



INSTITUTO DE  
INGENIERÍA  
ELÉCTRICA



FACULTAD DE  
INGENIERÍA  
UDELAR



UNIVERSIDAD  
DE LA REPÚBLICA  
URUGUAY

# Arquitecturas de software embebido

## Sistemas embebidos para tiempo real

Este material didáctico fue elaborado por docentes del Departamento de Electrónica de la Universidad de la República a lo largo a varios años. Se pone a disposición de la comunidad bajo la licencia “Creative Commons Attribution 4.0 International License”.

Ver detalles de la licencia aquí: <https://creativecommons.org/licenses/by/4.0/>



Co-funded by the  
Erasmus+ Programme  
of the European Union

# Objetivos

- Describir las principales arquitecturas de software.
- Identificar sus características.
- Compararlas las ventajas y desventajas de cada una.
- Seleccionar la arquitectura adecuada para una aplicación particular.

# Índice

- Introducción
- Round-Robin (procesamiento secuencial)
- Round-Robin con interrupciones
- Planificación por cola (o encolado) de funciones
- RTOS (Sistema Operativo de Tiempo Real)
- Resumen

# Introducción

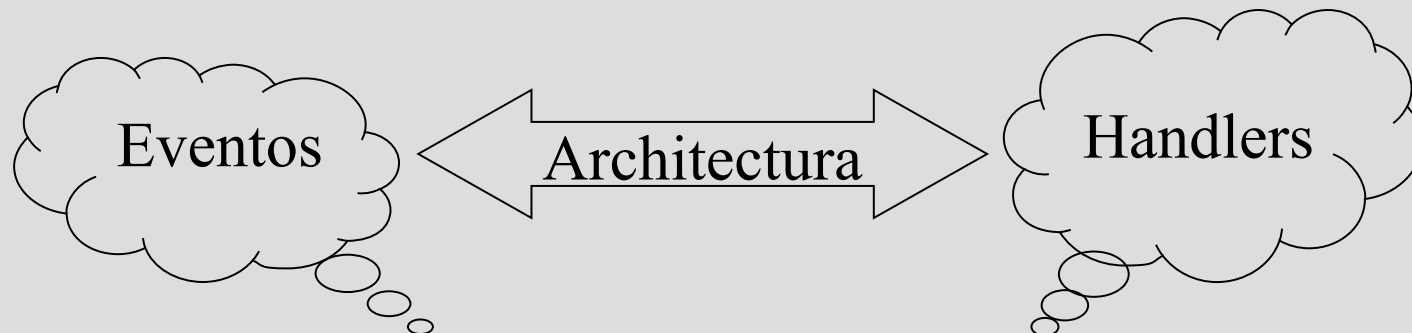
- Recordar:
  - ISR (Interrupt Service Routine): “llamadas” por HW
  - Tareas: ejecutada por software
- Restricciones de tiempo de respuesta.
- Para un problema dado:
  - ¿Cómo seleccionar una arquitectura de software?

# Elección de la arquitectura: Factores

- Factor principal: tiempo de respuesta del sistema
  - ¿Cuánto se necesita controlar el tiempo de respuesta?
  - Requerimiento de tiempo de respuesta absoluto
- ¿A cuántos eventos diferentes se ha de responder?
  - Cada uno tendrá *dead-lines* y prioridades (importancia).
- ¿Cuáles son los requerimientos de procesamiento de cada evento?
  - Un procesamiento “pesado” puede requerir “mucho” tiempo
- En resumen:
  - ¿Qué tiene que hacer el sistema?

# Arquitecturas de software

- Controladores o *handlers* de eventos
  - realizan el trabajo para responder a los eventos.
- La arquitectura del software determina:
  - cómo se detectan los eventos, y
  - cómo se llama al *handler* del evento.



# (1) Round-Robin

- No intervienen interrupciones: polling
- Evento: se consulta periódicamente si “ocurrió”
- Handler: se llama desde main

## Unico evento

```
while (TRUE)
{
    if (evento)
        controlEvento();
}
```

## Múltiples eventos

```
while (TRUE)
{
    if (eventoA)
        controlEventoA();
    if (eventoB)
        controlEventoB();
    if (eventoC)
        controlEventoC();
}
```

# Round-Robin: Características

- Prioridades:
  - Ninguna: las acciones son todas iguales
  - Cada *handler* debe esperar su turno.
- Tiempo de respuesta (peor caso):
  - Una vuelta al bucle (gestionando todos los eventos restantes primero)
- Desventajas:
  - Tiempo de respuesta malo para todos los eventos si uno solo requiere procesamiento pesado.
  - “Frágil”: agregar un nuevo *handler* modifica tiempos de respuesta restantes y puede provocar pérdida de *deadlines*.
- Ventajas:
  - Sencillez: única tarea, sin problema de datos compartidos, sin ISR.



# Round-Robin: Modificaciones

- Método para mejorar el tiempo de respuesta para un evento

```
while (TRUE)
{
    if (eventoA)
        controlEventoA();
    if (eventoB)
        controlEventoB();
    if (eventoC)
        controlEventoC();
    if (eventoD)
        controlEventoD();
}
```

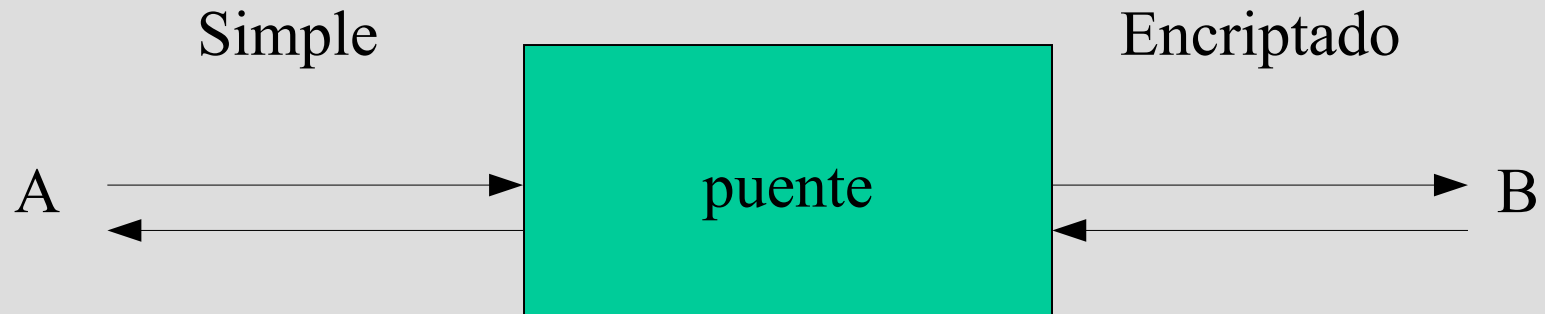
```
while (TRUE)
{
    if (eventoA)
        controlEventoA();
    if (eventoB)
        controlEventoB();
    if (eventoA)
        controlEventoA();
    if (eventoC)
        controlEventoC();
    if (eventoA)
        controlEventoA();
    if (eventoD)
        controlEventoD();
}
```

# Round-Robin: aplicabilidad

- Ejemplo del Simon: multímetro digital
  - Pocas entradas, pocos eventos a responder.
  - Tiempos de respuesta no exigentes.
  - Sin requerimientos de procesamiento pesado.
- Conclusión de Simon:
  - “Debido a esas limitaciones, la arquitectura Round-Robin es apropiada para dispositivos muy sencillos como relojes digitales y microondas (horno) y posiblemente ni siquiera para estos últimos”.

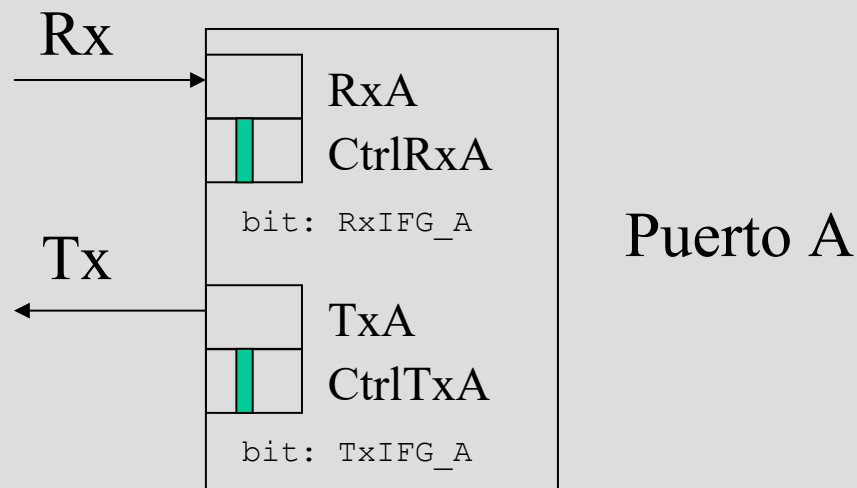
# Ejemplo: puente

- Características:
  - conecta enlaces de datos: simples (sin encriptar) y encriptados
  - dos puertos de comunicación
  - rutinas de encriptado y desencriptado



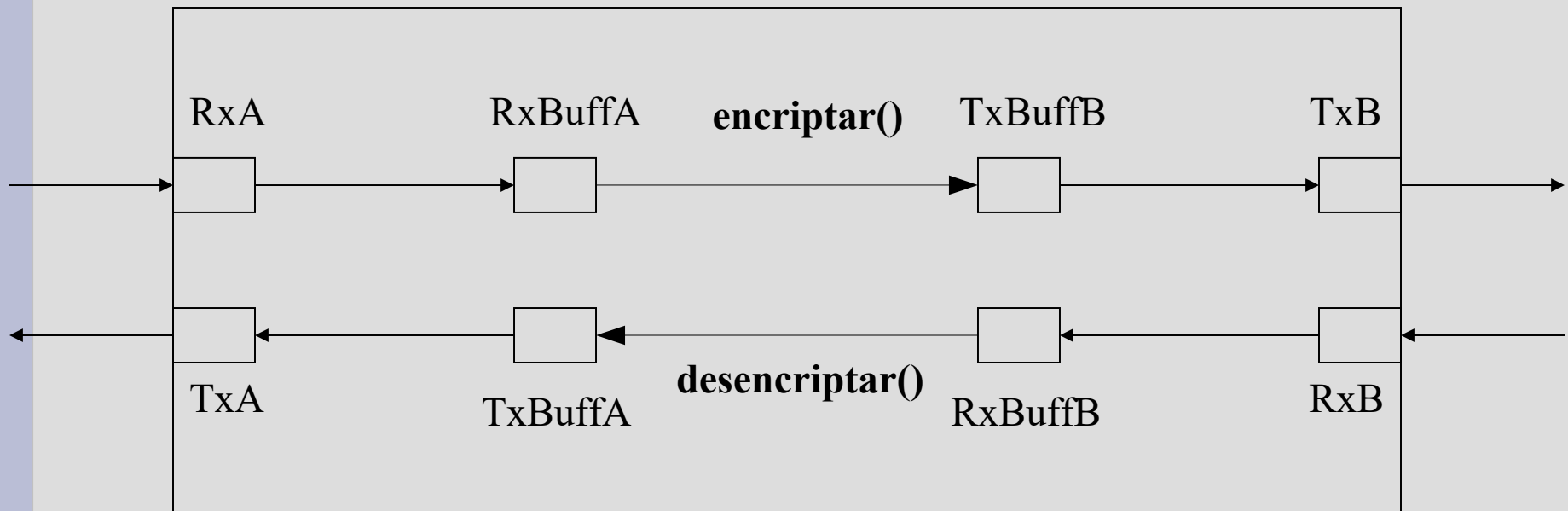
# Ejemplo: puente

- Características:
  - dos puertos de comunicación
    - registros para recepción y transmisión (no buffers internos)
    - registro de control indica: terminó recepción de un dato y finalizó transmisión de un dato
      - Pregunta: ¿Por qué son necesarios?



# Ejemplo Puente

## Implementación Round-Robin (buffer un reg)



# Solución: Round-Robin

```
char RxBufA, RxBufB;

// Rx A, encriptado, Tx B
if (RxIFG_A)
{
    RxBuffA = RxA;
    TxBuffB = encriptar(RxBuffA);
    while (!TxIFG_B) {};
    TxB = TxBuffB;
}
// Rx B, desencryptado, Tx A
if (RxIFG_B)
{
    RxBuffB = RxB;
    TxBuffA = desencryptar(RxBuffB);
    while (!TxIFG_A) {} ;
    TxA = TxBuffA;
}
```

- Problemas:
  - el tiempo en que se recibe un dato (en el puerto) y se copia puede ser grande
  - no soporta ráfagas
- Mejoras:
  - (1) esperar sin bloquear TxIFG (aunque el problema es el procesado: encriptado y desencryptado)
  - (2) se escuchan sugerencias...

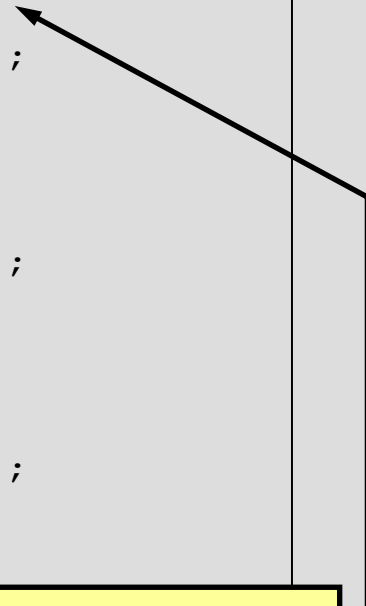
## (2) Round-Robin con interrupciones

- Agregar interrupciones:
  - ISR realiza la respuesta inicial.
  - Resto realizado por funciones llamadas en el bucle.
  - ISR indica con banderas la necesidad de procesado.
- Presenta mayor flexibilidad:
  - Respuesta de tiempo crítico disparadas por interrupciones y realizadas en ISR.
  - Código con tiempo de procesamiento “largo” ubicado en *handlers*.
- Llamado también: Foreground/Background

# Round-Robin con interrupciones

```
// subrutinas de atención a las int
void interrupt ISR_A()
{
    !! Respuesta inicial para A
    flagA = true;
}
void interrupt ISR_B()
{
    !! Respuesta inicial para B
    flagB = true;
}
void interrupt ISR_C()
{
    !! Respuesta inicial para C
    flagC = true;
}
```

```
// codigo en main luego de inicial.
while (true)
{
    if (flagA) {
        flagA = false;
        handle_eventA();
    }
    if (flagB) {
        flagB = false;
        handle_eventB();
    }
    if (flagC) {
        flagC = false;
        handle_eventC();
    }
}
```



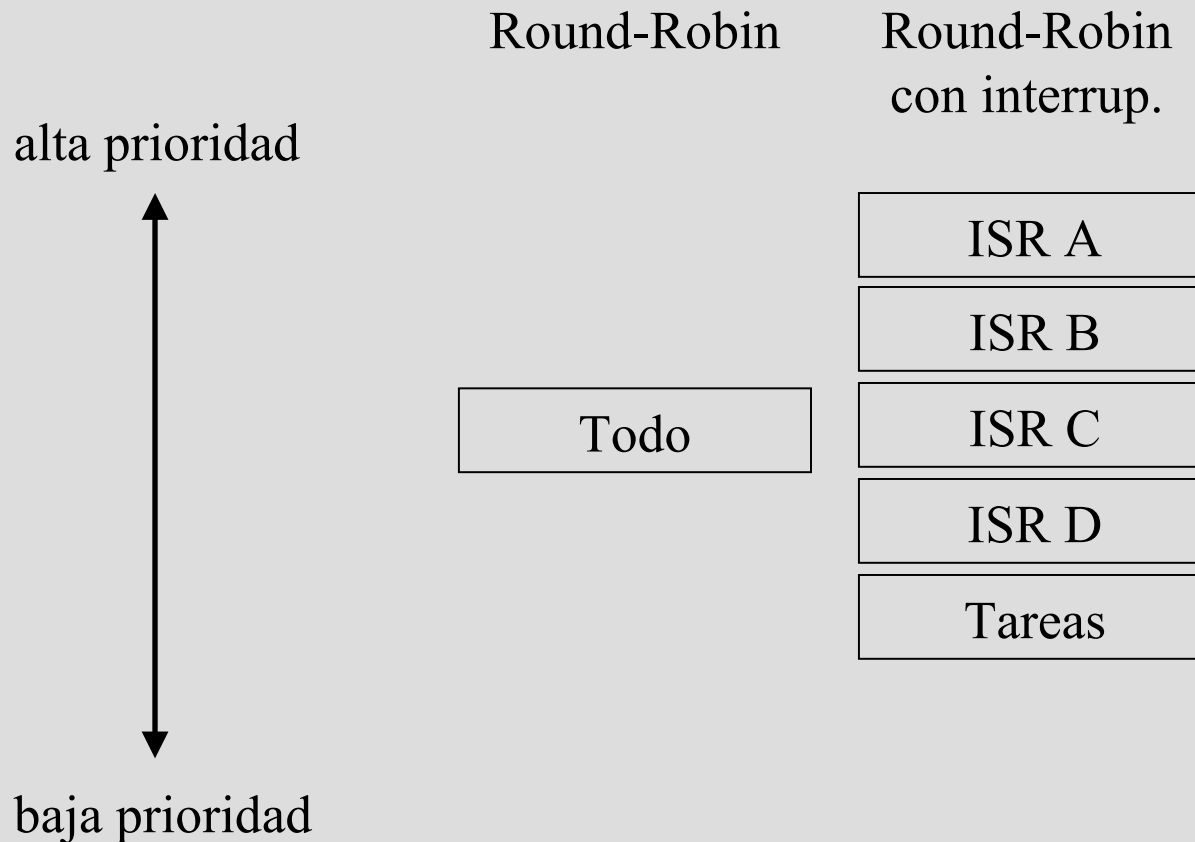
Las flags son variables compartidas: ¿es necesario protegerlas?



# Round-Robin con int: Características

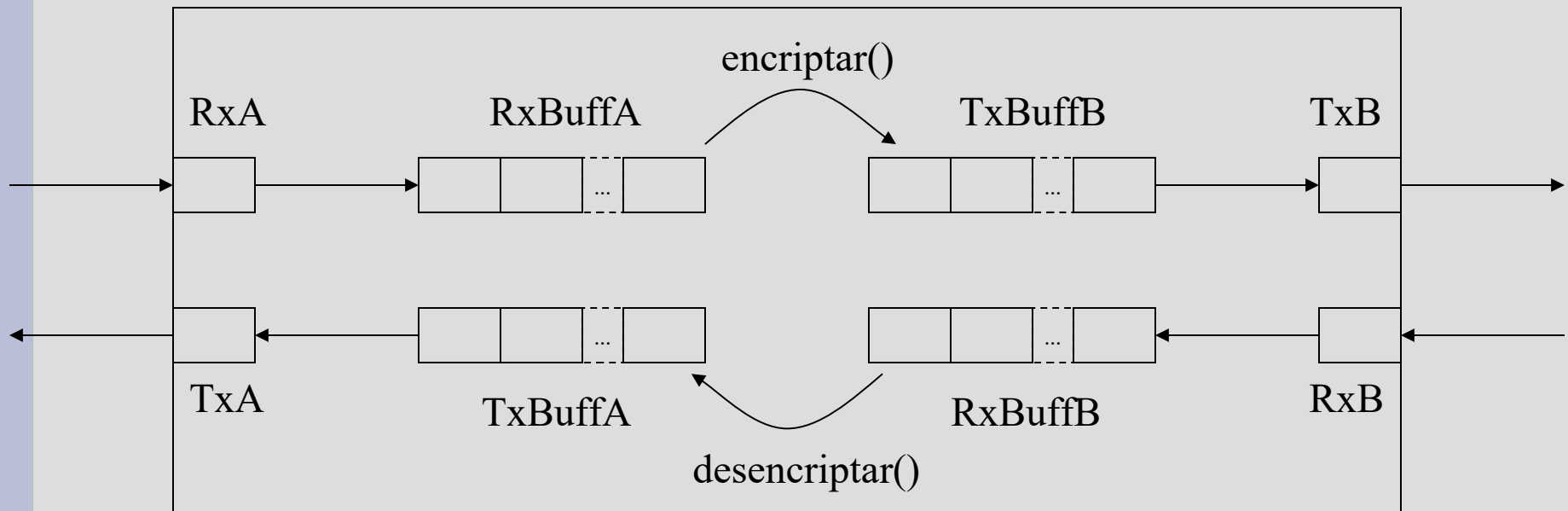
- Prioridades:
  - Interrupciones son servidas en orden de prioridad.
  - Todos los *handlers* tienen igual prioridad.
- Tiempo de respuesta:
  - ISR: tiempo de ejecución de las ISR de mayor prioridad.
  - *Handler*: suma de tiempos del resto de los handlers + interrupciones
- Ventajas:
  - Trabajo realizado en ISR tiene mayor prioridad.
  - Tiempo de respuesta de ISR estable si el código de la tarea cambia.
- Desventajas:
  - ISRs y *handlers* comparten datos.
  - Tiempo de respuesta de *handler* no estable cuando código cambia.

# Comparación: Niveles de prioridad



# Ejemplo Puente

## Solución Round-Robin con interrupciones y colas.



# Actividad en grupo

- Bosquejar solución (seudo-código) del puente con Round-Robin con interrupciones y colas de datos.
  - Objetivo:
    - Usar cola definida/desarrollada en clase anterior
    - ¿cuántas colas se precisan? ¿cuántas ISR?
    - Funciones disponibles: `char encrypt (char)` y `char desencrypt (char)`
  - Grupos:
    - 2 a 4 participantes
  - Tiempo:
    - 10 minutos
  - Puesta en común.

# Solución: Round-Robin c/int

```
while (TRUE) {  
    if (!qRxA_isempty()) {  
        qTxB_add(encrypt(qRxA_get()));  
    }  
    if (!qRxB_isempty()) {  
        qTxA_add(desencrypt(qRxB_get()));  
    }  
    if (!qTxA_isempty()) {  
        while (!TxIFG_A) {};  
        TxA = qTxA_get();  
    }  
    if (!qTxB_isempty()) {  
        while (!TxIFG_B) {};  
        TxB = qTxB_get();  
    }  
}
```

```
#pragma vector=UART_RxA  
__interrupt void RxA_ISR(void) {  
    qRxA_add(RxA);  
}  
  
#pragma vector=UART_RxB  
__interrupt void RxB_ISR(void) {  
    qRxB_add(RxB);  
}  
  
#pragma vector=UART_TxA  
__interrupt void TxA_ISR(void) {  
    TxA = qTxA_get();  
}  
  
#pragma vector=UART_TxB  
__interrupt void TxB_ISR(void) {  
    TxB = qTxB_get();  
}
```

Cuidado de no sacar datos de colas vacías o poner datos en colas llenas

# Solución: Round-Robin c/int

```
TxIE_A=0;
while (TRUE) {
    if (!qRxA_isempty()) {
        qTxB_add(encrypt(qRxA_get()));
    }
    if (!qRxB_isempty()) {
        qTxA_add(desencrypt(qRxB_get()));
    }
    if (!qTxA_isempty()) {
        while (!TxIFG_A) {};
        TxA = qTxA_get();
        TxIE_A=1;
    }
    if (!qTxB_isempty()) {
        while (!TxIFG_B) {};
        TxB = qTxB_get();
    }
}
```

```
#pragma vector=UART_RxA
__interrupt void RxA_ISR(void) {
    qRxA_add(RxA);
}

#pragma vector=UART_RxB
__interrupt void RxB_ISR(void) {
    qRxB_add(RxB);
}

#pragma vector=UART_TxA
__interrupt void TxA_ISR(void) {
    TxA = qTxA_get();
    if (!qTxA_isempty()) {
        TxIE_A=0;
    }
}

#pragma vector=UART_TxB
__interrupt void TxB_ISR(void) {
    TxB = qTxB_get();
}
```

# (3) Planificación por encolado de funciones

- Similar a Round-Robin con interrupciones:
  - Trabajo dividido en ISR y *handlers*
  - ISR encola su *handler*
- Características de la cola de funciones:
  - FIFO: First Input First Output (cola clásica)
  - Permite agregar prioridades a la tarea: “cola con prioridades” (implementación posible: una cola por prioridad)

# Planificación por encolado de funciones

```
// rutinas de atención a las int
```

```
void interrupt ISR_A()
{
    !! Respuesta inicial para A
    queue_add(handle_eventA)
}
void interrupt ISR_B()
{
    !! Respuesta inicial para B
    queue_add(handle_eventB)
}
void interrupt ISR_C()
{
    !! Respuesta inicial para C
    queue_add(handle_eventC)
}
```

```
// código en main luego de iniciar
```

```
void (*task)();

while (true)
{
    while (queue_is_empty()){};
    task = queue_get();
    task();
}
```



# Planificación por cola de funciones (con prioridades): características

- Prioridades:
  - Interrupciones son servidas en orden de prioridad (igual que RR c/int).
  - Tareas son encoladas y ejecutadas en orden:
    - Según su prioridad.
    - Según la ejecución de la ISR dentro de la misma prioridad.
- Tiempo de respuesta para tarea prioritaria (peor caso)
  - Hipótesis: una única tarea prioritaria.
  - El peor caso se da si justo es encolada cuando se comienza a ejecutar otra tarea (incluyendo tareas de baja prioridad, no expropia).
  - Retardo = tarea más larga (incluyendo baja prioridad) + tiempo de ejecución ISRs
  - ¿Cómo sería si hay más de una tarea prioritaria? El peor caso sería lo anterior + la ejecución de todas las tareas previamente encoladas de igual prioridad.
  - ¿Cómo puede arreglarse? Una cola para cada prioridad.
- Ventajas:
  - Mejora de tiempo de respuesta y estabilidad cuando el código cambia.
- Desventaja:
  - Incremento en la complejidad: se debe implementar la cola.

## (4) RTOS

- Trabajo dividido en ISR y tareas.
- Tareas priorizadas y ejecutadas por un *scheduler*.
  - Tarea con la mayor prioridad se ejecuta primero.
  - Si una tarea de mayor prioridad pasa a *ready*, la tarea de prioridad baja es despojada (se le expropia el  $\mu$ P, *preempt*).
- Tareas se bloquean si esperan eventos o recursos
  - ISR puede desbloquear tareas.
  - Tareas pueden retardarse por un tiempo dado.
- RTOS provee código para:
  - Crear tareas, bloquear y desbloquear tareas, agendar tareas, comunicar ISR y tareas, etc.

# RTOS: Ejemplo (simplificado)

```
void interrupt vHandleDeviceA()
{
    !! Take care of I/O Device A
    !! Signal X
}

void interrupt vHandleDeviceB()
{
    !! Take care of I/O Device B
    !! Signal Y
}
```

```
void main ()
{
    InitRTOS();
    StartTask(vTask1,HIGH_PRIORITY);
    StartTask(vTask2,LOW_PRIORITY);
    StartRTOS();
}
```

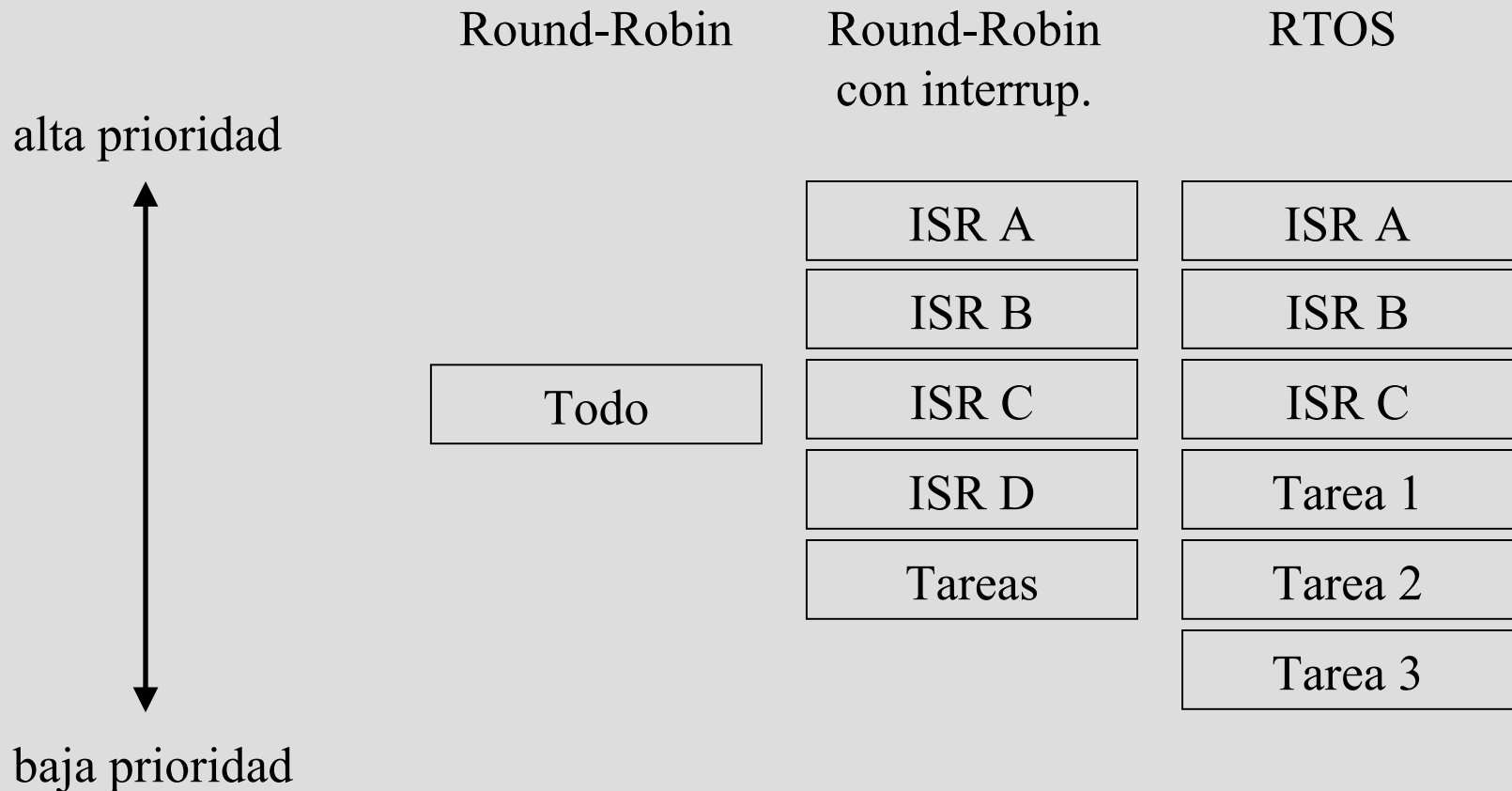
```
void vTask1()
{
    while (true) {
        !! Wait for Signal X
        !! Handle data to/from A
    }
}

void vTask2()
{
    while (true) {
        !! Wait for Signal Y
        !! Handle data to/from B
    }
}
```

# RTOS: Características

- Prioridades:
  - Interrupciones son servidas en orden de prioridad.
  - Tareas son agendadas por orden de prioridad.
  - Tareas de menor prioridad son expropiadas.
- Tiempo de respuesta:
  - suma de los tiempos de ISR + tareas de mayor prioridad.
- Ventajas:
  - Estabilidad cuando el código cambia: agregar una tarea no afecta el tiempo de respuesta de las tareas de mayor prioridad.
  - Muchas opciones disponibles: comerciales y GNU.
- Desventajas:
  - Complejidad: mucho dentro de RTOS, algo para usarlo.
  - Requerimientos mínimos y overhead del RTOS.

# Niveles de prioridad



# Selección de la arquitectura

- Más simple que cumpla con los requerimientos de tiempos de respuesta actuales y futuros...
- Optar por usar RTOS si se tienen requerimientos difíciles:
  - beneficios extra: soporte para debugging, profiler, etc.
- Se pueden construir arquitecturas híbridas:
  - RTOS + polling: una tarea del RTOS realiza polling.
  - RR c/int + polling: bucle principal de RR c/int consulta hardware lento directamente mediante polling.

# Comparación entre arquitecturas

	Prioridades	Tiempo de resp. de tarea prioritaria (*)	Escalabilidad	Simplicidad
Round-Robin	Ninguna (tarea misma prioridad)	Suma de todas las tareas	Mala	Muy simple
Round-Robin con interrup.	ISR en orden, tarea misma prioridad	Idem RR (pero tarea crítica en ISR)	Buena para ISR, mala para tarea	Problema de datos compartidos
Encolado de funciones con prioridades	ISR y tareas en orden (no expropia)	Tarea + larga	Buena	Idem anterior + cola
RTOS	ISR y tareas en orden (expropia)	Cero	Muy buena	Complejo (pero dentro de RTOS)

# Bibliografía

- “An Embedded Software Primer” David E. Simon
  - Chapter 5: Survey of Software Architectures.