7919 | Sistemas Embebidos

2º Cuatrimestre de 2017

CLASE 6: ARQUITECTURAS DE SOFTWARE EMBEBIDO

Prof: José Moyano
Autor original: Sebastián Escarza

Dpto. de Cs. e Ing. de la Computación
Universidad Nacional del Sur
Bahía Blanca, Buenos Aires, Argentina

GrIDSE

Tareas e ISRs

Tareas

- El software de un sistema embebido realiza diversas funciones tales como:
  - atender y controlar dispositivos
  - realizar cómputos
  - medir intervalos de tiempo
  - configurar e inicializar el hardware
  - etc.

- A cada una de dichas funciones podemos asociar una unidad de ejecución que la implementa (Tarea).

Tareas

- En Sistemas Operativos de Tiempo Real se habla de tareas como unidades de programa secuenciales y planificables.
- Tarea (Sistemas Embebidos):
  - es una unidad de ejecución secuencial
  - su ejecución se planifica o coordina de alguna manera
  - posee cohesión funcional (se origina como resultado de descomponer funcionalmente al sistema)
  - la noción de tarea trasciende los constructores del lenguaje de programación (una tarea puede ser una función, módulo, método, objeto, etc. o simplemente una secuencia de código en una unidad mayor)

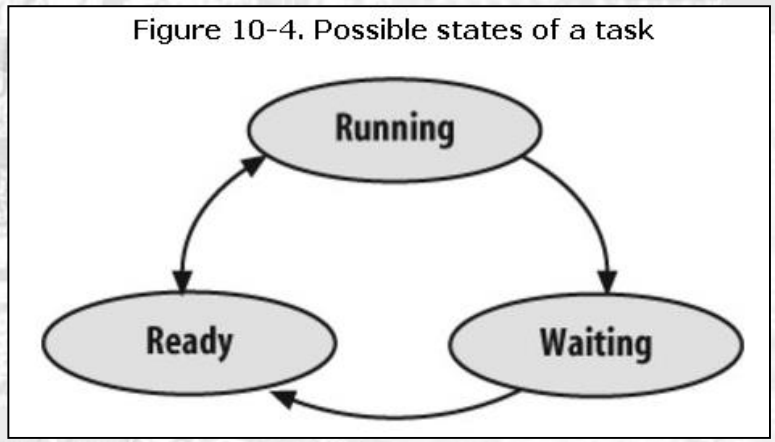
Tareas

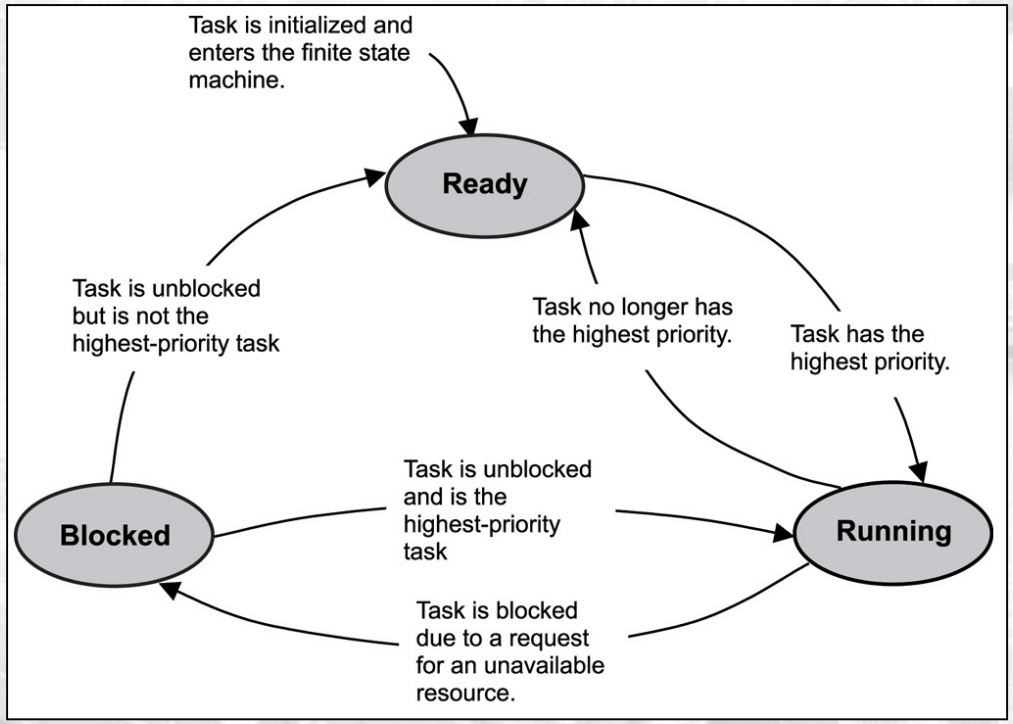
- Planificación y coordinación de tareas:
  - En ausencia de un SO, las tareas son atendidas y coordinadas por un módulo principal que se encarga de planificarlas de acuerdo a algún esquema.
  - En presencia de un SO, es el scheduler del sistema operativo quien abstrae al implementador de estos detalles. El implementador simplemente especifica sus tareas como unidades funcionales separadas, y la lógica de coordinación queda "oculta" por el sistema.

- En ambos casos, si existe comunicación entre tareas, hay que sincronizar su ejecución.

Tareas

- Tanto en presencia de Sistemas Operativos como sin ellos, podemos considerar que las tareas se encuentran en determinado estado:





Tareas

- Las tareas suelen tener:
  - diversas prioridades.
  - deadlines (tiempos máximos en los cuáles la tarea debe cumplirse – tiempo real).
- La asignación de prioridades determinará qué tareas tienen privilegios para acceder al CPU.
- Los deadlines deben ser tenidos en cuenta para asegurar que ningún dispositivo deja de ser atendido en tiempo y forma.
- En sistemas con restricciones de tiempo real: correctitud funcional + **correctitud temporal**.



ISRs

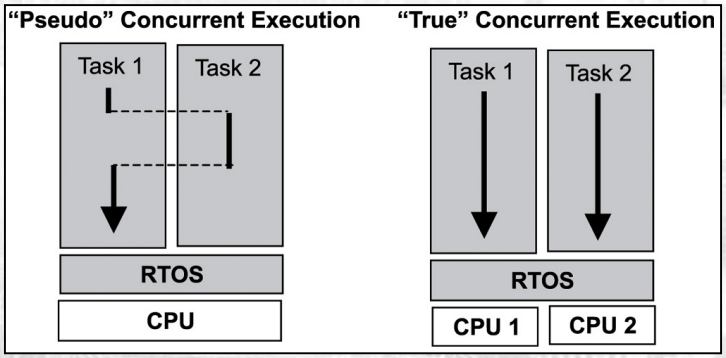
- En sistemas que realizan E/S usando interrupciones, la ejecución de las tareas puede ser interrumpida por la ocurrencia de eventos notificados por el hardware.
- El control es transferido a rutinas que dan atención a dichos eventos y que poseen mayor prioridad que las tareas: las Interrupt Service Routines (ISRs).
- Las ISRs comparten con las tareas el hecho de:
  - ser unidades de ejecución secuencial
  - poseer cohesión funcional (se originan como resultado de descomponer funcionalmente al sistema)

ISRs

- Las ISRs se diferencian de las tareas en que:
  - inician su ejecución ante la ocurrencia de eventos asincrónicos
  - no se rigen por los esquemas de planificación y coordinación habituales
  - su implementación se realiza mediante un constructor específico del lenguaje de programación (normalmente una función con un calificador especial que denota su condición de ISR)
- Generalmente la implementación funcional de un sistema está compuesta por un conjunto de Tareas e ISRs.

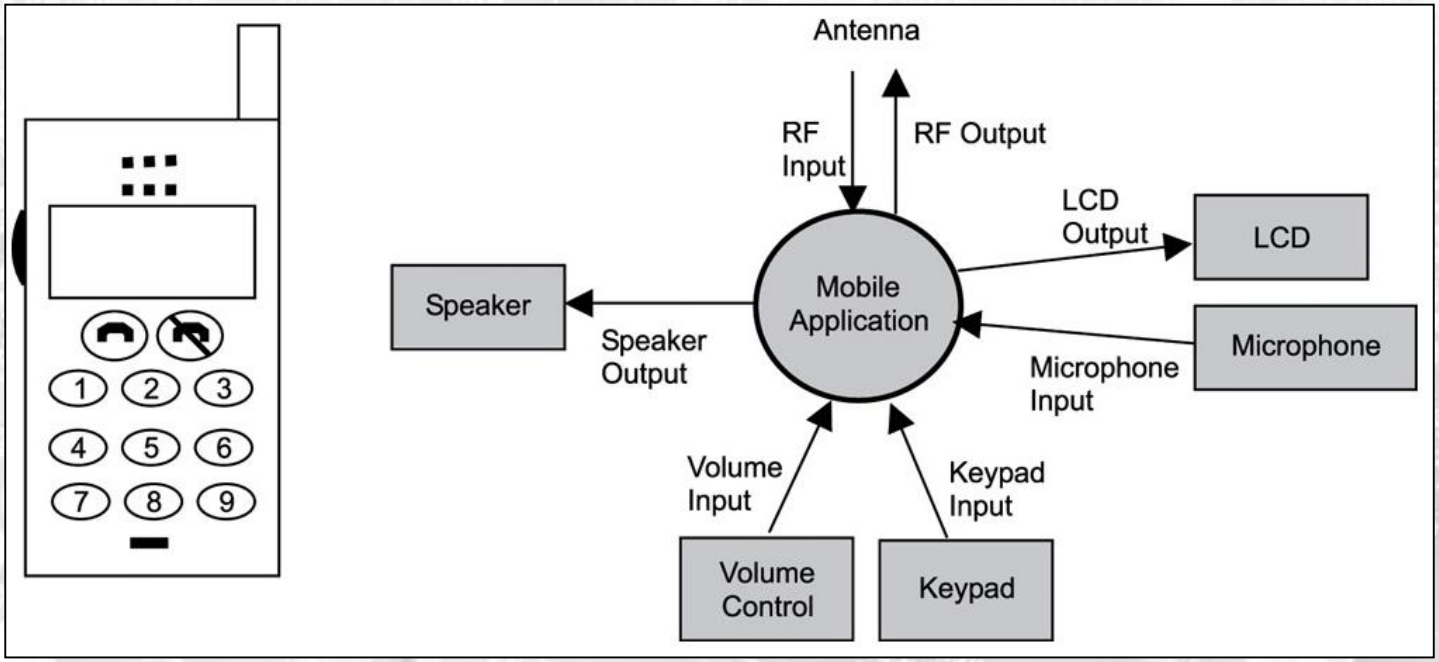
Tareas e ISRs

- De la descomposición funcional de un sistema:
  - Se identifican y especifican Tareas e ISRs
  - Se determinan dependencias entre las mismas (datos y recursos compartidos, sincronización y comunicación, oportunidades de concurrencia, etc).
  - Se fijan las restricciones temporales (tiempo real).
  - Se aplican criterios de modularidad (optim. tareas e ISRs)
  - Se analiza el esquema y la posibilidad de planificación (por ej. Rate Monotonic Analysis) ante restricciones de tiempo real (el análisis de planificabilidad lo veremos más adelante).



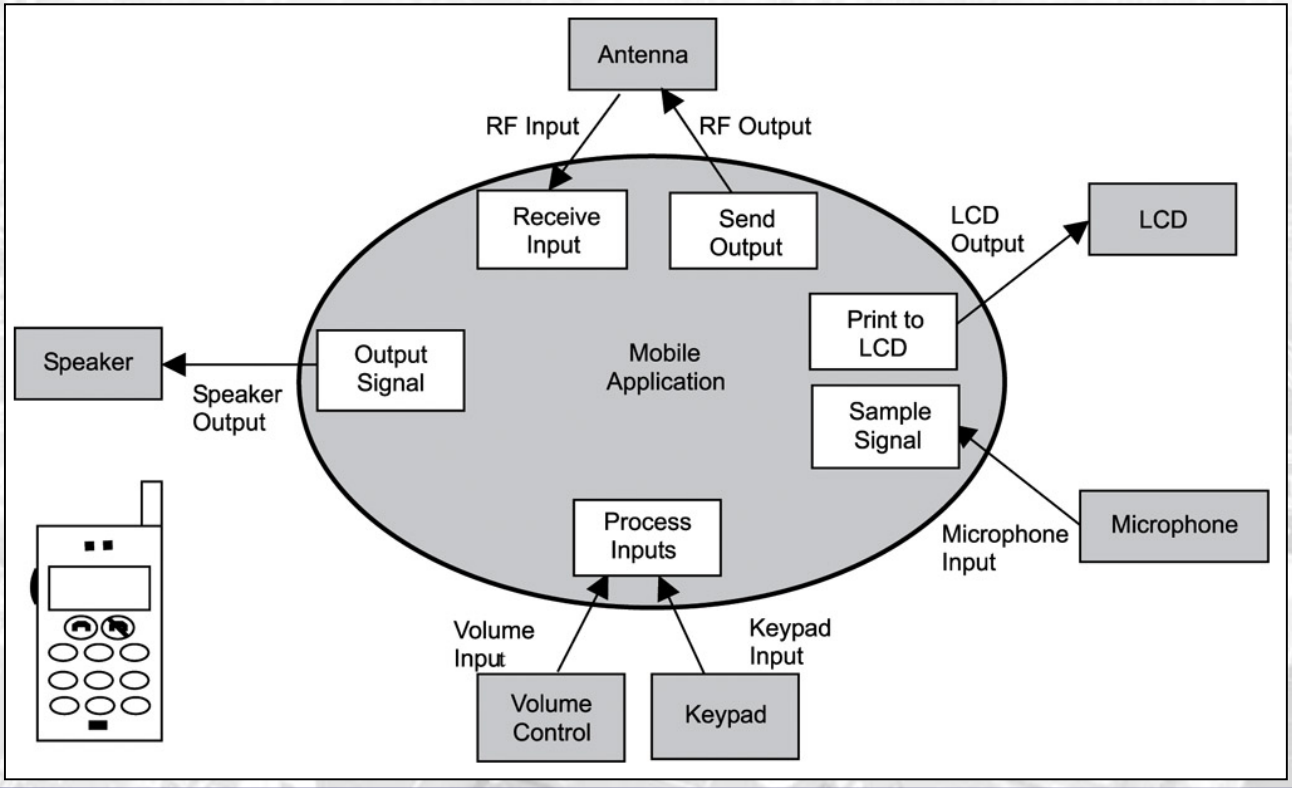
Tareas e ISRs

- Ej: Un teléfono celular simple



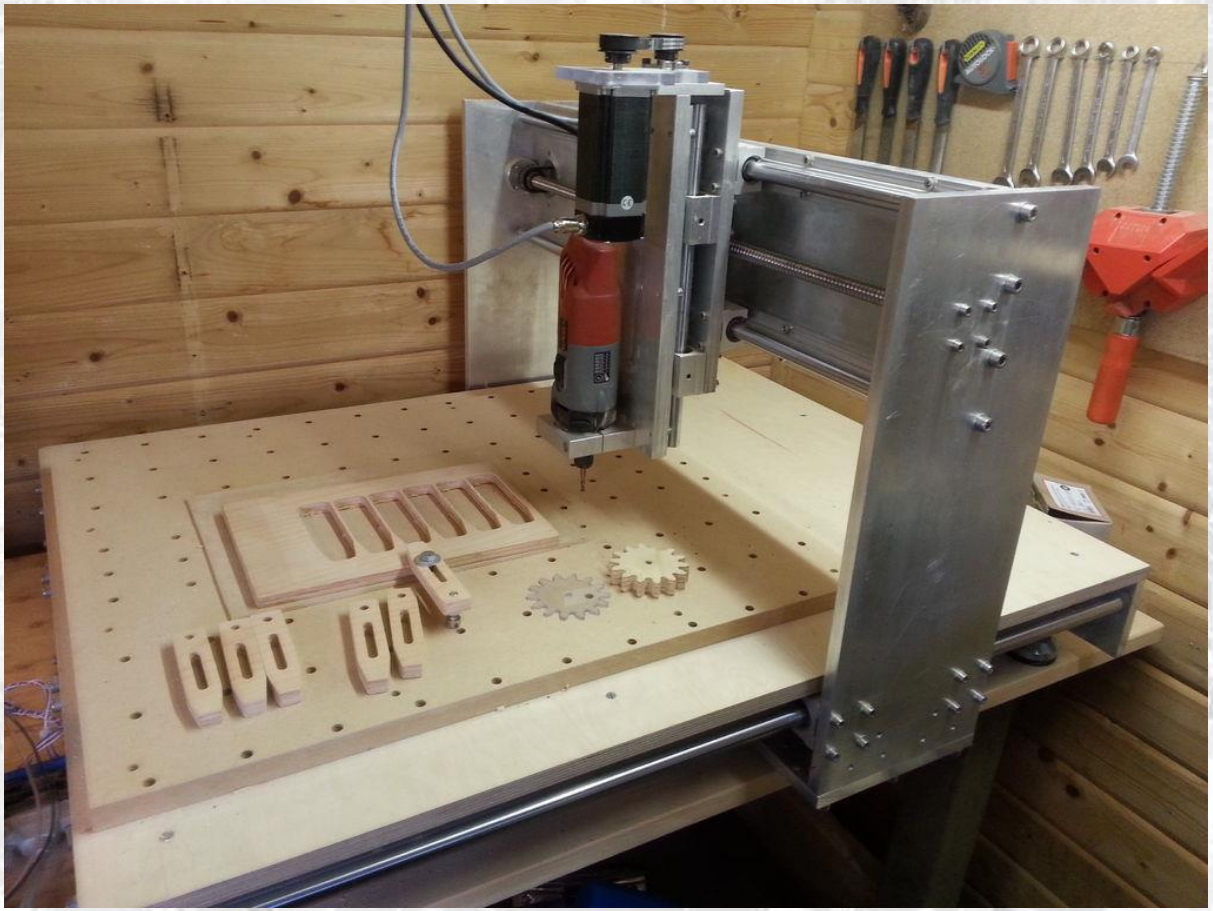
Tareas e ISRs

- Ej: División de un sistema en tareas e ISRs (outside-in approach).

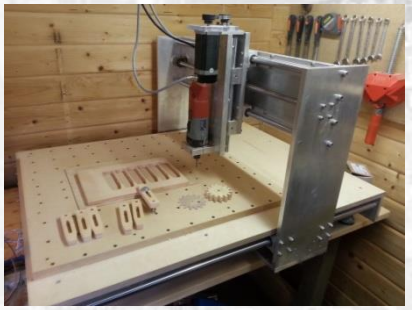


Antes de seguir: un ejemplo...

Caso de Estudio: Router CNC

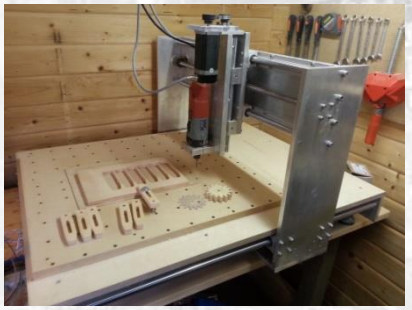


Caso de Estudio: Router CNC



- Router CNC de 3 ejes:
  - Calado de piezas a distinta profundidad en base a comandos por puerto serie.
  - Monitoreo y control posicional
    - Ejes X,Y: Motores DC con encoder lineal digital (ext. Int)
    - Eje Z: Stepper con realimentación analógica (sampling)
  - ADC Pasivo (múltiples canales)
  - Timer(s)
  - Comandos por puerto serial (movimientos en X, Y y Z, start, stop). UART x Interrupciones.
  - Comandos mediante botones pasivos/polling: (start, stop, emergency).
  - Sensor analógico de temperatura de la broca.

Caso de Estudio: Router CNC



- Identificar Tareas e ISRs (partición funcional)
  - Especificación: Diagrama de Flujos
- Identificar dependencias y oportunidades de comunicación y sincronización
- Definir temporizados, prioridades y deadlines
- Modos de operac. del sistema (idle, working, calibration, error, etc.)
  - Diagramas de Transición de Estados
- Protocolo de comunicación
  - Diagramas de secuencia

Arquitecturas de Software

Arquitecturas de Software

- El esquema de planificación de tareas define la estructura del software embebido.
- La elección de qué arquitectura de software utilizar condiciona:
  - el tipo de respuesta del sistema ante eventos
  - la forma en que el sistema gestiona múltiples eventos (ej. prioridades)
  - la complejidad del sistema (testeo, mantenimiento, interoperabilidad, etc)
- A continuación analizaremos diversas arquitecturas de software usuales en sistemas embebidos, en orden creciente de complejidad.

Arquitectura Round-Robin

- Un único bucle donde todos los eventos y acciones se tratan sincrónicamente.
  - Inicialmente se controla la ocurrencia de eventos en los dispositivos (polling).
  - Luego se atiende cada uno de los dispositivos que así lo requieran siguiendo un orden preestablecido.
- No hay interrupciones.
- No hay problemas de sincronización.
- La arquitectura más simple.
- Utilizado en sistemas muy simples (pocas tareas): multímetros digitales, hornos a microondas, etc.

Arquitectura Round-Robin

- Ej: el ciclo infinito principal revisa uno a uno los dispositivos y en función de dichas condiciones, los atiende siguiendo un orden preestablecido.

Figure 5.1 Round-Robin Architecture

```
void main (void)
{
   while (TRUE)
   {
      if (!! I/O Device A needs service)
      {
         !! Take care of I/O Device A
         !! Handle data to or from I/O Device A
      }
      if (!! I/O Device B needs service)
      {
         !! Take care of I/O Device B
         !! Handle data to or from I/O Device B
      }
      etc.
      etc.
      if (!! I/O Device Z needs service)
      {
         !! Take care of I/O Device Z
         !! Handle data to or from I/O Device Z
      }
   }
}
```

Arquitectura Round-Robin

- **Ej: Multímetro digital.**

Figure 5.2 Digital Multimeter

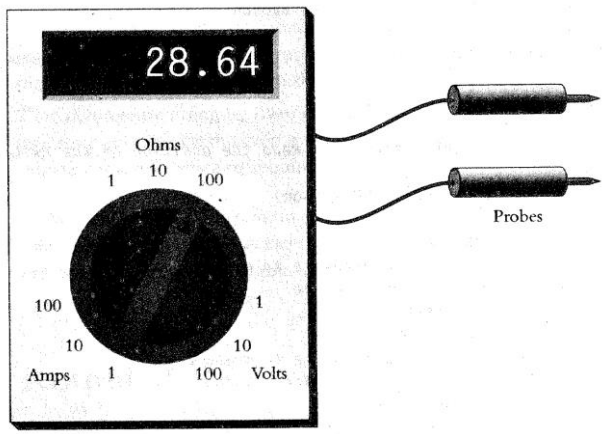


Figure 5.3 Code for Digital Multimeter

```
void vDigitalMultiMeterMain (void)
{
   enum {OHMS_1, OHMS_10, ..., VOLTS_100} eSwitchPosition;

   while (TRUE)
   {
      eSwitchPosition = !! Read the position of the switch;

      switch (eSwitchPosition)
      {
         case OHMS_1:
            !! Read hardware to measure ohms
            !! Format result
            break;
         case OHMS_10:
            !! Read hardware to measure ohms
            !! Format result
            break;
         .
         .
         .
         case VOLTS_100:
            !! Read hardware to measure volts
            !! Format result
            break;
      }
      !! Write result to display
   }
}
```

Arquitectura Round-Robin

- Útil cuando:
  - hay pocos dispositivos
  - no hay restricciones de tiempo importantes
  - no hay tiempos de procesamiento demasiado elevados en respuesta a eventos.
- No es adecuada cuando:
  - el bucle se hace demasiado largo (o un dispositivo necesita ser atendido rápidamente)
  - se pueden perder eventos o datos
  - añadir un nuevo dispositivo puede comprometer los resultados alcanzados con este esquema

Arq. Round-Robin c/Interrupciones

- Un ciclo principal que gestiona el procesamiento siguiendo un orden preestablecido.
- Mediante interrupciones y flags se determina qué dispositivo atender. El ciclo principal hace polling sobre los flags.
- Incorpora un mejor manejo de prioridades ya que las ISRs pueden priorizarse entre sí además de por sobre el ciclo principal.
- Aparecen potenciales problemas al compartir datos entre las ISRs y el ciclo principal (flags).

Arq. Round-Robin c/Interrupciones

- **Ej:**

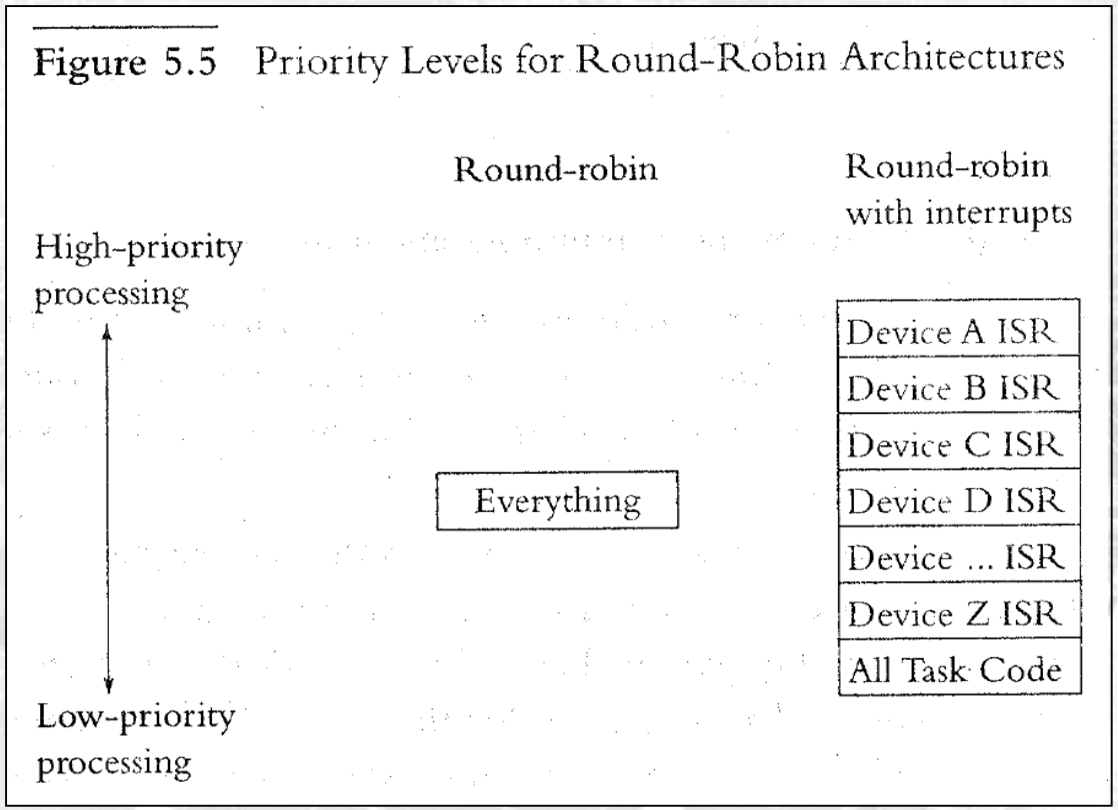
Figure 5.4 Round-Robin with Interrupts Architecture

```
BOOL fDeviceA = FALSE;
BOOL fDeviceB = FALSE;
:
BOOL fDeviceZ = FALSE;

void interrupt vHandleDeviceA (void)
{
   !! Take care of I/O Device A
   fDeviceA = TRUE;
}

void interrupt vHandleDeviceB (void)
{
   !! Take care of I/O Device B
   fDeviceB = TRUE;
}
:
void interrupt vHandleDeviceZ (void)
{
   !! Take care of I/O Device Z
   fDeviceZ = TRUE;
}
```

```
void main (void)
{
   while (TRUE)
   {
      if (fDeviceA)
      {
         fDeviceA = FALSE;
         !! Handle data to or from I/O Device A
      }
      if (fDeviceB)
      {
         fDeviceB = FALSE;
         !! Handle data to or from I/O Device B
      }
      :
      if (fDeviceZ)
      {
         fDeviceZ = FALSE;
         !! Handle data to or from I/O Device Z
      }
   }
}
```

Arq. Round-Robin c/Interrupciones

- Comparación de prioridades:



Arq. Round-Robin c/Interrupciones

- Limitaciones:
  - El código de cada tarea corre a la misma prioridad (introduce latencias importantes en presencia de otras tareas demandantes en el tiempo).
    - Opción 1: Mover el código de la tarea a su ISR (complica los tiempos de atención de los eventos de las demás tareas).
    - Opción 2: Incrementar la frecuencia de polling de las tareas demandantes (por ej. chequeando flags varias veces por ciclo). Es un esquema limitado: mientras más veces se chequean los flags de una tarea, más demora potencial introducimos en la atención de las otras.
  - Tiempos largos de respuesta si justo ocurre el evento inmediatamente después de que se pasó el polling del flag.

Arq. Function-Queue-Scheduling

- Cada ISR de cada tarea, añade un puntero a la función que lleva a cabo dicha tarea a una cola con prioridad.
- El ciclo principal, simplemente invoca a las funciones de la cola.
- De esta manera, la función de las tareas con mayor prioridad se ejecutan primero.
  - Disminuye el tiempo de atención de las rutinas con mayor prioridad.
  - Puede aumentar el tiempo de atención de las rutinas con menor prioridad (en presencia de eventos más prioritarios). Riesgo de inanición.

Arq. Function-Queue-Scheduling

- Ej: En el ciclo principal se accede a la cola según las prioridades y se pone a correr (sólo cuando el CPU está ocioso) la tarea de mayor prioridad. Sólo las ISRs pueden interrumpir las tareas.

Figure 5.8 Function-Queue-Scheduling Architecture

```
!! Queue of function pointers;

void interrupt vHandleDeviceA (void)
{
   !! Take care of I/O Device A
   !! Put function_A on queue of function pointers
}

void interrupt vHandleDeviceB (void)
{
   !! Take care of I/O Device B
   !! Put function_B on queue of function pointers
}

void main (void)
{
   while (TRUE)
   {
      while (!!Queue of function pointers is empty)
         ;

      !! Call first function on queue
   }
}

void function_A (void)
{
   !! Handle actions required by device A
}

void function_B (void)
{
   !! Handle actions required by device B
}
```

Arq. Function-Queue-Scheduling

- Funciones largas de baja prioridad pueden comprometer los tiempos de respuesta de las funciones de alta prioridad.
- Si bien se puede reestructurar en bloques menores las rutinas largas (algo complejo de hacer), en el fondo la planificación de las rutinas no es apropiativa.

- Para escenarios de mayor complejidad se utilizan sistemas operativos embebidos (RTOS).

Arquitectura RTOS

- Para escenarios complejos: Real-Time Operating Systems.
- La aplicación se estructura en un conjunto de ISRs y tareas que se sincronizan.
- Las ISRs señalizan a las tareas.
- Las tareas son planificadas por el RTOS (prioridades, apropiación, etc.).
- Los mecanismos de sincronización (al igual que otros servicios) son provistos por el Sistema Operativo.

Arquitectura RTOS

- Ej: El bucle principal ahora forma parte del scheduler del sistema operativo.
- La sincronización y planificación son servicios provistos por el RTOS.

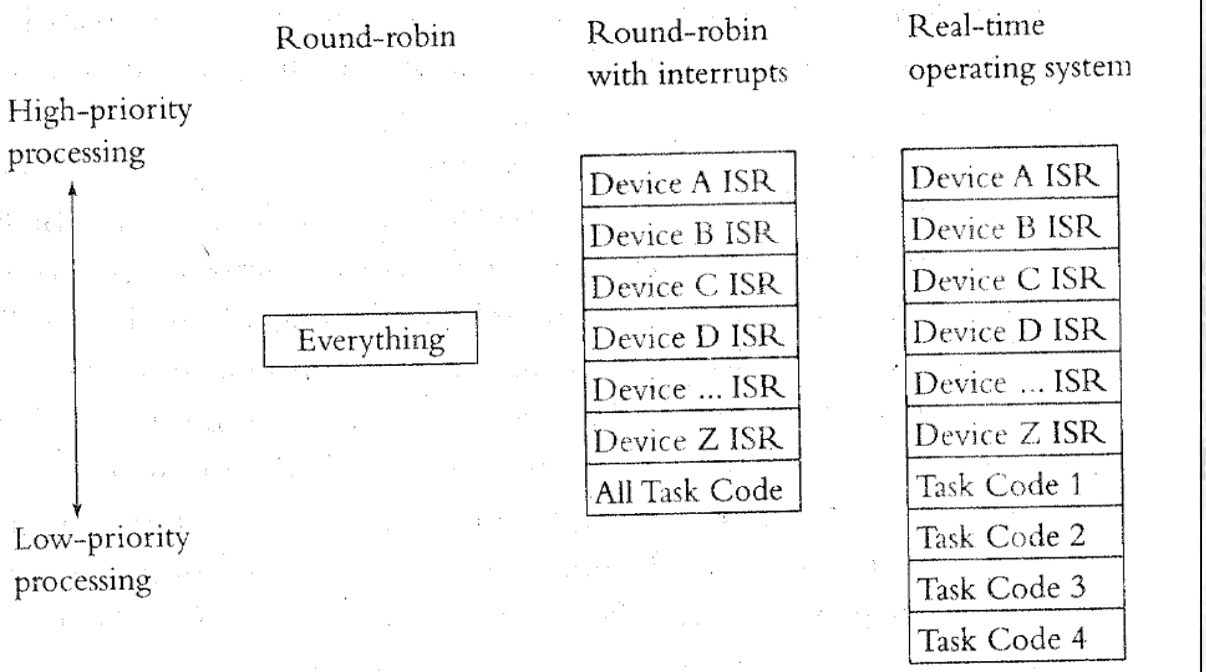
Figure 5.9 Real-Time-Operating-System Architecture

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

void interrupt vHandleDeviceB (void)
{
   !! Take care of I/O Device B
   !! Set signal Y
}
.
.
.
void Task1 (void)
{
   while (TRUE)
   {
      !! Wait for Signal X
      !! Handle data to or from I/O Device A
   }
}

void Task2 (void)
{
   while (TRUE)
   {
      !! Wait for Signal Y
      !! Handle data to or from I/O Device B
   }
}
.
.
.
```

Arquitectura RTOS

- Comparación de prioridades:

Figure 5.10 Priority Levels for Real-Time-Operating-System Architectures



Arquitectura RTOS

- Ventajas:
  - El tiempo entre un evento y su atención dependen del quantum asignado a cada tarea (además de la latencia de interrupción, tiempo del cambio de contexto, etc.).
  - Si se cambia una tarea, el esquema de temporizado continúa siendo estable (cambios en las tareas menos prioritarias no afectan a los tiempos de respuesta de las tareas más prioritarias).
  - Los RTOS proveen muchos servicios adicionales.
- Desventaja:
  - El RTOS consume recursos (el scheduler insume tiempo), se requiere espacio en memoria para alojarlo, etc. Sólo para sistemas con ciertos recursos.

Seleccionando una arquitectura

- Elegir la más simple posible dentro de los requerimientos funcionales y temporales de la aplicación.
- Ciertas soluciones utilizan arquitecturas híbridas.

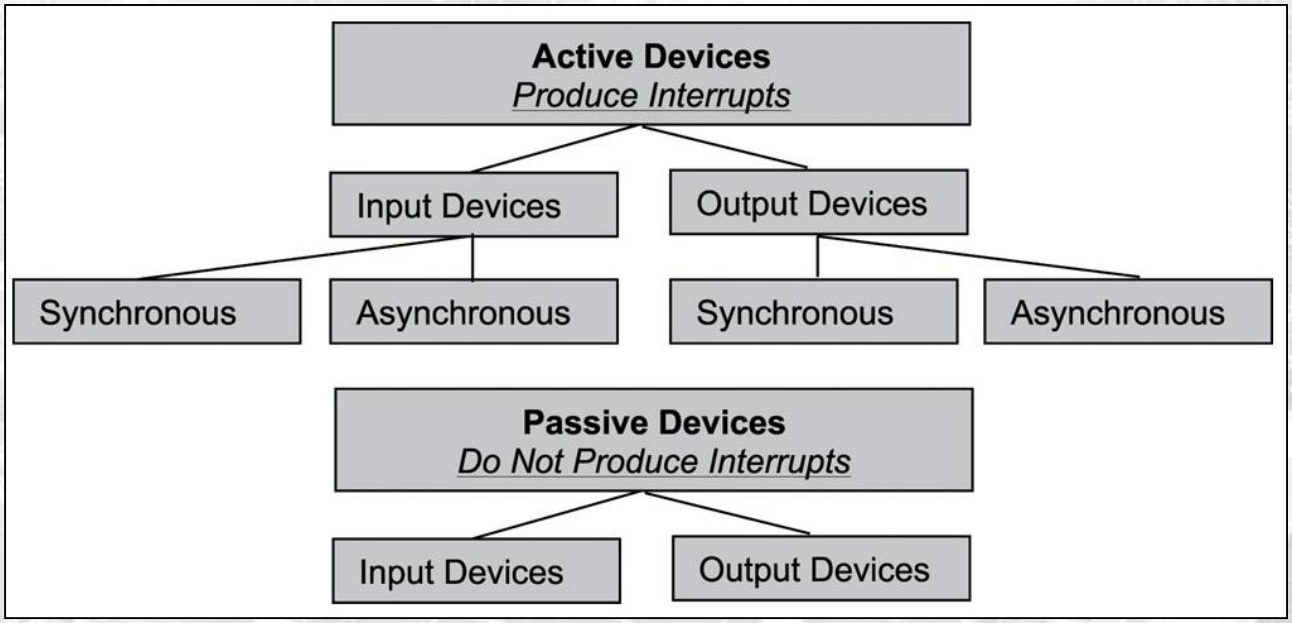
Table 5.1 Characteristics of Various Software Architectures

	Priorities Available	Worst Response Time for Task Code	Stability of Response When the Code Changes	Simplicity
Round-robin	None	Sum of all task code	Poor	Very simple
Round-robin with interrupts	Interrupt routines in priority order, then all task code at the same priority	Total of execution time for all task code (plus execution time for interrupt routines)	Good for interrupt routines; poor for task code	Must deal with data shared between interrupt routines and task code
Function-queue-scheduling	Interrupt routines in priority order, then task code in priority order	Execution time for the longest function (plus execution time for interrupt routines)	Relatively good	Must deal with shared data and must write function queue code
Real-time operating system	Interrupt routines in priority order, then task code in priority order	Zero (plus execution time for interrupt routines)	Very good	Most complex (although much of the complexity is inside the operating system itself)

Criterios de modularidad

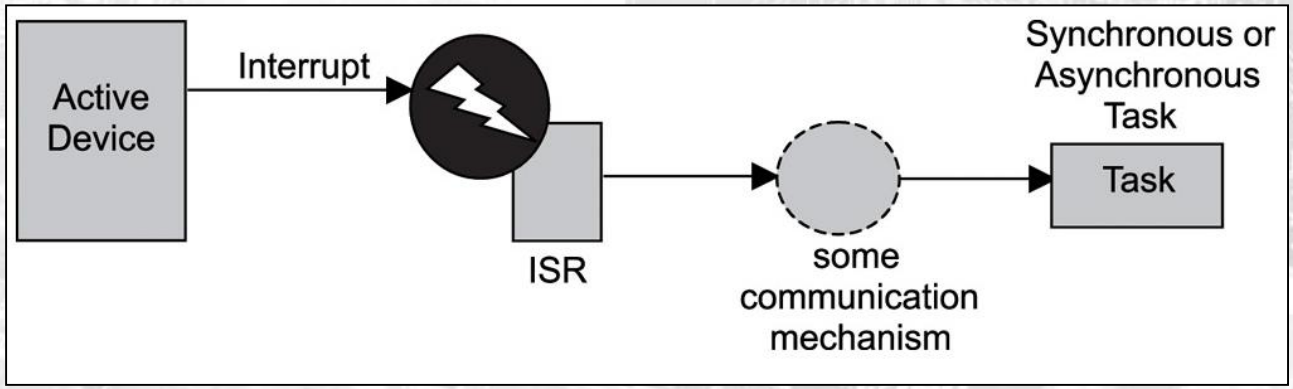
Criterios de modularidad

- Analizar las dependencias entre los dispositivos y el resto del sistema constituye un factor clave al momento de descomponer el sistema en tareas e ISRs.



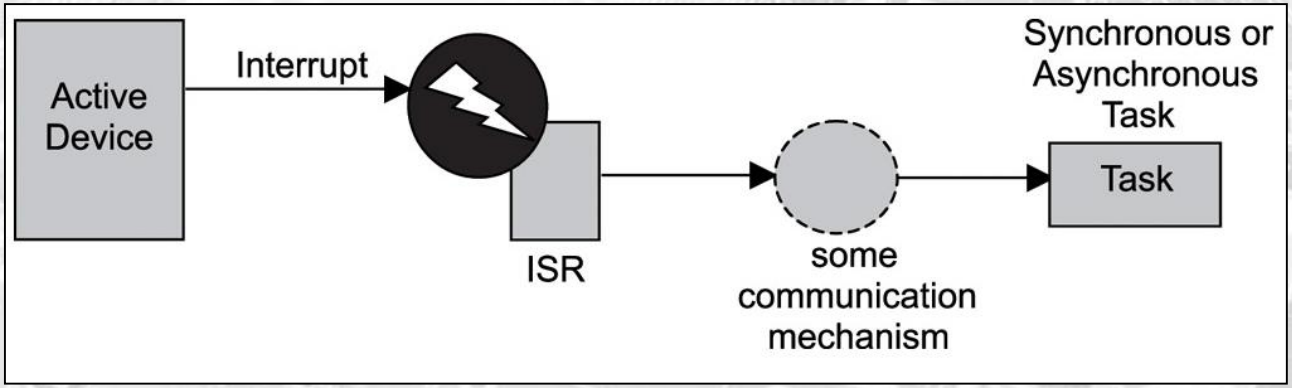
Criterios de modularidad

- Los dispositivos activos producen interrupciones
- La atención de este tipo de dispositivos tiene fuertes restricciones temporales (ISRs) y existe un mecanismo de comunicación inter-tarea entre la ISR y la tarea.



Criterios de modularidad

- Ejemplos de tareas activas:
  - Tareas asincrónicas: reciben eventos aperiódicos.
  - Tareas sincrónicas: reciben eventos periódicos o su operación está sincronizada con otras tareas del sistema.
  - Tareas que controlan el acceso a un dispositivo compartido.
  - Tareas de despacho de eventos hacia otras tareas.



Criterios de modularidad

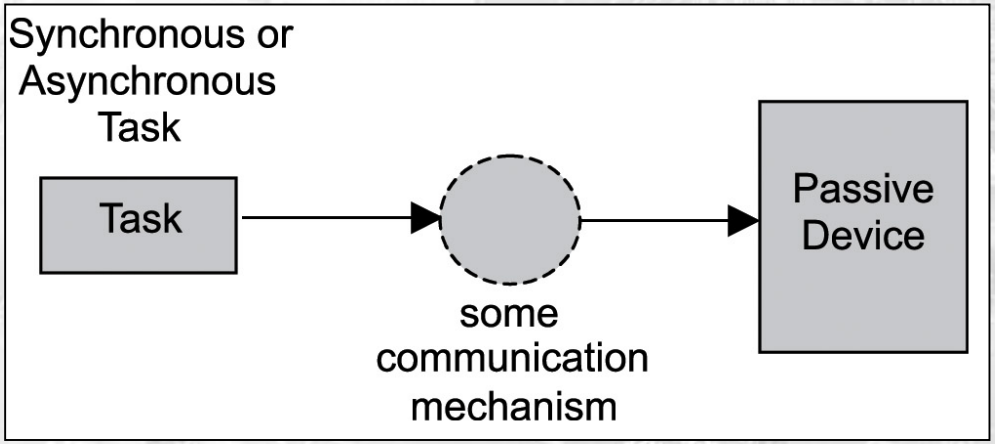
- Recomendaciones:
  - Asignar tareas separadas para cada dispositivo de E/S que presente un comportamiento asincrónico (el dispositivo fija la tasa de transferencia).
  - Combinar tareas para dispositivos que interrumpen infrecuentemente o cuyos deadlines (tiempo real) son largos.
  - Asignar diferentes tareas a dispositivos que presentan tasas de transferencia dispares.
  - Asignar prioridades más altas a las tareas asociadas a dispositivos que generan interrupciones (en general estos dispositivos requieren tiempos de respuesta más demandantes).

Criterios de modularidad

- Recomendaciones:
  - Asignar una tarea que gestione el acceso a dispositivos compartidos (dispositivos que deben ser accedidos por múltiples tareas).
  - Utilizar tareas de despacho de eventos para aquellos dispositivos que requieran proveer información a múltiples tareas.

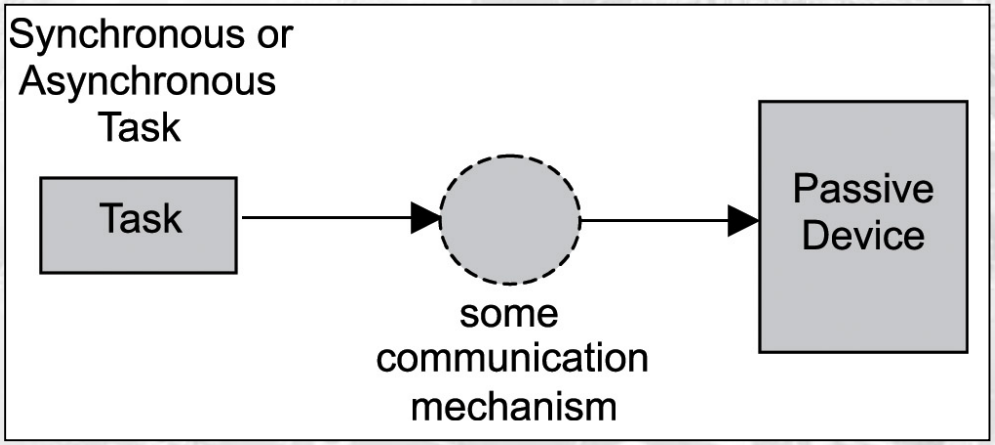
Criterios de modularidad

- Los dispositivos pasivos no producen interrupciones
- El control en la comunicación está en manos de la tarea (normalmente polling o respuesta a eventos de otras tareas).



Criterios de modularidad

- Ejemplos de tareas pasivas:
  - Tareas que operan sobre el dispositivo esporádicamente.
  - Tareas que realizan polling a intervalos periódicos.
  - Tareas que controlan el acceso a dispositivo compartido.
  - Tareas de despacho de eventos (originados a partir del polling) hacia otras tareas del sistema.



Criterios de modularidad

- Recomendaciones:
  - Asignar una tarea para manejar dispositivos que se operan eventualmente o cuyos deadlines no son urgentes.
  - Asignar tareas individuales para cada dispositivo que necesite ser sondeado/escrutado a diferentes tasas (evitar under/oversampling).
  - Disparar el polling a dispositivos pasivos mediante timers (evitar delays en SW que podrían demorarse ante la ocurrencia de interrupciones).
  - Incrementar la prioridad de las tareas que realizan polling de mayor frecuencia (sólo hasta el nivel necesario, sino se corre el riesgo de sobre-ocupar el cpu).

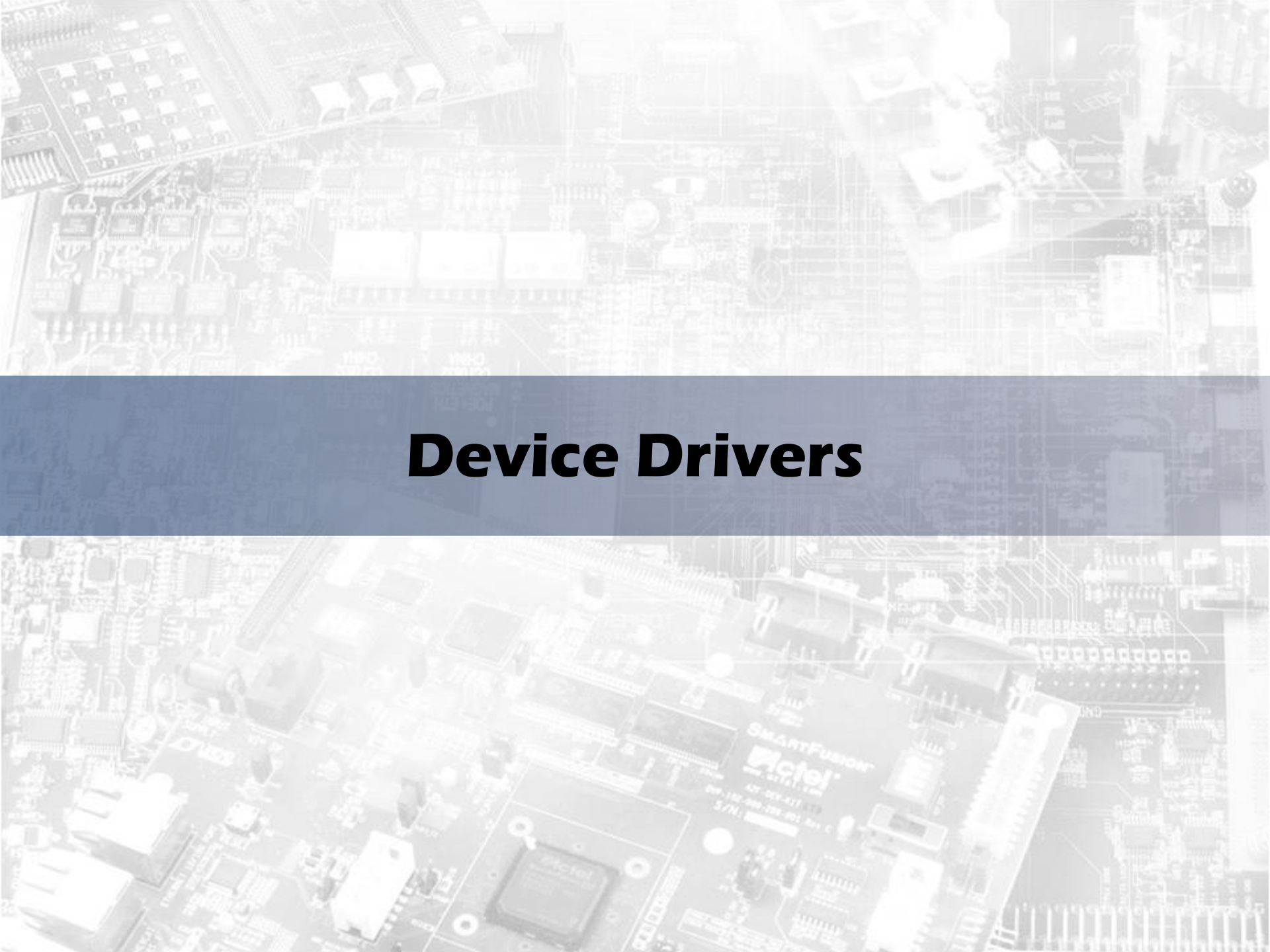
Criterios de modularidad

- Una vez identificadas las tareas e ISRs, y teniendo en cuenta sus prioridades relativas es necesario:
  - identificar las dependencias entre eventos (tareas que producen información que otras necesitan)
  - identificar dependencias temporales:
    - identificar actividades críticas y urgentes
    - identificar las tasas de ejecución de tareas periódicas
    - identificar cohesión temporal: aquellas porciones de código que se ejecutan en un mismo momento (por ej. dependen de un mismo evento) pueden agruparse en una tarea para reducir la sobrecarga.
  - Identificar actividades limitadas por CPU (intensivas en procesamiento) y asignarles prioridades bajas.

Criterios de modularidad

- Una vez identificadas las tareas e ISRs, y teniendo en cuenta sus prioridades relativas es necesario:
  - identificar cohesión funcional: agrupar en tareas funciones relacionadas o tareas que comunican grandes cantidades de datos.
  - identificar tareas de propósitos específicos: agrupar tareas que se combinan para lograr un propósito particular y tratarlas como unidades funcionales separadas.
  - identificar cohesión secuencial: agrupar tareas que revisten un ordenamiento secuencial en una única tarea, reduce los recursos consumidos por las mismas y simplifica el diseño.

Device Drivers

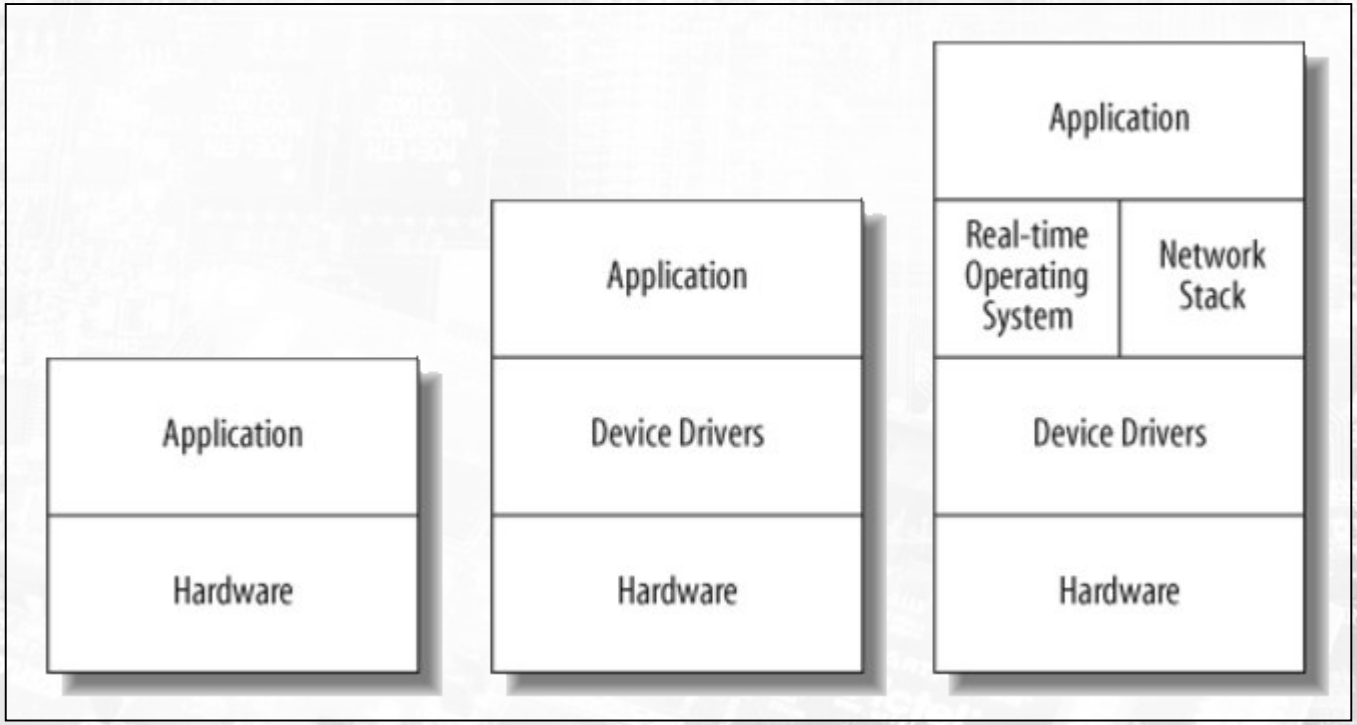


Device Drivers

- Son porciones de software destinadas a aislar al resto del sistema de los detalles específicos de un dispositivo particular de hardware.
- El acceso al dispositivo se realiza exclusivamente vía el driver (protección del recurso) mediante una interfaz que abstrae de los detalles de implementación (abstracción de hardware).
- Se dan tanto en presencia de sistemas operativos (seguirán el modelo de drivers para el sistema particular) como en su ausencia (presentarán la forma de una API que el desarrollador podrá integrar en su sistema).

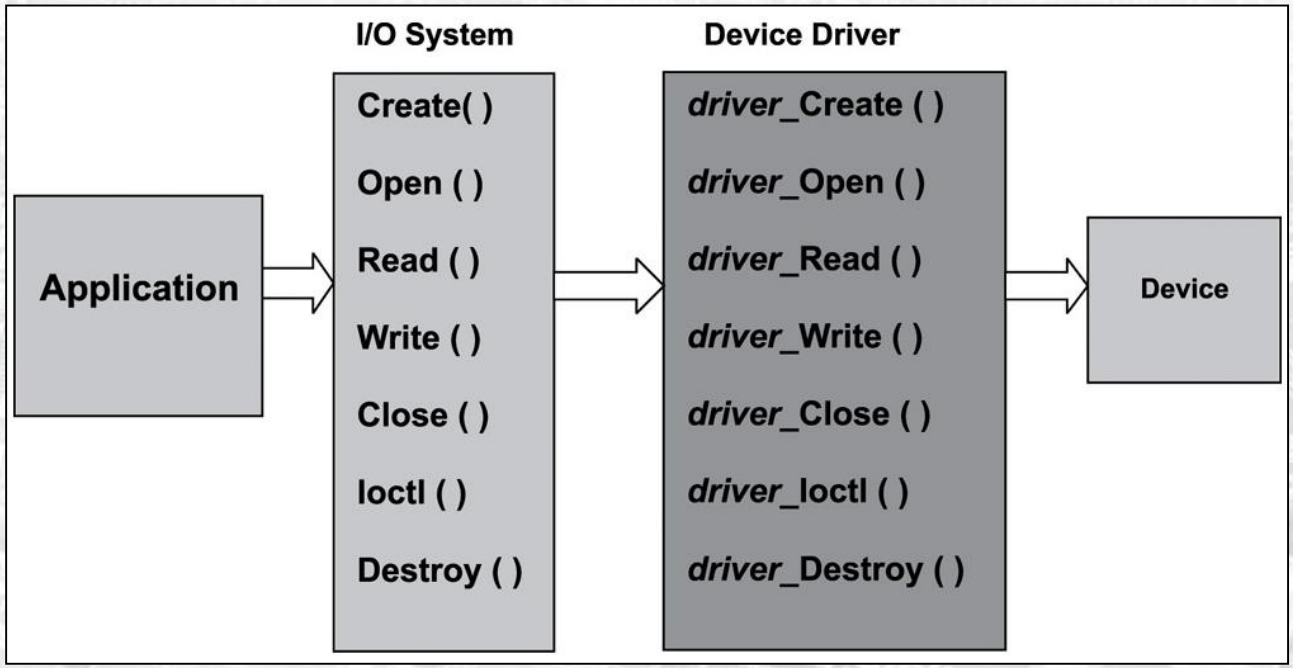
Device Drivers

- Capas de software en un sistema embebido:



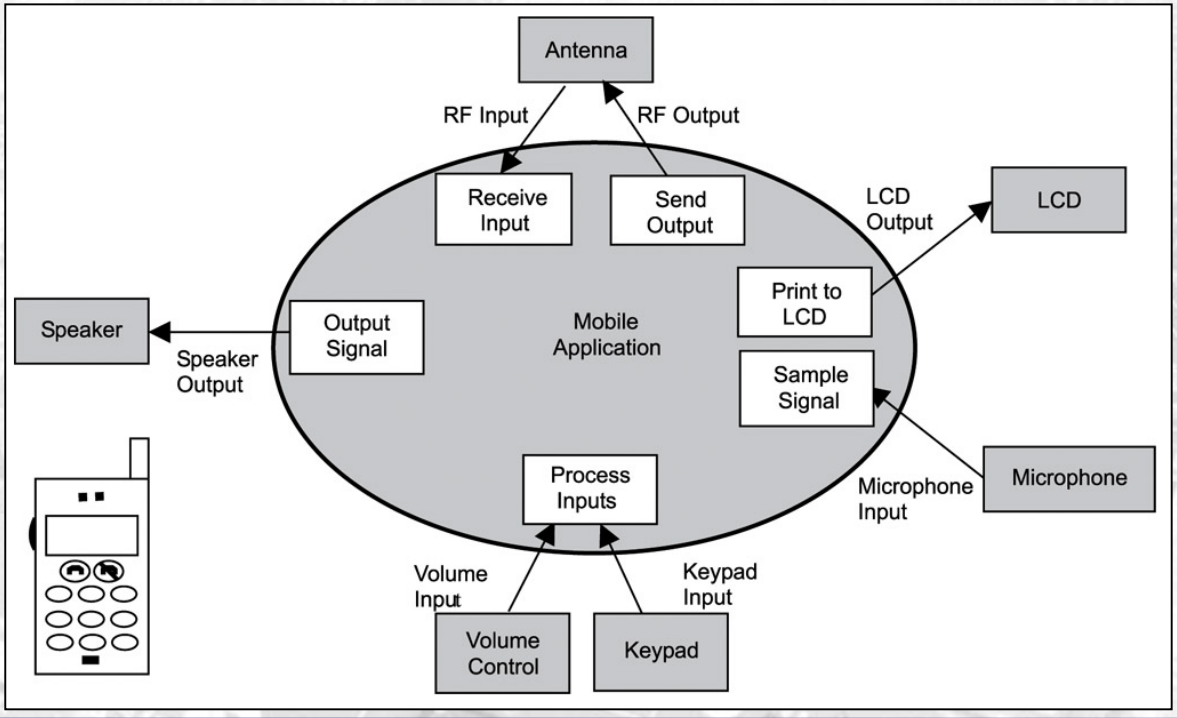
Device Drivers

- Los drivers como capas de abstracción de hardware.



Device Drivers

- Ej: con excepción de sistemas muy simples, la atención de cada dispositivo (ISRs y tareas) se encapsula en device drivers (módulo/subsistema).



Referencias

- Barr, M., Massa, A. Programming Embedded Systems: With C and GNU Development Tools, 2nd Edition. O'Reilly Media. 2006. ISBN: 978-0596009830. Capítulo 10.
- Li, Q., Yao, C. Real-time concepts for Embedded Systems. CMP. 2003. ISBN: 978-1578201242. Capítulos 5, 14 y 15.
- Simon, D. An Embedded Software Primer. Addison-Wesley Professional. 1999. ISBN: 978-0201615692. Capítulo 5.