

Carrera de Especialización en Sistemas Embebidos

Sistemas Operativos en Tiempo Real II

Clase 5: Sistemas reactivos

Repaso: Programación Secuencial Tradicional

- La mayoría de los sistemas embebidos están programados de manera secuencial.
- La espera de eventos se realiza:
 - Por consulta (polling)
 - Bloqueando y esperando
- ¿Qué sucede si tenemos un sistema donde los eventos no ocurren de manera secuencial ?

Repaso: Enfoque Baremetal

- El esquema foreground background (superloop/isr) intercambiar datos a través de variables globales.
- Tiempo de respuesta alto dentro del superloop.
- La temporización de las tareas del superloop no es precisa.
- Es MUY compleja la modularización.

```
void
main( void )
{
    /* global static and stack data */

    static int nTasks = 3;
    int currentTask;

    currentTask = 0;                /* initialization code */

    if(POST())                      /* Power On Self Test succeeds */
    {
        while(TRUE)                /* scheduling executive */
        {
            task1();
            task2();
            task3();
        };
    };

    /* end cyclic processing loop */
}
```

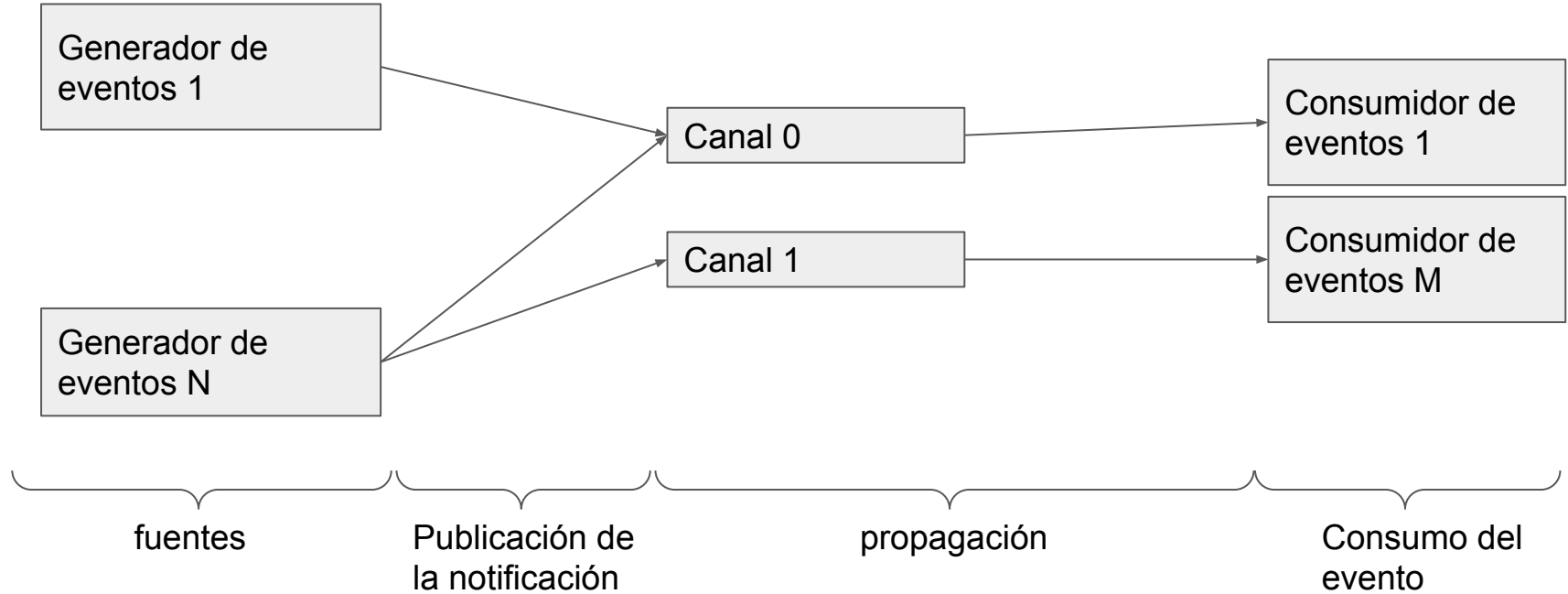
Repaso: Enfoque RTOS

- Problemáticas conocidas:
 - Condiciones de carrera
 - Deadlocks
 - Starvation
 - Inversion de Prioridades.
 - Otras
- Implementar un sistema reactivo en RTOS implica:
 - Bloquear la tarea que maneja un (o algunos) evento (s).
 - Dicha tarea no puede procesa más eventos. La codificación que viene posterior al bloque es la encargada de manejar el evento.
 - Es difícil insertar un evento nuevo en un sistema ya funcional, por lo que limita la escalabilidad.
 - Si se quisieran usar muchas tareas, por ahí no hay recursos para ello.

Sistemas reactivos.

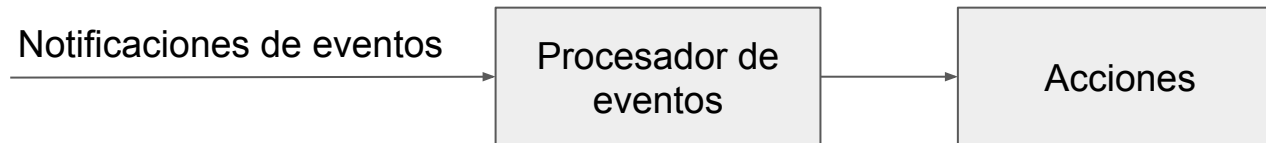
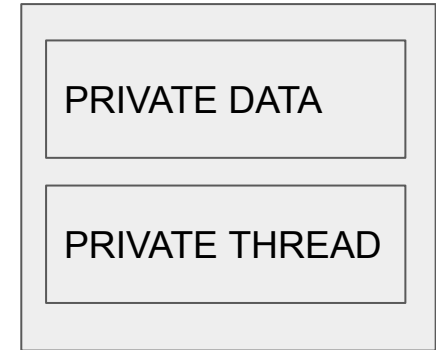
- Son sistemas que reaccionan a estímulos (eventos) que puede cambiar el estado de un objeto.
 - Eventos Internos : señales internas (ej: buffer full)
 - Eventos Externos : ISR
- Estos estímulos podrían disparar un acción dependiendo del contexto (estado) en el que se encuentre el sistema.
- Este conjunto de acciones define su comportamiento dinámico.
- Estas reacciones generan eventualmente transición de estados.

Arquitectura basada en eventos



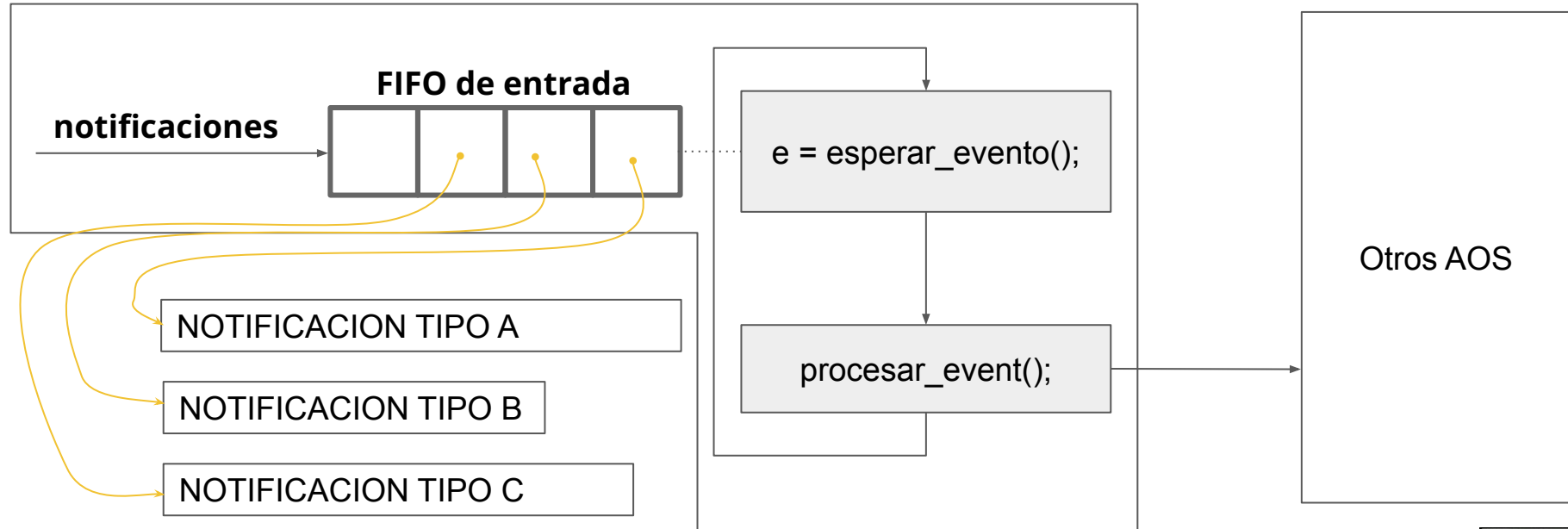
Objeto Activo

- Es un patrón de programación utilizado en sistemas con ejecución orientada a eventos.
- Encapsula un grupo de objetos que corren en su propio hilo de ejecución y se comunican entre sí de manera asincrónica, intercambiando eventos.
- Responsabilidades:
 - Coordinar el despacho de eventos.
 - Vincularse con el RTOS subyacente (si lo hubiera)
- Su comportamiento puede o no representarse por una máquina de estados.

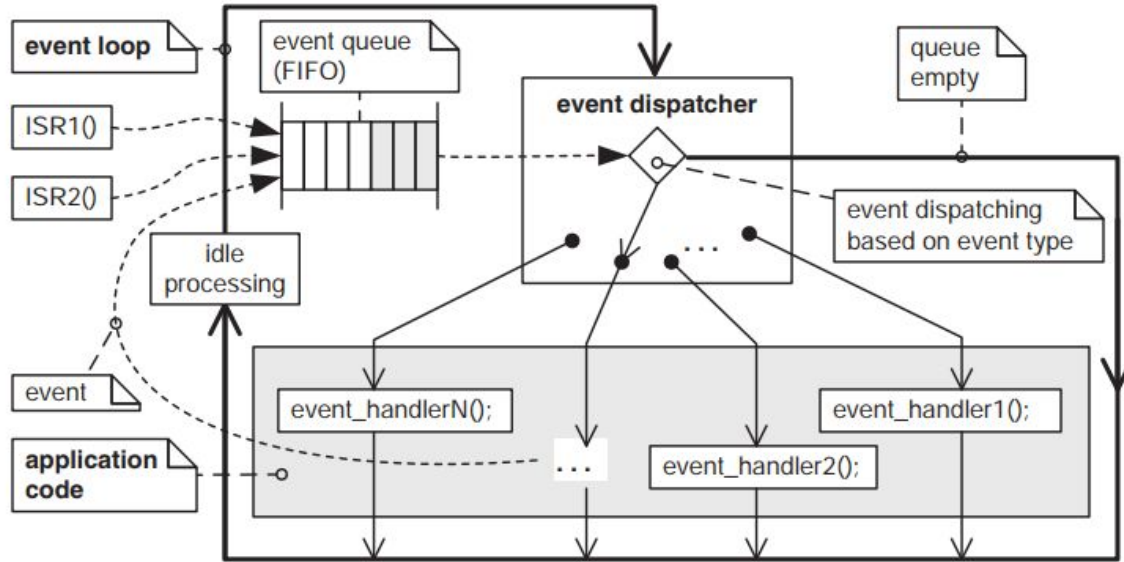


Objeto Activo

- Objeto Activo = hilo de ejecución y control + cola de eventos + procesamiento de notificaciones (maq. de estado u otra)
- Mientras no ocurran eventos, el sistema está en reposo.



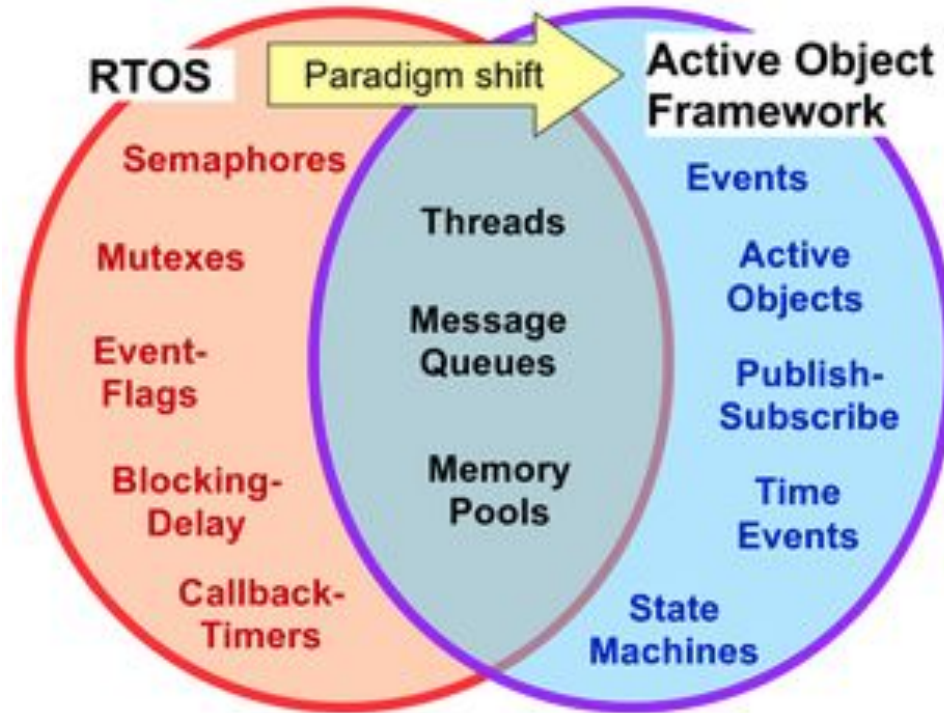
Arquitectura "Lazo de eventos"



```
void thread(void* param)
{
    ao_t* me = (ao_t*) param;

    while(1)
    {
        evnt_t* evnt;
        wait_event(ao->queue , &evnt);
        process_event(&evnt);
    }
}
```

Cambio de paradigma



Modelo de ejecución "Run To Completion"

- Cuando se detecta el evento, se despacha, y luego sus acciones son procesadas, una por vez.
- El concepto de bloqueo no existe.
 - Se utiliza el paradigma de "inversión de control". En vez de bloquear esperando un evento, las acciones se ejecutan hasta finalizar, y el objeto vuelve al estado de reposo.

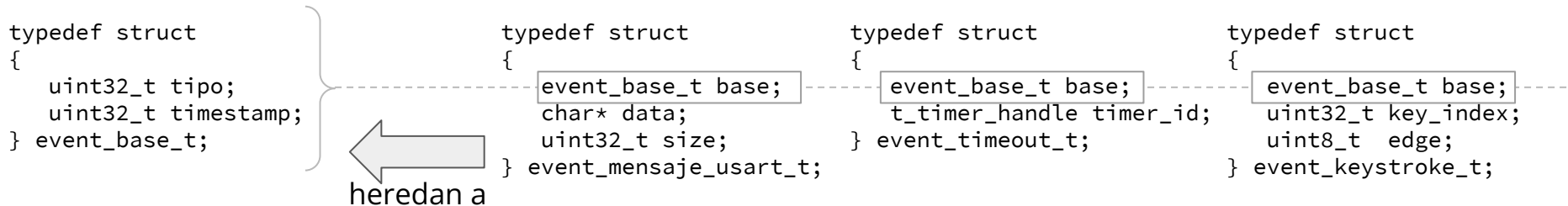
```
void accion_1(evnt_t *evento)
{
    guardar_en_sd( evento->data , evento->size );

    /* hacer otras cosas*/

    /* fin */
}
```

Evento → notificación de evento

- La notificación de un evento es un objeto y está caracterizado por un handle
- ¿Qué datos "lleva" la notificación ?
 - Encabezado : estructura "estandarizada" de la notificación
 - Cuerpo : estructura variable de la notificación
- Las "notificaciones de eventos" abstraen diferentes entidades, según la necesidad o naturaleza del driver/biblioteca que desea implementarse. Pueden ser, entre otros: datos recibidos, mensajes recibidos, señalizaciones entre OAs, etc.



Evento ➡ notificación de evento

- La notificación puede ser dinámica o estática:
- La estática no lleva información variable.

Ej	<pre>typedef struct { uint32_t code; } event_estatico_t;</pre>	<pre>const event_estatico_t eventos[CANT_EVENTOS]= { [TECLA_UP] = TECLA_UP_CODE, [TECLA_DOWN] = TECLA_DOWN_CODE, ... };</pre>	<pre>event_post(&oa1, &eventos[TECLA_UP]); event_post(&oa2, &eventos[TECLA_UP]);</pre>
	DEFINICIÓN DE TIPO	INSTANCIACIÓN DE OBJETOS	ENVIO DE EVENTO A DOS OA

- La dinámica lleva información asociada.

Ej	<pre>typedef struct { uint32_t code; uint32_t timestamp; } event_tecla_t;</pre>	No hay	<pre>event_tecla_t* evnt = pedir_bloque(); evnt->code = TECLA_UP_CODE; evnt->timestamp = xTaskGetTickCount(); event_post(&oa1, evnt); event_post(&oa2, evnt);</pre>
	DEFINICIÓN DE TIPO	INSTANCIACIÓN DE OBJETOS	ENVIO DE EVENTO A DOS OA

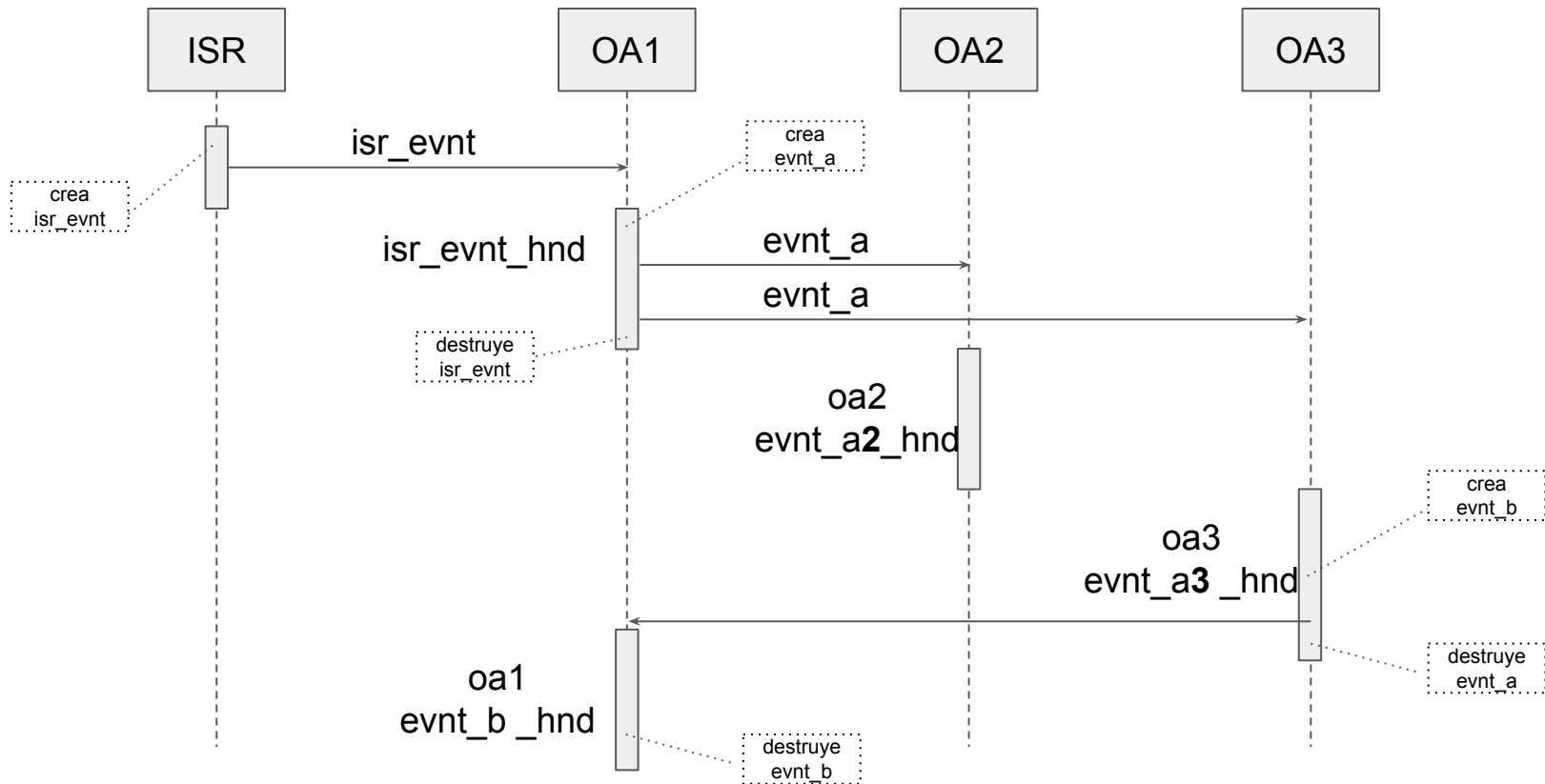
Event handlers

- Son porciones de código ejecutadas en respuesta a un evento.
- Se ejecutan uno por vez, según el comportamiento "Run To Completion"
- No se ejecuta otra acción, si la anterior no finalizó.
- No requieren mecanismo de exclusión mutua para acceder a recursos.
- No requieren que el usuario se preocupe por la sincronización de tareas.

Objeto Activo

- ¿ Cómo hace para que no haya problemas de concurrencia ?
 - Cada objeto activo es "dueño" de un grupo de recursos, y ningún otro tendrá acceso directo.
 - Solo el OA tiene acceso a ellos.
 - El OA es el gestor de estos recursos.
 - El único vínculo entre todos los OAs, serán los eventos asincrónicos.
 - Los recursos solo se utilizaran en un contexto único.
 - Los recursos están "aislados" en cada hilo.
 - Un mismo evento puede enviarse a varios OA, en cuyo caso, el objeto alocado posee un mecanismo de reference counting, con dealocación automática.

Dinámica de las notificaciones: Ejemplo



Gestión de memoria y eventos.

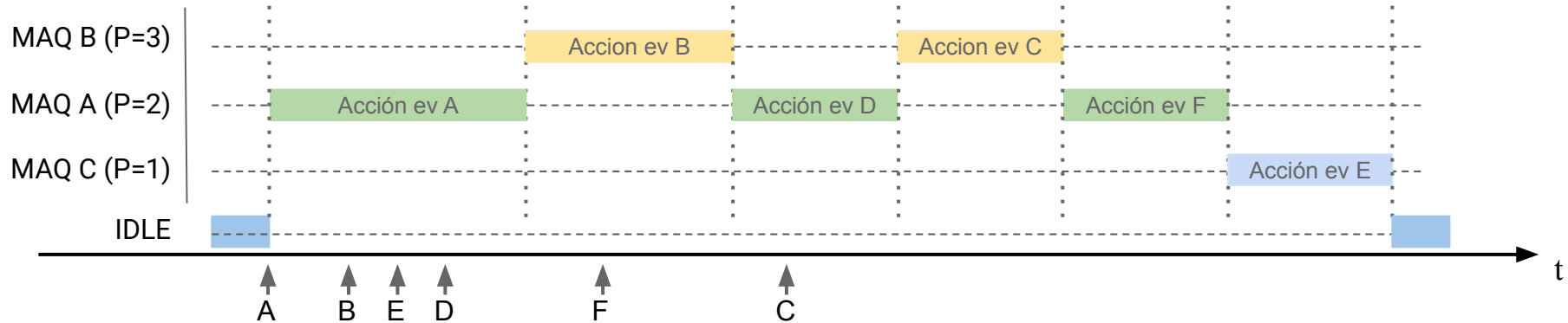
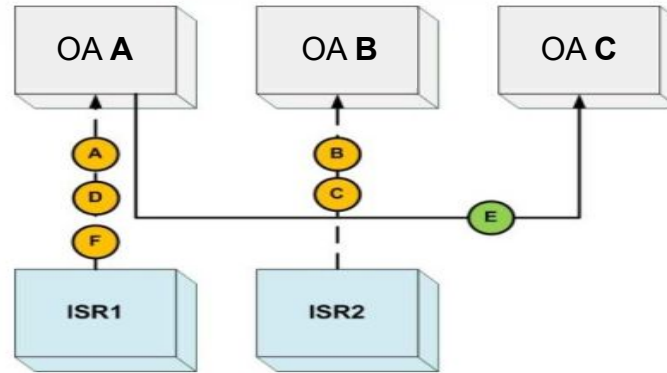
- Si un OA envía un objeto envía un evento dinámico a otros dos OAs, ¿Cómo se gestiona la liberación de memoria dinámica ?
- El framework debe gestionar la memoria que consumen los eventos de manera de proteger el contenido de los mismos hasta que todos los OAs lo hayan consumido.
- Multicast event con **reference counting**:
 - Cada evento posee un contador de la cantidad de destinos (OAs) que tiene asignado. Cada OA lo utiliza, ejecutando sus acciones, y cuando las acciones acaban, el framework, decrementa el contador y al llegar a cero, libera al evento y por consiguiente a la memoria asignada.

"Run To Completion" sin RTOS

- El lazo de eventos corre sobre un motor cooperativo.
- ¿ Porque usarlo ?
 - No se necesita apropiatividad del CPU para ejecutar otras tareas.
 - La plataforma de hardware no posee tantos recursos.
- Los únicos problemas de concurrencia posibles son entre el hilo principal y la ejecución de los handlers de interrupción.

```
infinite loop
{
    disable interrupts;
    if( is_active_object_ready_to_run )
    {
        find the active object with highest priority;
        enable interrupts;
        e = get the event from the active object's queue;
        dispatch the 'e' event to the active object's state machine;
        recycles event 'e';
    }
    else
        execute the idle processing;
}
```

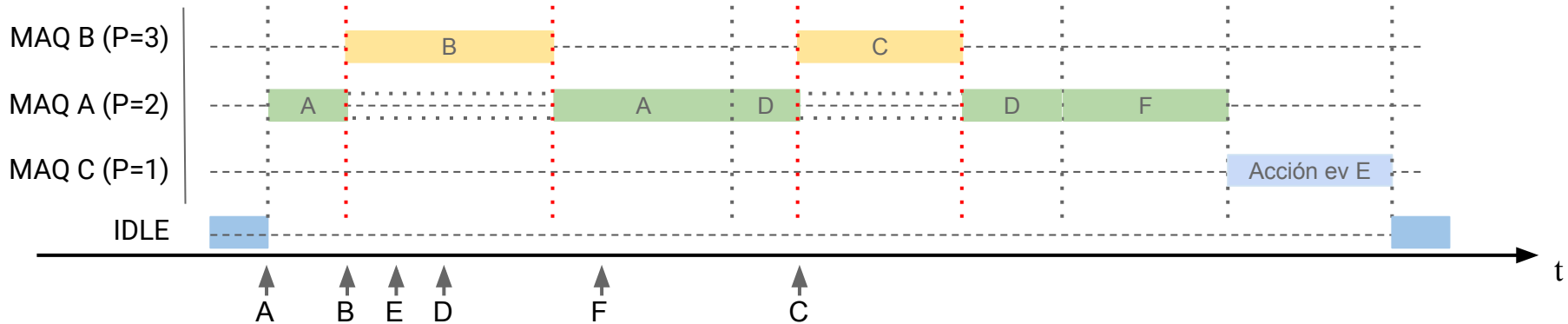
"Run To Completion" sin RTOS



"Run To Completion" en RTOS

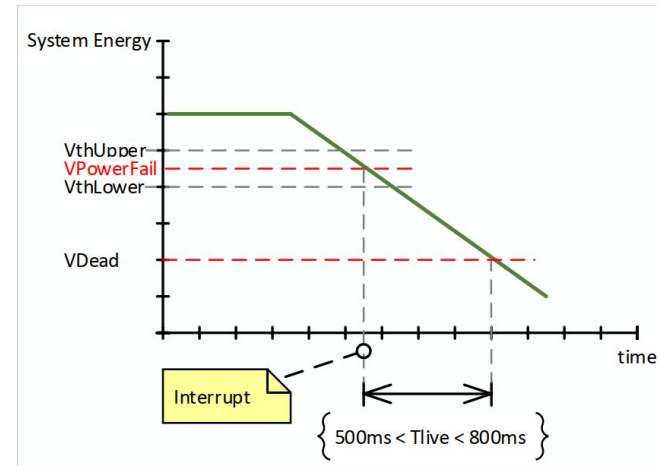
- Cada OA ejecuta su actividad dentro de una tarea del RTOS.
- Las acciones de cada máquina de estados SI pueden ser interrumpidas por una de mayor prioridad.

```
thread_loop
{
    running = 1;
    while( running )
    {
        e = get the event from the active object's queue;
        dispatch the 'e' event to the active object's state machine;
        recycles event 'e';
    }
    remove active object from the framework;
    delete thread;
}
```



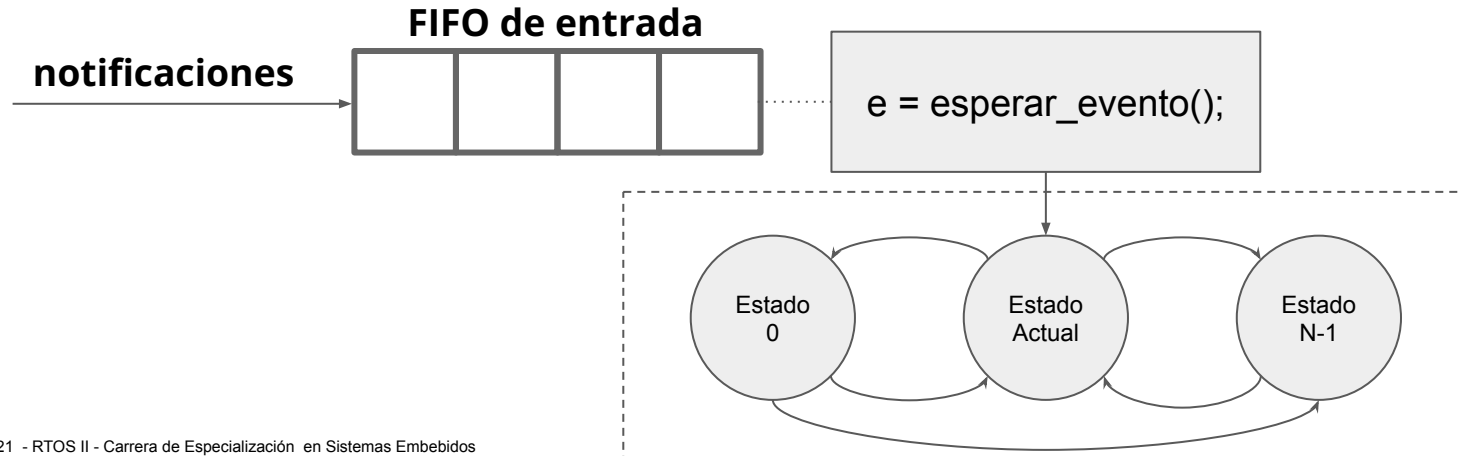
Arquitectura "Lazo de eventos": Problemáticas

- Si no se utiliza un RTOS, no es buena para un sistema de tiempo real.
 - Los "event handlers" que pudieran estar "pendientes", si o si tiene que esperar la ejecución de los que "llegaron antes" .
- Workaround:
Para resolver los "eventos importantes" podría implementarse una tarea dedicada de más prioridad que otro OA .
 - Ej: Power fail



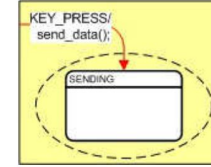
OAs y máquinas de estados

- Un OA procesa las notificaciones de eventos a través de una máquina de estados.
- De esta manera se brinda un mecanismo formal y natural para representar el comportamiento dinámico de un sistema reactivo.
 - Esta compuesto por un conjunto finito de estados y transiciones.
 - Los eventos del sistema, causarán, dependiendo del estado inicial de la máquina, una salida específica y/o una transición entre estados.



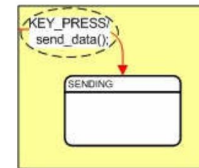
Estado

- El estado es la condición en la que se encuentra un cierto sistema que es:
 - Distinguible.
 - Disjunto a otro (ortogonal)
 - Persiste durante un cierto tiempo



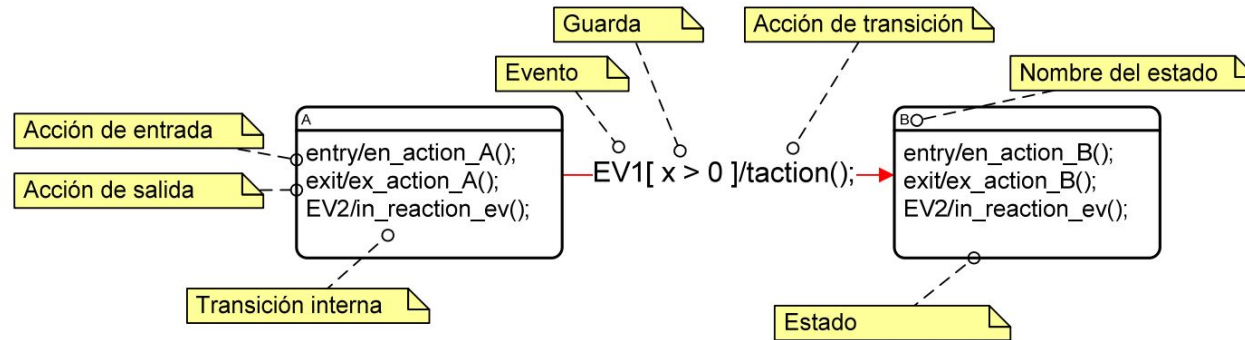
Transición

- Es la respuesta a un evento de interés que permite mover al objeto de un estado a otro.



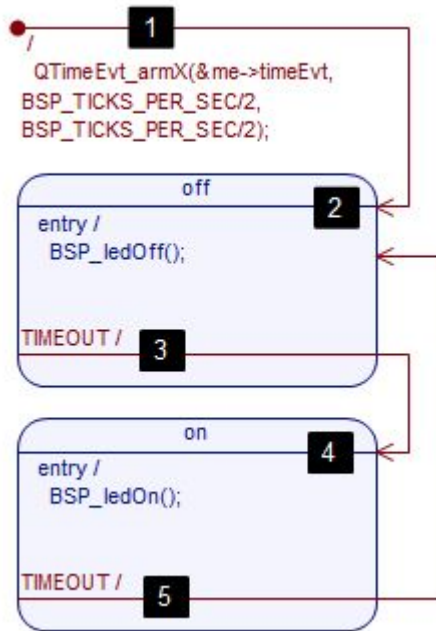
Acciones (event handlers)

- Ocurren en varias instancias del procesamiento:
 - Al salir de un estado
 - Al ocurrir una transición de estado
 - Al entrar en un estado.



Ej: Maquinas de estado y statecharts

- Blinky:



- La primera transición tiene como acción el encendido de un timer. Y el estado de destino es "off"
- La acción de entrada al estado "off" es, justamente, apagar el LED.
- El evento de timeout, dispara la transición del estado "off" al estado "on".
- La acción de entrada al estado "on", es encender el LED.
- Tal como en "off" la única transición posible que tiene el estado "on" es la de timeout.

Abstracción de objetos: ejemplos

```
typedef struct
{
    uint32_t    type;
    void*       task_hnd;
    void*       queue_hnd;
    oa_action_t* fcn;
} oa_base_t;
```



hereda a

```
typedef struct
{
    oa_base_t base;
    State_t   state;
} oa_sm_t;
```



hereda a

```
typedef struct
{
    oa_sm_t base;
    uint32_t counter;
    char    buffer[MAX];
} my_oa_sm_t;
```

```
typedef void (*oa_action_t)(oa_base_t* oa , evnt_base_t* evnt);
```

```
void oa_dispatch(oa_base_t* oa)
{
    evnt_base_t* evnt = wait_event();

    oa->fcn(oa, evnt);
}
```

```
State_t oa_sm_get_state(oa_sm_t* oa)
{
    return oa->state;
}
```

```
void oa_sm_set_state(oa_sm_t* oa , State_t s)
{
    oa->state = s;
}
```

```
void my_oa_sm_process_sm(my_oa_sm_t* oa , event_base_t* evnt)
{
    my_evnt* event = (my_evnt*) evnt;
    switch( oa_sm_get_state( oa ) )
    {
        ...
        case READING:

            if(oa->counter==MAX)
                oa_sm_set_state( oa , SENDING);
            else
                oa->buffer[oa->counter] = event->data;

            break;

        ...
    }
}
```

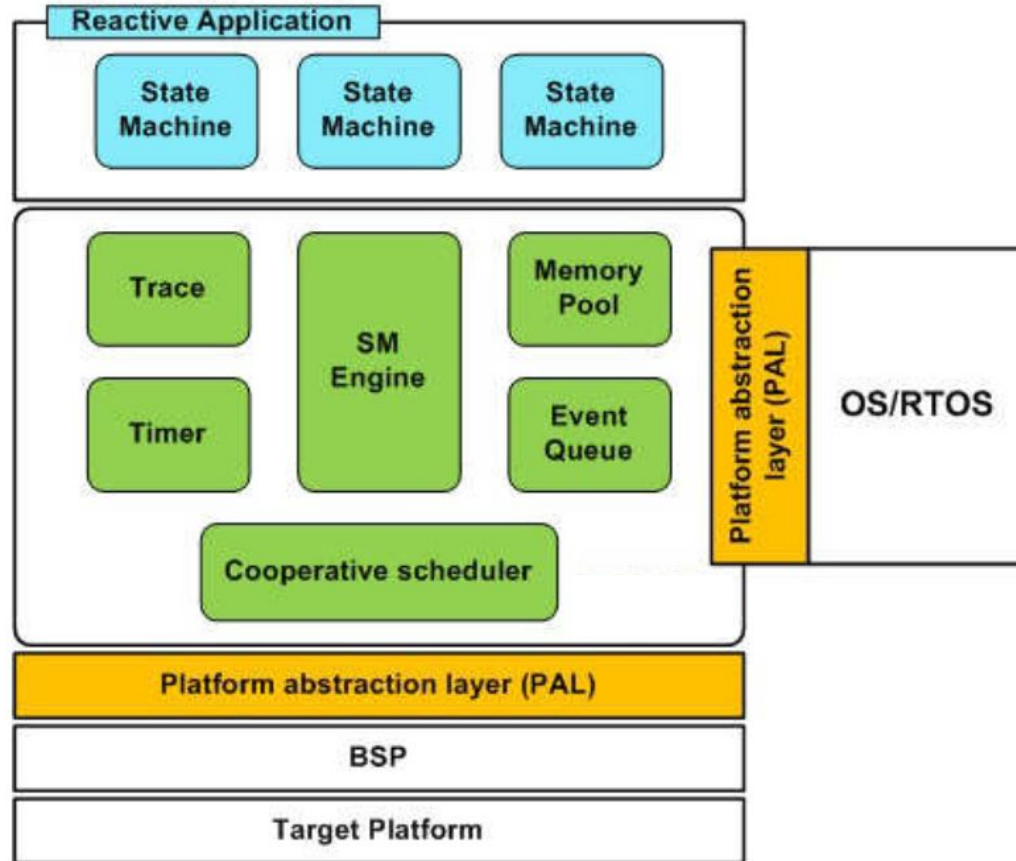
¿Por qué usar un Framework?

- La infraestructura entre aplicación del mismo tipo es la misma.
- En una nueva aplicación, entre el 60% y el 90% es código ya implementado.
 - Brinda un medio que permite la reutilización de código.
 - Facilita y agiliza el desarrollo de aplicaciones robustas.
 - Permite al desarrollador concentrarse en los problemas del dominio de la aplicación.



Framework para sistemas Reactivos

- Permiten:
 - Anidamiento de estados (máquinas de estado jerárquicas)
 - Múltiples máquinas de estados operando concurrentemente.
 - Inicialización de estados
 - Transiciones condicionales (guardas)
 - Acciones de entrada y salida de cada estado
 - Pseudoestados
 - En general poseen su propio kernel, pero pueden portarse a un RTOS sin mayores inconvenientes.

Framework ¿donde encaja?



Frameworks y Software de Modelado

		Modelado			
		<u>Yakindu</u>	<u>Papyrus</u>	<u>QM modeling tool</u>	<u>UML Designer</u>
Frameworks	<u>RKH</u>				
	<u>QPC</u> <u>QPC++</u> <u>QPnano</u>				



Generación de código en C o C++

Bibliografia

- Frameworks basados en eventos, Franco Bucafusco, CESE 2020
- [EL paradigma de la programacion dirigida a eventos, Leandro Francucci, SASE 2017](#) consultado 2020-09-19
- [Inversion of Control](#), consultado 2018-09-23
- [Simple Blinky Application](#), consultado 2018-09-23
- [MBED Event Driven Framework](#) , consultado 2018-09-25
- [Object Oriented Programming in C](#), App note. Rev H, 2018
- [Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems, 2nd Edition, Miro Samek](#)
- [SOLID](#)

Licencia



"Sistemas reactivos"

Por Mg. Ing. Franco Bucafusco, se distribuye bajo una [licencia de Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/)