Carrera de Especialización en Sistemas Embebidos

Sistemas Operativos en Tiempo Real II

Clase 3: Device drivers en RTOS

ACSE
Asociación Civil para la Investigación, Promoción y Desarrollo de los Sistemas Electrónicos Embebidos
FACULTAD DE INGENIERIA
Universidad de Buenos Aires
CIM
CADIEEL

Conceptos

- Handle:
  - Es la referencia abstracta a un recurso:
    - Pueden estar implementados como punteros, pero también pueden ser identificadores de otra índole (índices de tablas, por ej)
    - Ej: identifica una tarea del os, una cola, estructura de usuario, funcion, etc.

```
#include <stdio.h>
int main( )
{
    FILE *fp ;
    fp = fopen("test.c", "w") ;
    ....
    fclose(fp) ;
    ...
}
```

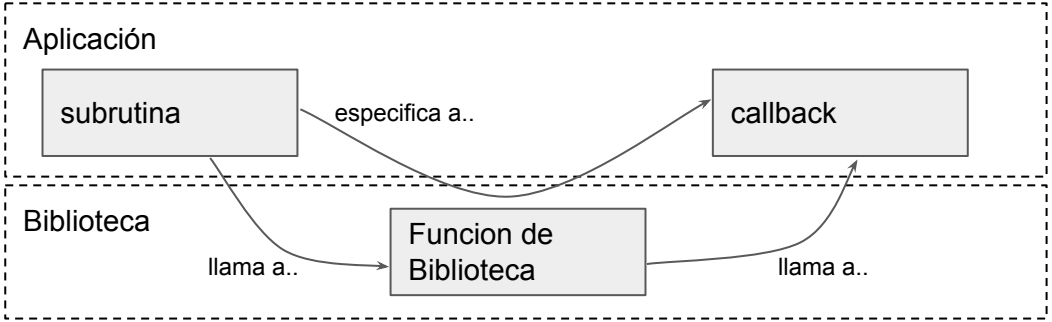
```
QueueHandle_t xQueue;

void vTask( void *pvParameters )
{
    xQueue = xQueueCreateStatic(QUEUE_LENGTH,
                                ITEM_SIZE,
                                ucQueueStorageArea,
                                &xStaticQueue );

    .....
 }
```

Conceptos

- Callback:
  - Es la referencia de una función (C) que se le pasa como argumento a otra función (F).
  - Envuelve el concepto de que una aplicación le delega la ejecución de A a la función F al llamar a F(A).
  - Sincronicos o bloqueantes: C se ejecuta antes de que finalice F
  - Asincrónicos o diferidos: C se ejecuta posterior a que finalice F (F se encarga de guardar la referencia en memoria).

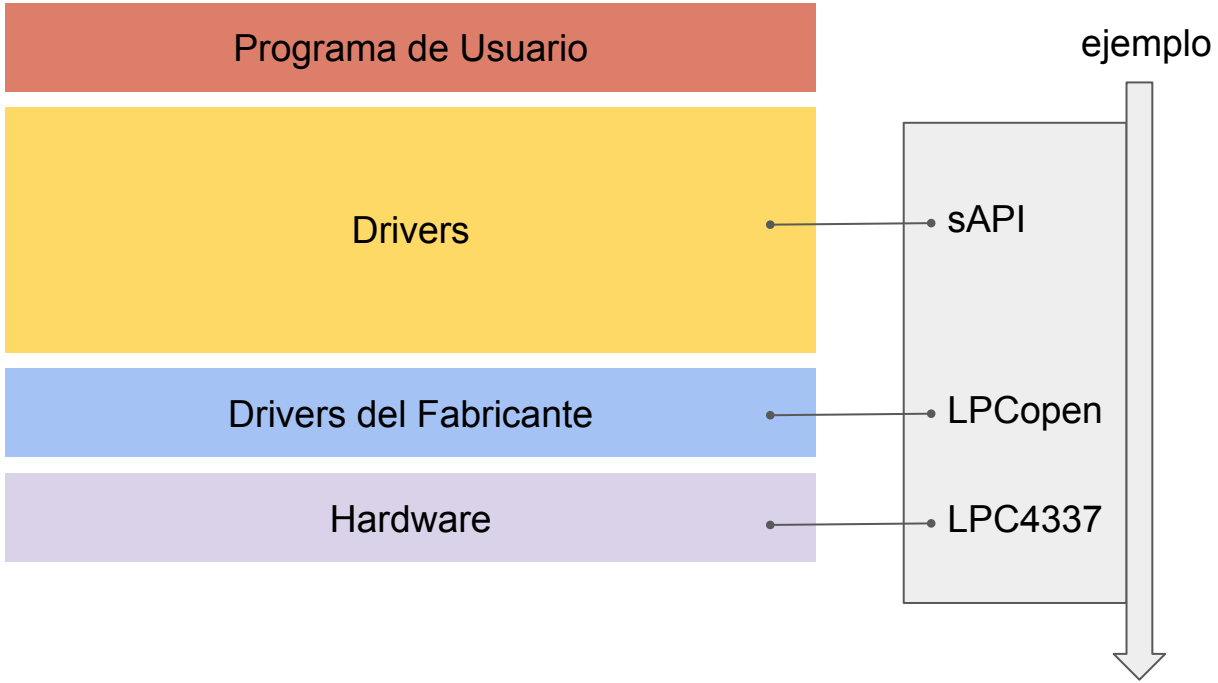


- Handler:
  - Es un caso especial de callback:
    - Event Handler
    - Interrupt Handler
    - Signal Handler
    - Exception Handler

Introducción

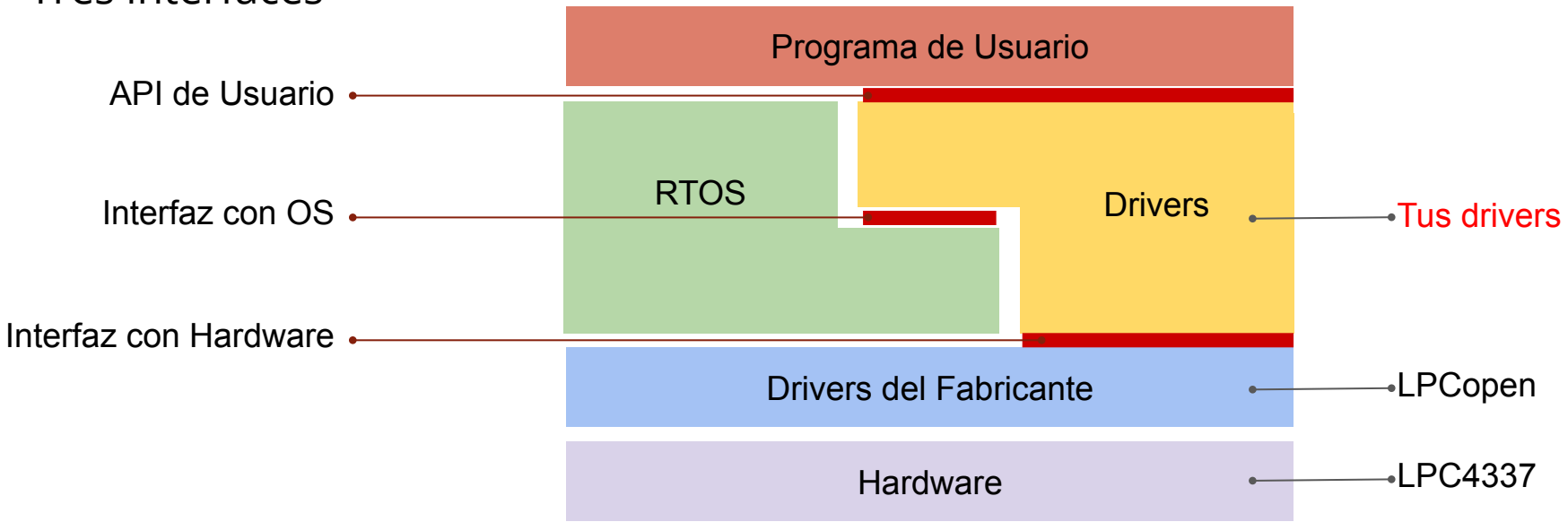
- Driver se traduce al español como "controlador" o "manejador".
- Es una porción de software que "controla" o "maneja" uno o varios dispositivos.
  - Ej: Un driver de puerto serie, puede solamente incluir a una UART pero además puede incluir DMA.
  - Ej: Un driver de chipset de radio, incluirá al periférico de comunicación (UART, I2C, SPI, etc) y además a algunas GPIO de control.
- Se escribe para darle al usuario una interfaz de programación para que utilice los dispositivos de hardware, o un abstracción de éstos, de manera amigable.
- Nos ocuparemos de drivers para ser usados con un RTOS, donde puede haber muchas tareas usando el mismo dispositivo al mismo tiempo, por ejemplo.

Drivers para una arquitectura "bare metal"



Drivers para RTOS

- Tres interfaces

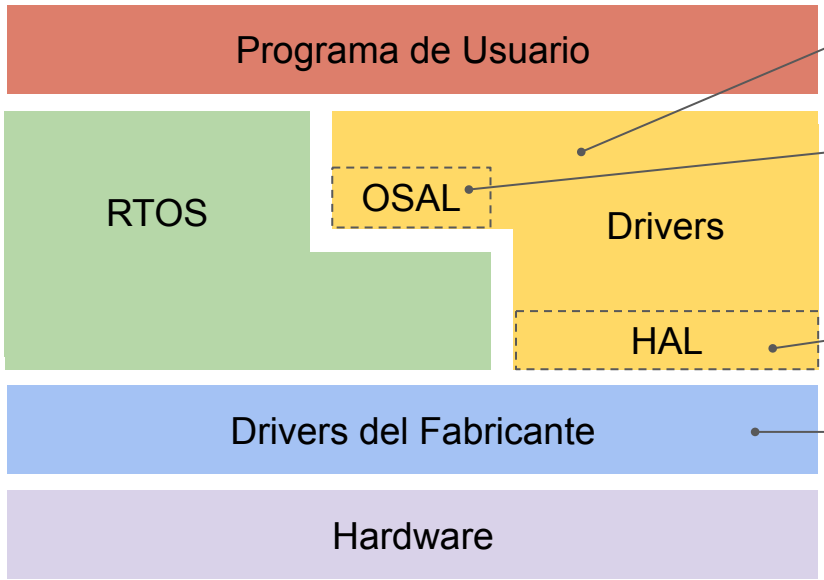


¿Es esta solución "independiente de la plataforma"?

¿Es esta solución "independiente de OS"?

Falta algo....

Drivers para RTOS



Tus drivers: Deben seguir un MODELO
(ej: UDI, HIS, WDM o LDDM)

OS Abstraction Layer

Hardware Abstraction Layer

LPCopen, CubeMx for STM32, MSP430ware, etc

Capas de abstracción

- Tanto la OSAL como la HAL pueden estar implementados por:
  - Un conjunto de macros (solo un .h)
  - Un conjunto de funciones, objetos instanciados, etc (un .h y un .c)

- La OSAL podría ser las más compleja porque, si los drivers utilizan ISRs, entonces los handlers de ISR de estos deberían estar definidos en ésta.

Ejemplo usando macros:

```
#define OS_WAIT_FOR_SIGNAL(S,T)  xSemaphoreTake( S ,T);          (PARA EL PORT DE FREERTOS)

#define OS_WAIT_FOR_SIGNAL(S,T)  SetRelAlarm( a , T , 0 );\      (PARA EL PORT DE OSEK)
                                 WaitEvent( S );

#define OS_WAIT_FOR_SIGNAL(S,T)  S.wait(T)                       (PARA EL PORT DE MBED, en c++)

#define OS_WAIT_FOR_SIGNAL(S,T)  chSemWaitTimeout(&S, T);        (PARA EL PORT DE CHIBIOS, en c)
```

Capas de abstracción: otro ejemplo

os_port.h

```
#define OS_WAIT_FOR_SIGNAL(S,T_MS)  int32_t my_os_al_sem_wait(S, T_MS)
```

my_os_al.h

```
int32_t my_os_al_sem_wait(os_sem_id_t semaphore_id, uint32_t millisec);
```

my_os_al.c

```
int32_t my_os_al_sem_wait(os_sem_id_t semaphore_id, uint32_t millisec)
{
    portBASE_TYPE taskWoken = pdFALSE;
    TickType_t ticks = 0;
    if (millisec == osWaitForever) ticks = portMAX_DELAY;
    else if (millisec != 0) ticks = millisec / portTICK_PERIOD_MS;

    if (inHandlerMode())
    {
        if (xSemaphoreTakeFromISR(semaphore_id, &taskWoken) != pdTRUE)
        {
           return my_os_al_error;
        }
        portEND_SWITCHING_ISR(taskWoken);
    }
    else if (xSemaphoreTake(semaphore_id, ticks) != pdTRUE) {
        return my_os_al_error;
    }
    return my_os_al_ok;
}
```

API de Usuario básica

- Típicamente la API de usuario posee al menos estos métodos (o de similar función) que definen actividades.

Inicializar(obj)	Inicialización de periférico(s), inicialización de estructura de datos, configuración (ej: pwmInit de sapi_pwm.c)
Leer(obj)	Obtención de datos del driver (ej, pwmIsAttached de sapi_pwm.c)
Escribir(obj)	Escritura de datos al driver (ej: pwmWrite de sapi_pwm.c)

- Si el uso del driver está orientado a "la sesión" se implementan estos métodos:

Abrir(obj)	Habilita al driver para ser utilizado (ej: Timer_Init de sapi_timer.c)
Cerrar(obj)	Deshabilita al driver para ser utilizado. (ej: Timer_DeInit de sapi_timer.c)

¿Cómo implemento un driver sobre un RTOS?

- El acceso al los periféricos deberá ser de exclusión mutua:
  - Deberá poder "utilizarlos" una tarea a la vez.

```
void Tarea_A()
{
   char duty = 100;

   while(1)
   {
      PWM_Cambiar_Duty(duty);
      /* otras acciones */
   }
}
```

```
void Tarea_B()
{
   uint16_t frec = 10000;

   while(1)
   {
      PWM_Cambiar_Frecuencia(frec);
      /* otras acciones */
   }
}
```

- Se debe incluir el mutex en el driver mismo para ocultarlo de la aplicación.

```
void PWM_Cambiar_Frecuencia(uint16_t frec)
{
   mutex_lock(pwm_driver.mutex)
   pwm_driver.frec = frec;
   HW_CAMBIAR_FREC(pwm_driver.frec);
   mutex_unlock(pwm_driver.mutex)
}
```

¿Cómo implemento un driver sobre un RTOS?

- El acceso a los objetos de RAM del driver deberán realizarse de manera atómica.
  - Implementar al driver "Thread safe" usando métodos vistos en RTOS 1
- Puede hacerse con un mutex, habilitado/deshabilitado ISR, o lo que convenga en cada caso.

```
void usart_add_byte_to_buffer( driver_usart_t *obj , char new)
{
   ___ENTER_CRITICAL(obj);
   if( obj->i < obj->max )
   {
      obj->buf[obj->i] = HW_LEER_BYTE_RECIBIDO(obj->usartx);
      obj->i ++;
   }
   else
   {
      /* algoritmo buffer full*/
      OS_SEND_SIGNAL(obj,DRV_BUFF_FULL);
   }
   ___EXIT_CRITICAL(obj);
}
```

Orientado a objetos

Abre sección crítica (HAL/OSAL)

Utilización de la HAL

Utilización de la OSAL

Cierra sección crítica (HAL / OSAL)

Preguntas importantes

- ¿ Queremos que la tarea que llamó a una función de la API driver espere el resultado de la operación ?

- ¿ Queremos que la tarea haga otro trabajo mientras la operación es procesada por el driver?

- La arquitectura del driver será muy diferente según estas alternativas.

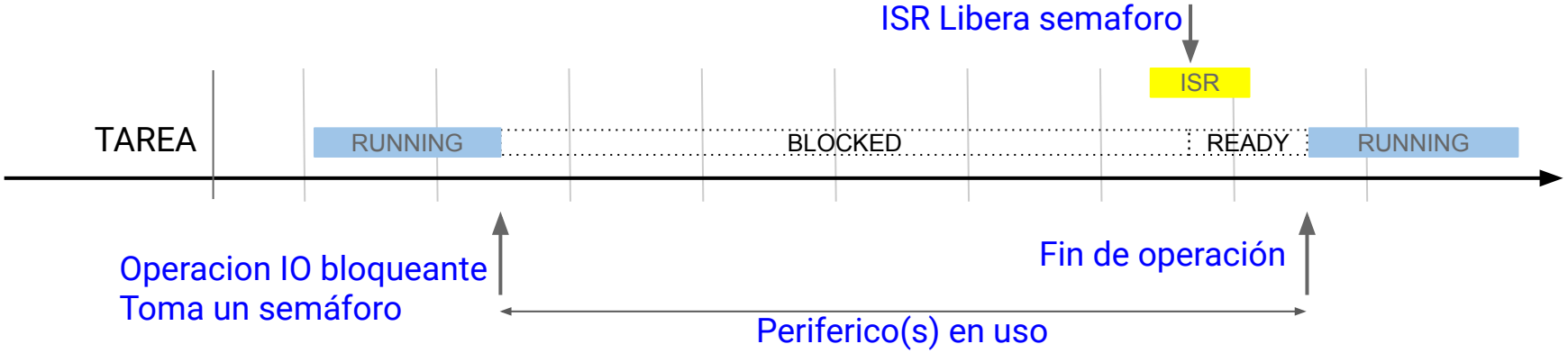
Complejidad

- Un Driver de I/O está compuesto al menos por dos partes.
  - Una mitad superior, de interfaz con el usuario (API). Es así como recibe las operaciones que debe ejecutar y devuelve los resultados de las mismas.
  - Una mitad inferior, de interfaz con el hardware que maneja. Atiende las interrupciones del dispositivo.

Arquitectura: Drivers sincrónicos

- Se llaman así porque la aplicación, al solicitar una operacion al driver, esperará el resultado.
- Solo la tarea que invocó al driver deberá esperar, el resto sigue haciendo otro trabajo.
- Esto simplifica mucho el diseño del driver y la aplicación que lo usa.
- Puede usar un semáforo o una cola para señalizar el fin de la operación.
  - Bloquea a la tarea que le solicite una operación mientras el driver está ocupado.

Driver sincrónico básico




```
uint8_t  driver_x_get( driver_x_t *obj  )
{
   uint8_t resultado;

   os_mutex_lock( obj->mutex );
   HW_INICIAR_OPERACION( obj->peripheral_instance );
   os_semaphore_wait( obj->sem_operation_complete );
   resultado = HW_GET_RESULT( obj->peripheral_instance );
   os_mutex_unlock( obj->mutex );

   return resultado;
}
```

Drivers sincrónicos

- Pros
  - El programa se puede escribir secuencialmente (más simple).
  - Es más fácil manejar el resultado de la operación (el CPU está “en el lugar correcto en el momento adecuado”).
- Contras
  - Una tarea bloqueada no puede responder a otros estímulos ⇒
    Me obliga a una arquitectura con múltiples tareas.
  - Puede causar inversión de prioridades.

```
void tarea()
{
   char array[20];
   uint8_t size;

   /* init */
   driver_init( &driver_instance_a );

   while(1)
   {
      size = driver_get_new_value( &driver_instance_a , array );
      /* usar los "size" bytes que hay en array */
   }
}
```

Drivers Asincrónicos

- La tareas delegan en el driver la responsabilidad de que éste les indique cuando hay datos nuevos.
  - Se pueden encargar múltiples operaciones sin esperar el resultado de la primera.
- Cuando es necesario manejar el resultado de las operaciones es requerido un mecanismo.
- Pros:
  - No me obligan a tener tareas dedicadas a la operación bloqueante.
    - Las tareas pueden hacer más de una solo cosa.
  - Son necesarios para arquitecturas orientadas a eventos.

Drivers Asincrónicos : Complejidad

- Cada vez que el dispositivo tiene una novedad genera una interrupción y atrae la atención del CPU:
    - Se debe acceder al hardware para obtener el dato
    - El dato de debe almacenarse en alguna zona de memoria (a veces alcanza con la memoria del periférico, pero en general no es así)
    - Cuando el driver considera que debe informar, genera una señal.
        - Hay varias opciones:
            - Semáforos o Colas
            - Callbacks
    - Si no se desea que el driver genere señales, deberá poseer una interfaz de usuario para que el mismo pueda consultar por datos nuevos.

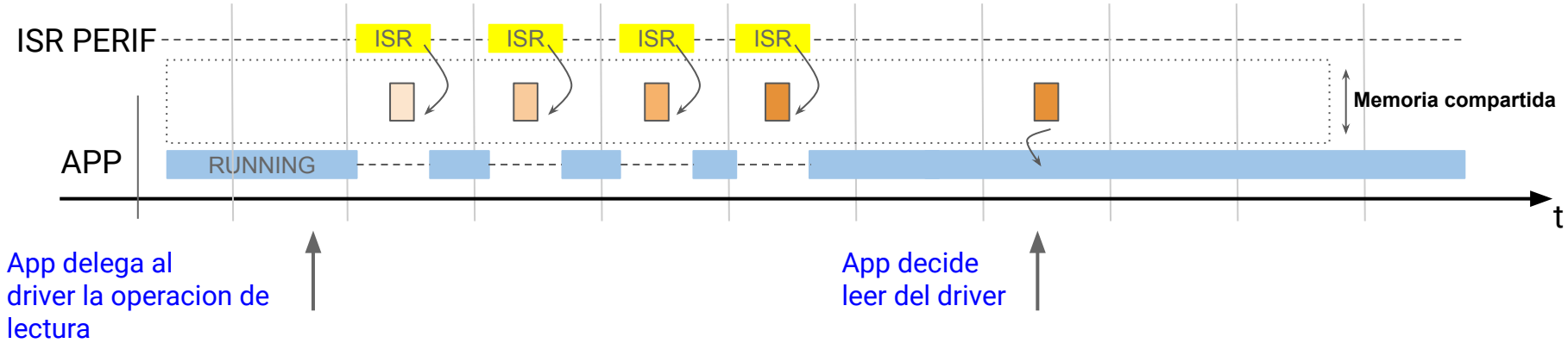
Drivers Asincrónicos : Uso de la memoria

- Al dividir el driver en dos mitades desacopladas (métodos de API para el usuario, métodos internos del driver y métodos de ISR) es necesaria una zona de memoria en común.
- Lo más simple es tener un buffer de entrada y uno de salida, en una zona de memoria compartida.
- La aplicación copia al buffer de salida los datos a transmitir y lee del buffer de entrada los datos recibidos.

Caso: Entrada única sin memoria

- Cada vez que hay una entrada se la guarda en una zona de memoria compartida.

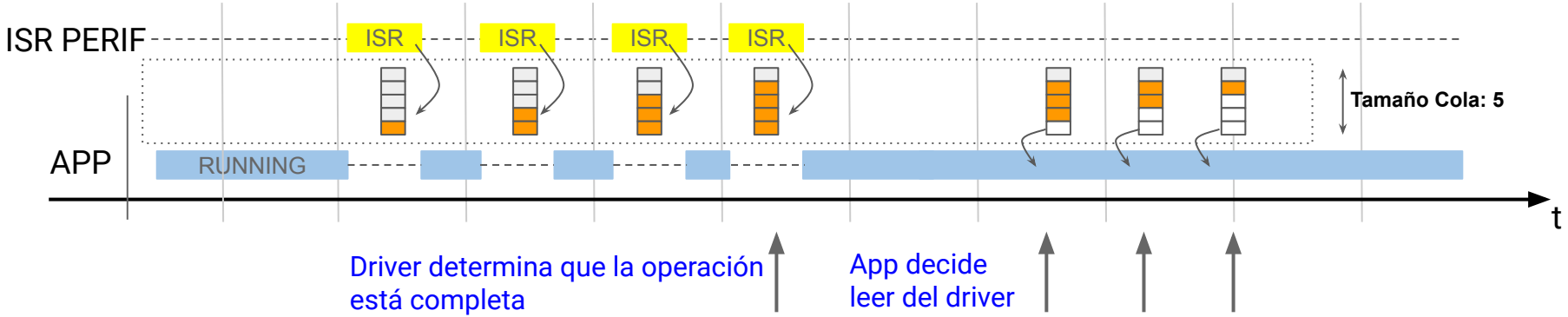
- Se debe proteger el recurso compartido vía Mutex.



- Solo tiene sentido si lo importante es el resultado más actual

Caso: Entrada única con memoria

- Cada vez que hay una entrada se la guarda en una cola de mensajes.
- En promedio, hay que procesarlas a la misma tasa que se generan o se encolarán muchos datos viejos.



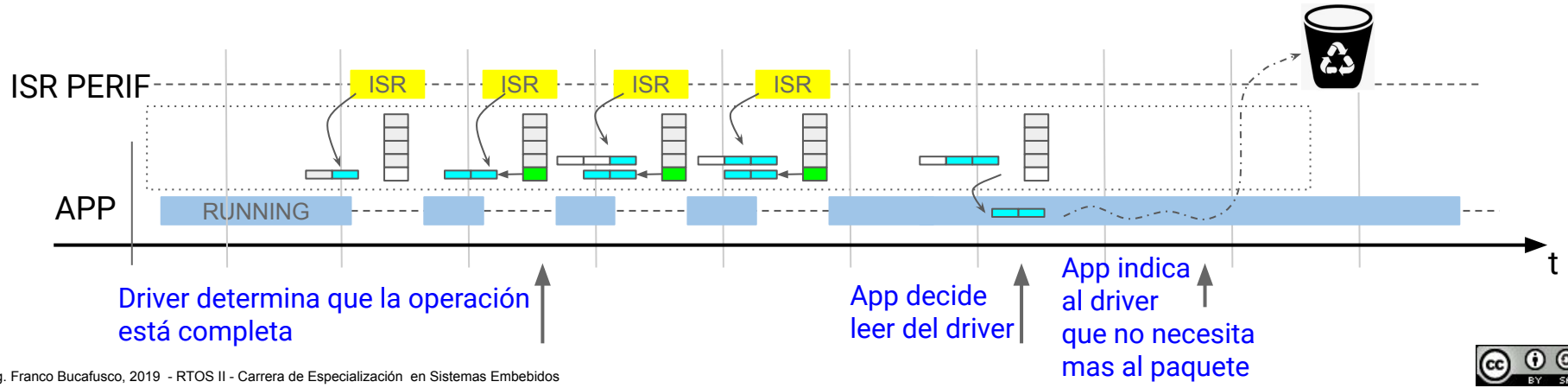
- Tiene sentido cuando procesar un resultado a destiempo no es un problema.

Caso: Encolador de entrada serial

- Si los datos recibidos son paquetes grandes es mejor pasar punteros a ellos.
- Se ahorra tiempo evitando hacer copias de la memoria.

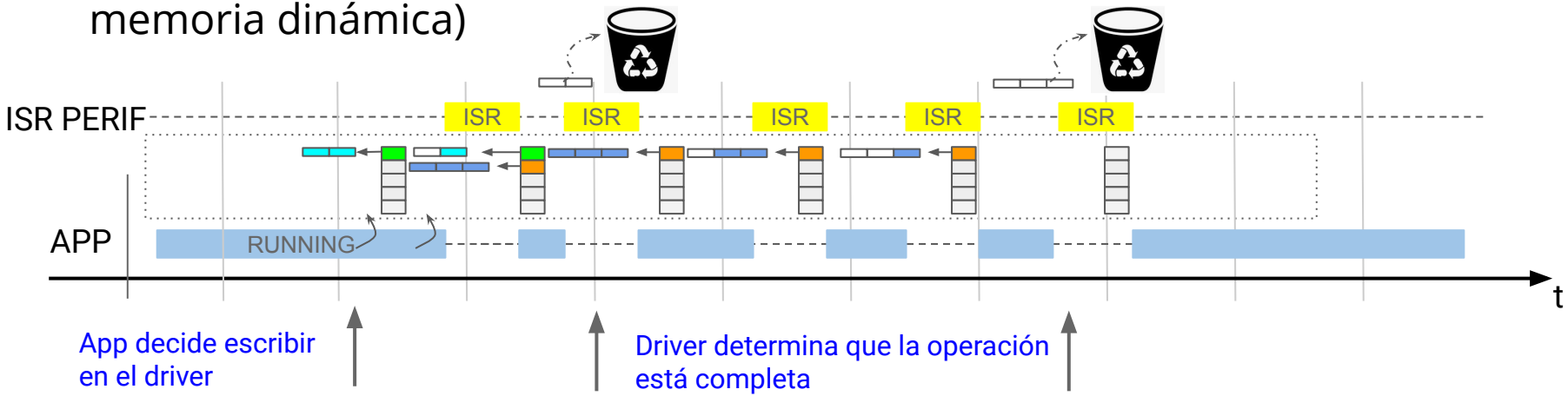
```
driver_read:
- Obtiene msg de cola.
- Procesa el mensaje
- Libera bloque de memoria
```

```
driver_process_isr:
- Obtiene dato del HW.
- Si se encuentra recibiendo, coloca el dato
en memoria
- Si no, pide buffer y coloca el dato
- Agrega una ref. del msg en la cola.
```



Caso: Encolador de salida serial

- El mismo caso pero en otro sentido: La app encarga operaciones al driver.
- Las interrupciones se ocupan de quitar los datos de la cola de mensajes y de enviarlos al dispositivo.
- Se debe tener mucho cuidado al caso de transmitir el último dato !
  - No vienen más interrupciones y el driver “se queda parado” (y no liberar memoria dinámica)



Bibliografía

- Writing device specific, RTOS independent , hardware independent device drivers, Brian Schrom
- Architecture of Device I/O Drivers
- Patron Reactor
- Patron Proactor
- Estrategias para drivers en RTOS, Alejandro Celery, CESE, 2019
- Operating system abstraction layer
- REACTOR, Patterns in C, Adam Petersen (PDF)
- Event Handler

Licencia

"Device drivers en RTOS"