en lazo abierto es aplicables si:
  - La relación entre la entrada y la salida es conocida
  - Las perturbaciones no tienen un efecto significativo en el estado real del sistema
- El diseño del software de control es simple:
  - Cada entrada tiene una orden o secuencia de ordenes de control fijas
    - La precisión del sistema requiere de una adecuada calibración
  - Dado un valor de X*, U puede calcularse:
    - en tiempo real mediante la ejecución de la correspondiente función
    - accediendo a una lookup table que contenga los resultados precalculados
  - Sistemas que requieran secuenciamiento de órdenes se implementan como FSM.

J.M. Mendías
2022

Sistemas de control

control en lazo cerrado



- El control en lazo cerrado se usa cuando no es factible el control en lazo abierto:
  - Perturbaciones imprevisibles, variaciones no previstas en la planta, etc.
  - Pueden no ser estables a causa de la tendencia a sobrecorregir errores.
- Se implementan como sistemas muestreados background-foreground
  - la frecuencia de muestreo bastante superior a la constante de tiempo de la planta.

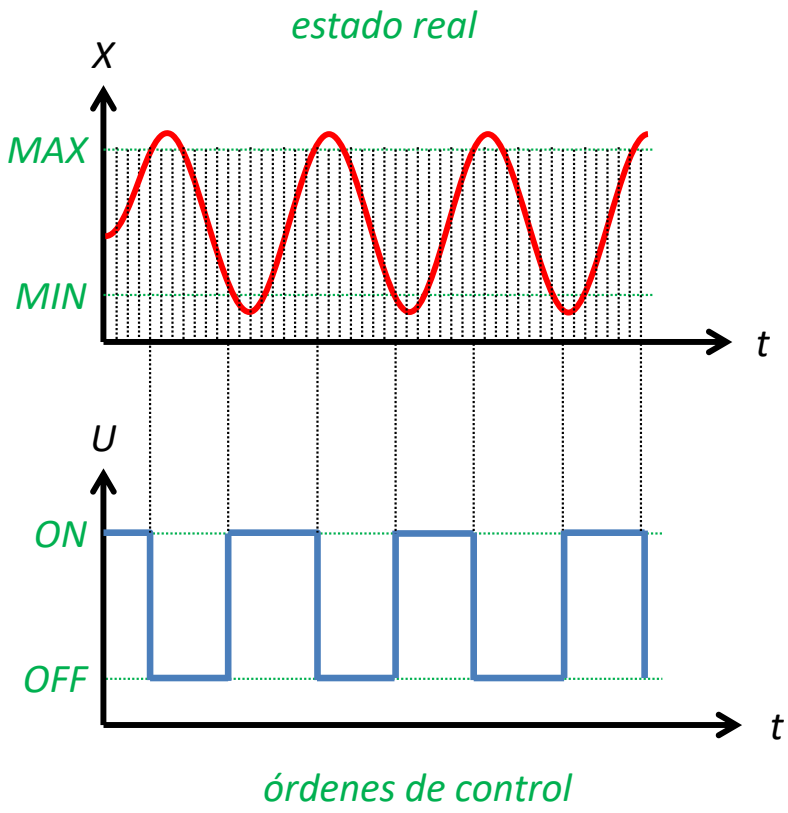
J.M. Mendías
2022

Sistemas de control

controlador bang-bang

- El controlador bang-bang es un sistema control en donde las acciones de control solo son ON-OFF.
  - Periódicamente lee el valor del estado del sistema y actúa en consecuencia (i.e un control de temperatura por termostato).

```
int8 min, max;
boolean u;

void isr( void )
{
  int8 x;

  ...borra flag de int. pendiente...

  x = estimate( getSample() );

  if( x < min )
    u = ON;
  else if( x > max )
    u = OFF;
}
```



J.M. Mendias
2022

Sistemas de control

controlador incremental

- Un controlador incremental es un sistema de control donde la salida del actuador toma un conjunto discreto de valores
  - Asume que cuando el actuador incrementa, el estado incrementa y viceversa (i.e. control de posición)

```
#define MAX_ERROR ...          máximo error tolerable
#define MIN ...
#define MAX ...
int8 u;
int8 xref;                     estado deseado del sistema

void isr( void )
{
  int8 x;
  int16 e;

  ...borra flag de int. pendiente...

  x = estimate( getSample() );
  e = xref - x;
  if( e < -MAX_ERROR )         incrementa/decrementa si el error es mayor de lo aceptable
    u = (u == MIN ? MIN : u-1);
  else if( e > MAX_ERROR )     satura la acción de control
    u = (u == MAX ? MAX : u+1);
}
```

Sistemas de control

controlador PID (i)

- En teoría de control lineal un controlador PID (Proportional-Integral-Differential) satisface la siguiente ecuación:

$$U(t) = \underbrace{K_P E(t)}_{P(t)} + \underbrace{K_I \int_0^t E(\tau)d\tau}_{I(t)} + \underbrace{K_D \frac{dE(t)}{dt}}_{D(t)} \quad donde\ E(t) = X^*(t) - X'(t)$$

- La ecuación está formada por la suma de 3 términos:
  - El primero (P), proporcional al error actual.
  - El segundo (I) proporcional a la integral del error sobre el tiempo:
    - Es decir, proporcional a un cierto nivel de error permanente.
  - El tercero (D) proporcional a la primera derivada del error.
    - Es decir, proporcional al velocidad de cambio del error.

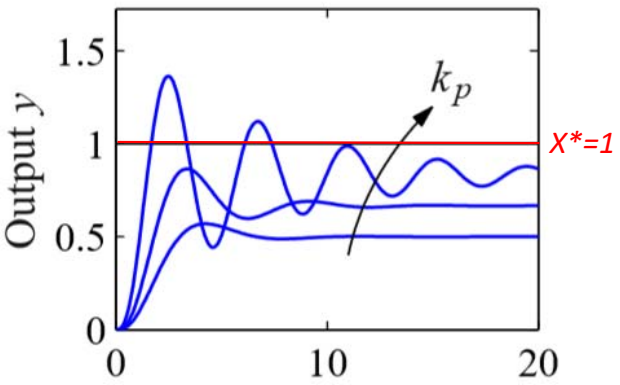
- Es usado por más del 90% de los controladores de sistemas SISO (single-input, single-output)
  - Efectivo y fácil de implementar
  - Para cada caso, requiere un ajuste cuidadoso de las constantes de proporcionalidad.

J.M. Mendías
2022

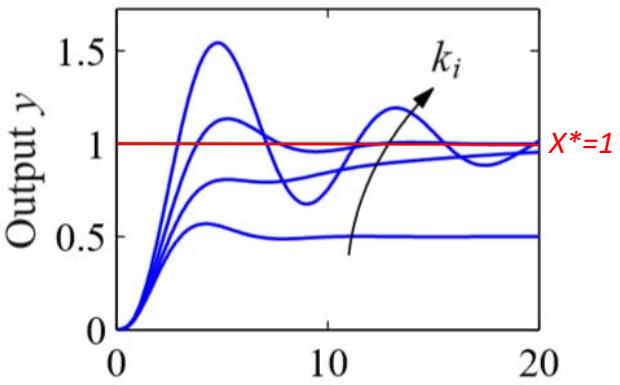
Sistemas de control

controlador PID (ii)

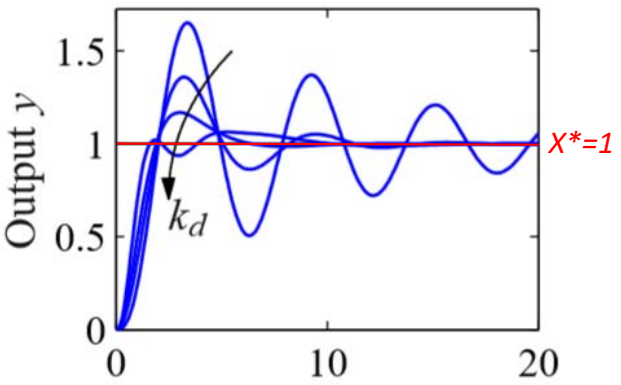
- El término proporcional minimiza el error aunque nunca lo hace 0
- El término integral consigue reducir a 0 el error permanente
- El término derivativo incrementa la estabilidad y reduce el sobreimpulso



Control P



Control PI



Control PID

fuente figuras: K. J. Aström & R. M. Murray, *Feedback Systems.* Princeton Univeristy Press, 2009

Sistemas de control

controlador PID – algoritmo de posición (i)

- Para implementarlo como sistema muestreado a frecuencia f_s expresamos en tiempo discreto cada una de las componentes de la ecuación.
  - El término proporcional se expresa de manera análoga:

$$P(t) = K_P E(t) \;\Rightarrow\; \boldsymbol{P_n = K_P \cdot E_n}$$

  - El término integral se expresa como un sumatorio:

$$I(t) = K_I \int_0^t E(\tau)d\tau \;\Rightarrow\; \boldsymbol{I_n} = K_I \sum_{i=1}^{n} (E_i \cdot \Delta t) = \boldsymbol{I_{n-1} + \frac{K_I \cdot E_n}{f_s}}$$

    - dado que es acumulativo suele acotarse para evitar que domine sobre otros términos, a esto se denomina anti-reset windup
  - El término diferencial se expresa como una diferencia entre distintas muestras:

$$D(t) = K_D \frac{dE(t)}{dt} \;\Rightarrow \boldsymbol{D_n} = K_D \frac{E_n - E_{(n-1)}}{\Delta t} = \boldsymbol{K_D \cdot f_s \cdot (E_n - E_{n-1})}$$

    - para evitar que el ruido en la toma de alguna muestra genere grandes errores, se suele hacer la media de 2 diferencias calculadas en lapsos de tiempo distintos

$$D(t) = K_D \left[\frac{1}{2}\frac{E_n - E_{n-3}}{3\Delta t} + \frac{1}{2}\frac{E_{n-1} - E_{n-2}}{\Delta t}\right] = K_D \cdot f_s \cdot \frac{E_n - 3E_{n-1} - 3E_{n-2} - E_{n-3}}{6}$$

J.M. Mendías
2022

Sistemas de control

controlador PID – algoritmo de posición (ii)

- Supongamos que se desea implementar un algoritmo de control PID:
  - Como un sistema muestreado a 100 Hz, con un DAC y un ADC de 8 bits.
  - La planta tiene una entrada analógica V(t) ∈[0..1000] y una salida analógica X(t)∈[0..10]
  - El sistema de control debe satisfacer la ecuación de control:

$$V(t) = 0.1 \cdot E(t) + 0.5 \cdot \int_0^t E(\tau)d\tau + 0.005 \cdot \frac{dE(t)}{dt} \; donde \; E(t) = X^*(t) - X(t)$$

- Cada uno de los términos en tiempo discreto de la ecuación sería:

$$P_n = 0.1 \cdot E_n \qquad I_n = I_{n-1} + \frac{0.5 \cdot E_n}{100} \qquad D_n = 0.005 \cdot 100 \cdot (E_n - E_{n-1})$$

- El valor calculado por cada término debe escalarse:
  - La ecuación calcula V(t)∈[0..10], pero el algoritmo escribe un DAC con resolución [0..256]

$$\frac{V_n}{U_n} = \frac{10}{256} \Rightarrow U_n = \frac{256}{10} \cdot V_n \qquad \boldsymbol{P_n = \frac{256}{100} \cdot E_n} \qquad \boldsymbol{I_n = I_{n-1} + \frac{16}{125} \cdot E_n} \qquad \boldsymbol{D_n = \frac{64 \cdot (E_n - E_{n-1})}{5}}$$

- Las muestras de entrada también deben escalarse:
  - Los valores de X(t) ∈[0..1000] se leen con un ADC con resolución [0..256]

$$\frac{Y_n}{X_n} = \frac{256}{1000} \Rightarrow X_n = \frac{1000}{256} \cdot Y_n \qquad \boldsymbol{E_n = X_n^* - \frac{1000}{256} \cdot Y_n}$$

J.M. Mendias
2022

Sistemas de control

controlador PID – algoritmo de posición (iii)

```
#define MIN_I ...
#define MAX_I ...
#define MIN_U ...
#define MAX_U ...
int32 xref;                                  estado deseado del sistema

void isr( void )
{
  int32 x, e, p, d, u;
  static int32 i = 0, last_e = 0;

  ...borra flag de int. pendiente...

  x = (1000 * getSample()) >> 8;             escala la entrada
  e = xref - x;

  p = (e << 8) / 100;
  i += (e << 4) / 125;
  d = ((e - last_e) << 6) / 5;
  if( i > MAX_I ) i = MAX_I;                 satura el término integral (anti-reset windpup)
  else if( i < MIN_I ) i = MIN_I;

  u = p+i+d;
  if( u > MAX_U ) u = MAX_U;                 satura la acción de control
  else if( u < MIN_U ) u = MIN_U;

  putsample( u );

  last_e = e;                                almacena el error para el calculo del término
}                                            diferencial de la próxima muestra
```

J.M. Mendias
2022

Sistemas de control

controlador PID – algoritmo incremental (i)

- Alternativamente, existe otra implementación del controlador PID que tiene algunas ventajas a efectos prácticos :
  - Partiendo de la ecuación discreta como sistema muestreado a frecuencia f_s:

$$U_n = K_P \cdot E_n + \frac{K_I}{f_s} \sum_{i=1}^{n} E_i + K_D \cdot f_s \cdot (E_n - E_{n-1})$$

$$U_{n-1} = K_P \cdot E_{n-1} + \frac{K_I}{f_s} \sum_{i=1}^{n-1} E_i + K_D \cdot f_s \cdot (E_{n-1} - E_{n-2})$$

  - Se restan ambas ecuaciones:

$$U_n - U_{n-1} = K_P \cdot E_n - K_P \cdot E_{n-1} + \frac{K_I}{f_s} \cdot E_n + K_D \cdot f_s \cdot (E_n - E_{n-1}) - K_D \cdot f_s \cdot (E_{n-1} - E_{n-2})$$

  - Se despeja la acción de control y se agrupan constantes:

$$U_n = U_{n-1} + \underbrace{\left(K_P + \frac{K_I}{f_s} + K_D \cdot f_s\right)}_{K_0} \cdot E_n + \underbrace{(-K_P - 2 \cdot K_D \cdot f_s)}_{K_1} \cdot E_{n-1} + \underbrace{(K_D \cdot f_s)}_{K_2} \cdot E_{n-2}$$

Sistemas de control

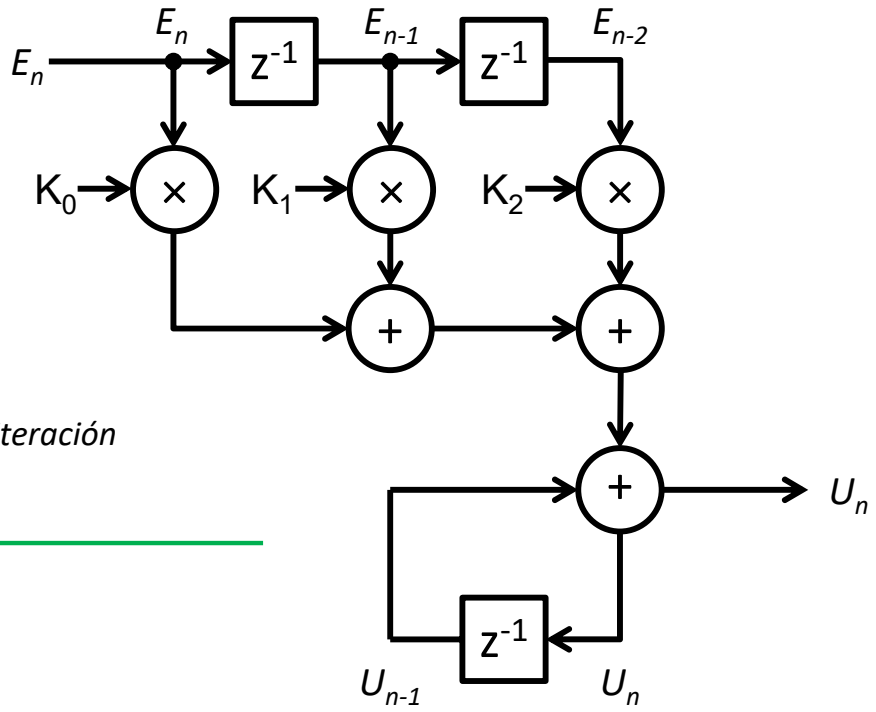
controlador PID – algoritmo incremental (ii)

```
#define MIN_U ...
#define MAX_U ...
int32 xref;

void isr( void )
{
  const int32 K0 = ..., K1 = ..., K2 = ...;
  int32 x, e, u;
  static int32 e1 = 0, e2 = 0, u1 = 0;

  ...borra flag de int. pendiente...

  x = (1000 * getSample()) >> 8;
  e = xref - x;

  u = u1 + K0*e + K1*e1 + K2*e2;
  if( u > MAX_U ) u = MAX_U;
  else if( u < MIN_U ) u = MIN_U;

  putsample( u );

  e2 = e1;
  e1 = e;
  u1 = u;
}
```

coeficientes escalados

desplaza las muestras para la siguiente iteración

diagrama de bloques del controlador PID (forma directa)



Acerca de *Creative Commons*

- Licencia CC (Creative Commons)
  - Ofrece algunos derechos a terceras personas bajo ciertas condiciones. Este documento tiene establecidas las siguientes:

Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.

Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Más información: https://creativecommons.org/licenses/by-nc-sa/4.0/