# Carrera de Especialización en Sistemas Embebidos

# Sistemas Operativos en Tiempo Real II

## Clase 3: Estrategias para 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

# Introducción

- Driver se traduce al español como "controlador" o "manejador".
- Es un trozo de software que "controla" o "maneja" un determinado dispositivo.
- Se escribe cuando hay alguna circunstancia en particular a manejar respecto a un dispositivo cualquiera.
- Nos ocuparemos de drivers para ser usados con RTOS, donde puede haber muchas tareas usando el mismo dispositivo al mismo tiempo.

# Introducción

- El requerimiento más básico de un driver es asegurarse que una sola tarea pueda acceder simultáneamente a un dispositivo en particular.

- Se puede hacer con un mutex. Cualquier tarea que accede al driver debe tomarlo y liberarlo luego de usarlo.

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

# Preguntas importantes

- Queremos que la tarea que llamó al driver espere el resultado de la operación de I/O?

- Queremos que la aplicación haga otro trabajo mientras la operación de I/O es procesada por el driver?

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

# Drivers sincrónicos

- Se llaman así porque la aplicación esperará el resultado de la tarea que pidió, entonces la operación de IO se hace sincronizada con la aplicación que la invoca.
- 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.

# SYNCHRONOUS I/O DRIVERS

- 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
  - Un programa bloqueado no puede responder a otros estímulos (salvo que la plataforma tenga el system call select() o similar).
  - Compromete la respuesta temporal de la aplicación.
  - Hace más difícil modificar el programa.

# Driver básico

- Usa un mutex para asegurarse de que solo puede usar una tarea a la vez (Si no fuera así podría haber corrupción de datos)
  - Dos tareas escribiendo a una misma UART mezclarían sus mensajes.
  - Dos tareas escribiendo a un mismo bus I2C corromperían la transacción.

- Debilidades
  - Bloquea a la tarea que le solicite una operación mientras el driver está ocupado.
  - Me obliga a una arquitectura con múltiples tareas secuenciales.
  - Esto puede causar inversión de prioridades. MUY malo.

# Por qué usar drivers asincrónicos

- Mantienen al CPU disponible para responder a cualquier evento.
- No me obligan a tener tareas dedicadas a la IO bloqueante.
- Son necesarios para arquitecturas orientadas a eventos.
- Tienen mejor performance (evitan los cambios de contexto).

# Por qué usar drivers asincrónicos

- En una driver asincrónico, la tarea que llamó al driver puede seguir ejecutándose, sin esperar el resultado de la operación que pidió.
- Se pueden encargar múltiples operaciones sin esperar el resultado de la primera.
- Esto es verdadero paralelismo, incluso en procesadores con un solo core.
- Cuando es necesario manejar el resultado de las operaciones de IO se hace necesario un mecanismo para esto.

# Complejidad

- Un Driver de I/O está compuesto por dos partes
  - Una mitad superior, de interfaz con el usuario. 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.

# Complejidad

- Cada vez que el dispositivo tiene una novedad genera una interrupción y atrae la atención del CPU:
  - Se terminó de transmitir un dato.
  - Es tiempo de transmitir el próximo.
  - Se acaba de recibir un dato:
    - Se debe recoger el mismo, ya que los I/O devices suelen tener una memoria interna limitada.
  - Se produjo un error en la operación en curso:
    - Se suele registrar un callback de error, para que la aplicación decida de qué manera manejarlo.

# Uso de la memoria

- Al dividir el driver en dos mitades desacopladas es necesaria una zona de memoria en común.

- Lo más simple es asignar al driver 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.

# Uso de la memoria

- Este enfoque conlleva la reserva de memoria para estos buffers.
- Esta memoria debe estar protegida en el caso de programación concurrente.
- Hay que asegurarse que la memoria mantiene el mensaje hasta que haya sido enviado (o terminado de recibir).

# Uso de la memoria

- Si el driver permite encolar múltiples operaciones deben mantenerse vigentes todos los buffers de memoria asignados.

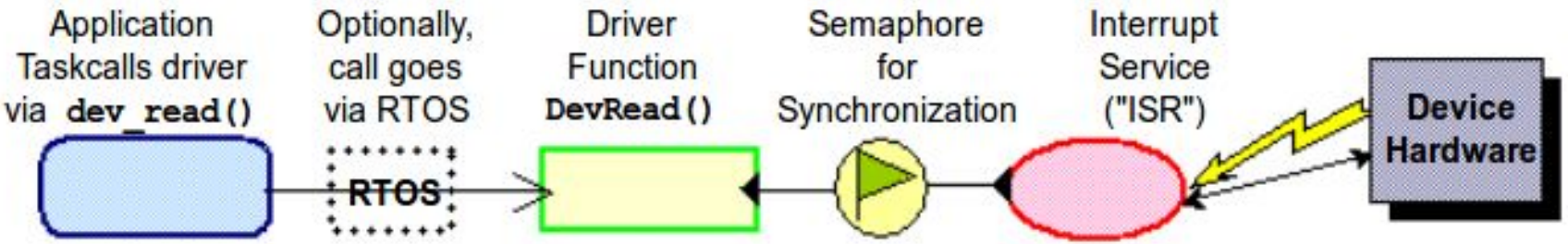
- Se puede resolver con pools de memoria.

# Ejemplos

- Driver sincrónico
- Entrada única con interrupciones
- Solo la última entrada
- Encolador de entrada serial
- Encolador de salida serial

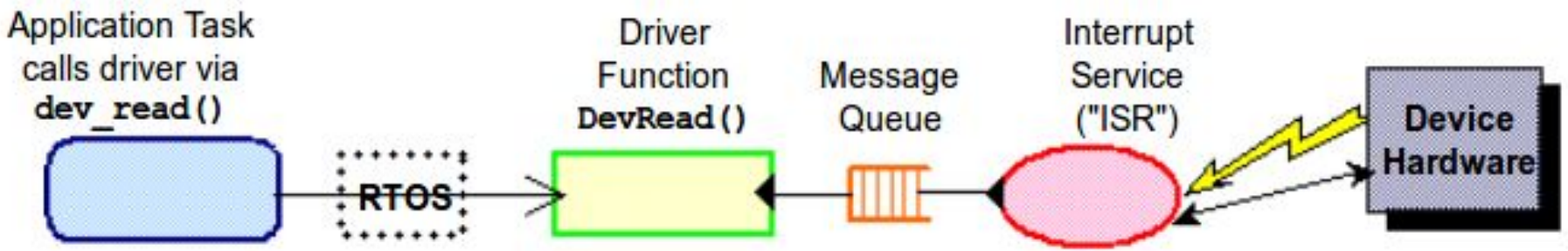
# Driver sincrónico

- El más sencillo primero



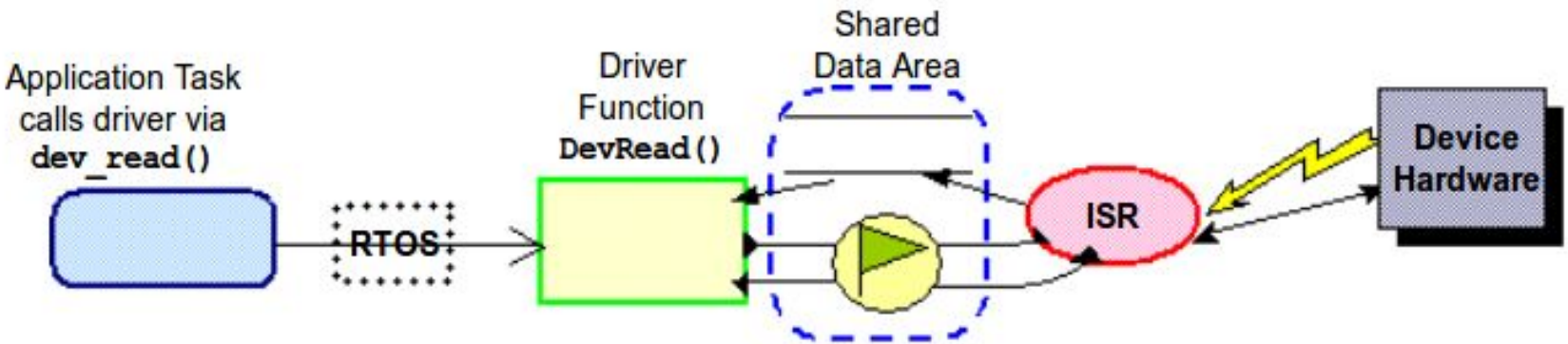
# Entrada única con interrupciones

- Cada vez que hay una entrada se la guarda en una cola de mensajes.
- Hay que procesarlas a la misma velocidad que se generan o se encolarán muchos datos viejos.
  - Esto puede ser un problema o no, según lo que sea el dato.



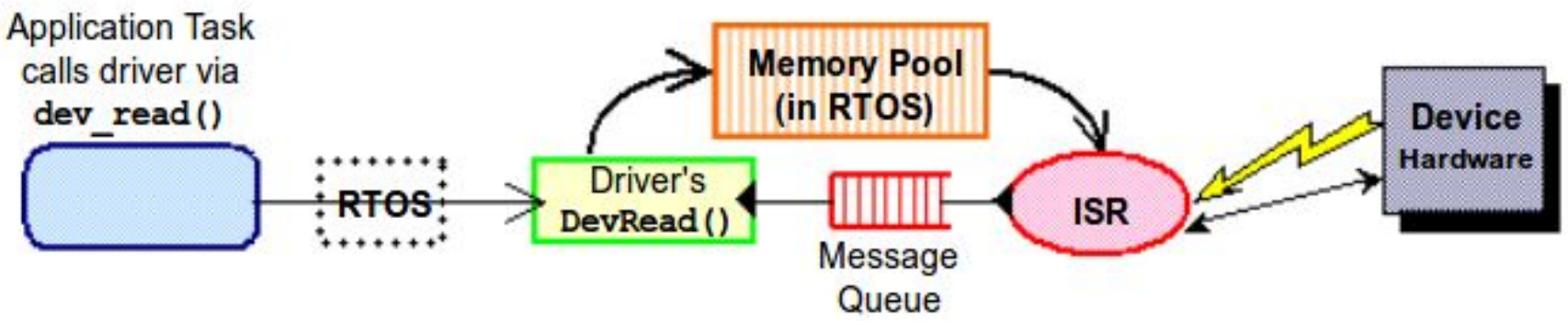
# Solo la última entrada

- En algunos casos un dato procesado a destiempo puede no servir.
- Cuando esto ocurre se lo descarta al llegar uno nuevo.
- En este caso es mejor no usar una cola de mensajes sino un área compartida de memoria.



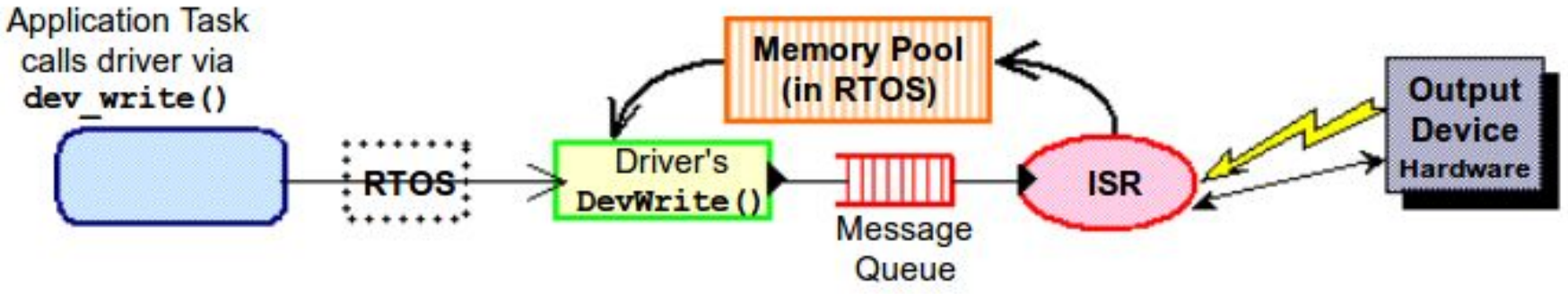
# Encolador de entrada serial

- Si los datos recibidos son paquetes grandes es mejor pasar punteros a ellos.
- Se ahorra tiempo en hacer copias de la memoria.
- En la ISR del driver se le pide al RTOS un área de memoria para guardar los datos.
- Luego se pone un puntero a esa área en una cola de mensajes para que la procese la mitad superior del driver.



# Encolador de salida serial

- El mismo caso pero para salidas.
- 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”, hay que generar una interrupción manualmente.



# Patrones de diseño avanzados

- Patrón Reactor
- Patrón Proactor
- Patrón Asynchronous completion token

# Patrón reactor

- Se usa cuando no hay soporte para operaciones asincrónicas.
- Se necesita un "event demultiplexer" que avisa a la aplicación cuando hay eventos que procesar.
- Estos eventos pueden ser datos recibidos, mensajes recibidos, etc.

# Patrón reactor

- Se puede hacer con una tarea y una cola de eventos.
- Se hace una lectura bloqueante hasta que haya un evento que procesar.
- Se despachan los eventos de forma RTC (Run to Completion).

# Patrón proactor

- Variante del reactor cuando hay soporte para operaciones asincrónicas.
- Se agregan dos componentes
  - Completion handlers: Se registra qué función se va a llamar cuando se complete la operación.
  - Completion dispatcher: Se especifica quién llamará al completion handler (puede ser un task dedicado a esto).

# Patrón proactor

- Se necesita un "procesador de operaciones asincrónicas”.
- Puede ser tan fácil como una ISR sacando datos de una cola circular y transmitiéndolos al hardware.
- La tarea que inicia la transmisión retorna inmediatamente.
- Compatible con arquitecturas orientadas a eventos.
- El iniciador debe registrar un callback a llamarse cuando se completa la operación.

# Patrón ACT (async completion token)

- A veces es necesario guardar el estado de una operación para devolverlo al iniciador.
- ACT es igual que el proactor pero se pasa un "completion token" entre los componentes.
- Es la variante más sofisticada.

# Q and A

# Bibliografía


- Architecture of Device I/O Drivers
  - http://www.kalinskyassociates.com/Wpaper4.html

- Patrón reactor
  - https://www.cse.wustl.edu/~schmidt/PDF/reactor-siemens.pdf

- Patrón proactor
  - https://www.cse.wustl.edu/~schmidt/PDF/proactor.pdf

# Further reading

- Patrón ACT
  - https://www.cse.wustl.edu/~schmidt/PDF/ACT.pdf
- Patrón chain of responsability
  - https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern
- Problema C10k
  - https://es.wikipedia.org/wiki/Problema_C10k

# Gracias por su atención!