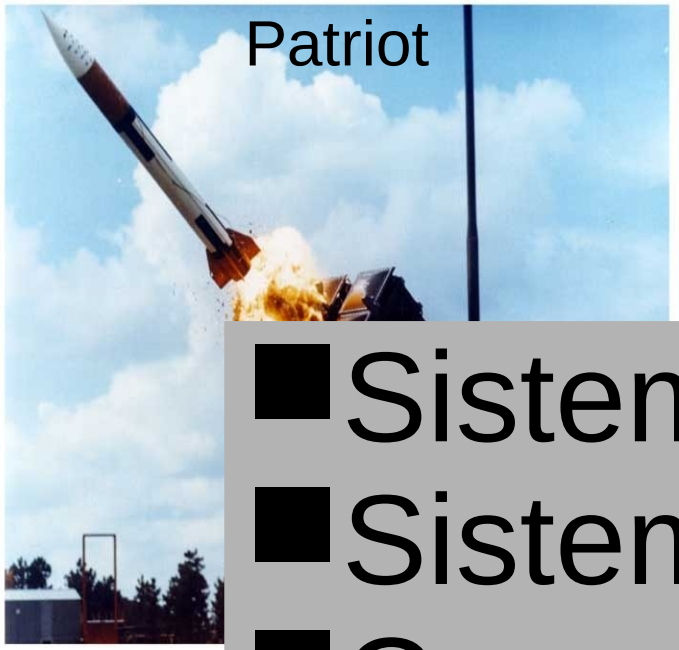
Desarrollo de drivers y aplicaciones para FreeRtos

Marzo 2010
Ing. Marcelo Lorenzati
http://sistemasembebidos.com.ar

Agenda

- Sistemas de tiempo Real y Sistemas embebidos
- Introducción a los RTOS embebidos y a FreeRTOS
- Ports, manejo de Memoria y configuración
- Control del Kernel y de la Tarea
- Sincronización y Comunicación entre tareas
- El por qué del Driver
- Partes de un Driver
- Desarrollo de un Driver
- Tipos de Drivers
- Drivers desarrollados
- Ejemplo de uso – audio player con FreeRTOS
- Conclusiones
- Bibliografía
- Preguntas

Sistemas de tiempo Real y Sistemas embebidos /1



- Sistemas en tiempo real
- Sistemas embebidos
- Complejos
- Tuvieron fallas

Therac-25

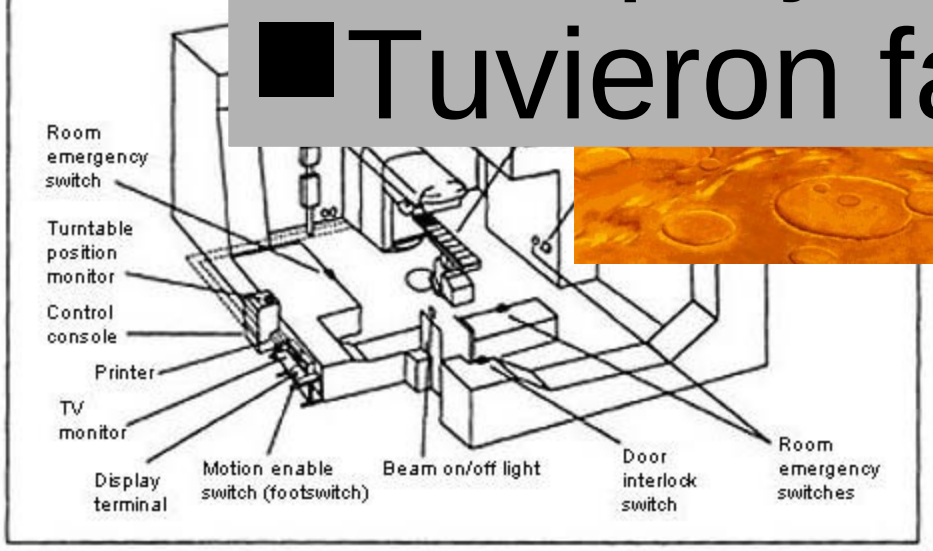
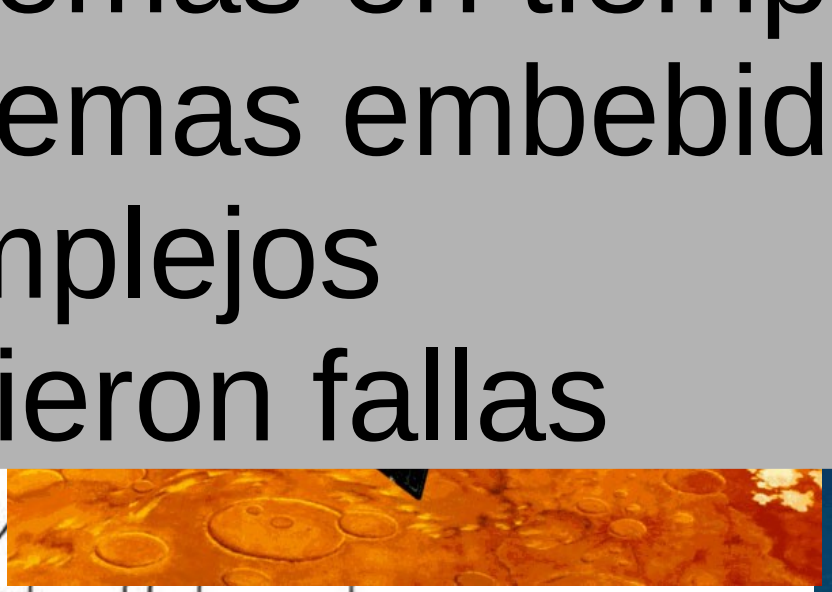
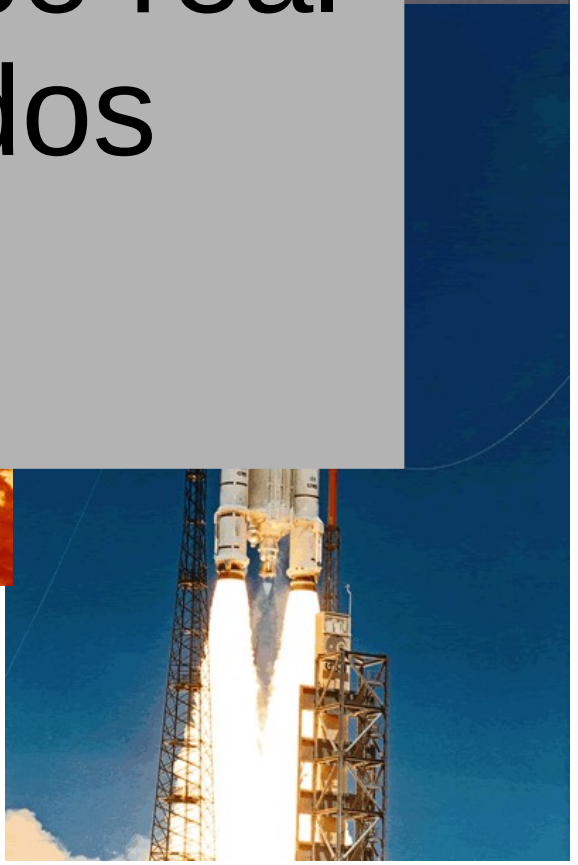
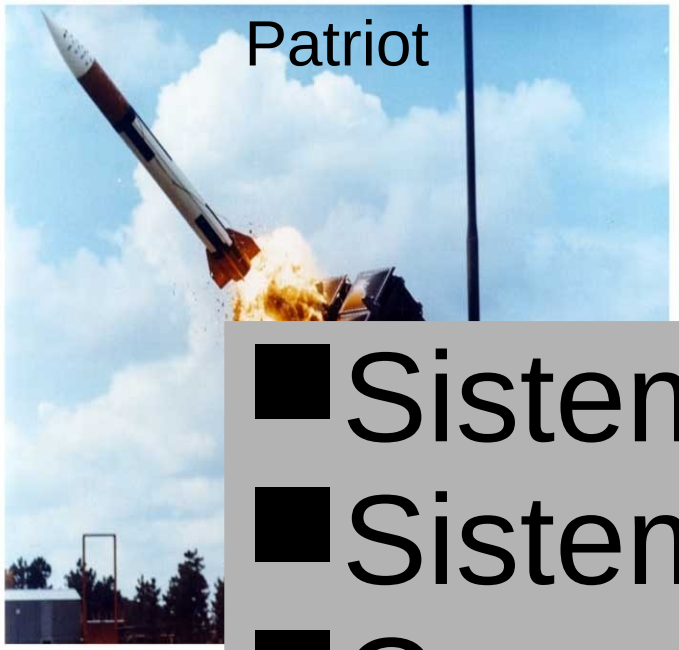


Figure 1. Typical Therac-25 facility





Sistemas de tiempo Real y Sistemas embebidos /1



- Sistemas en tiempo real
- Sistemas embebidos
- Complejos
- Tuvieron fallas

Therac-25

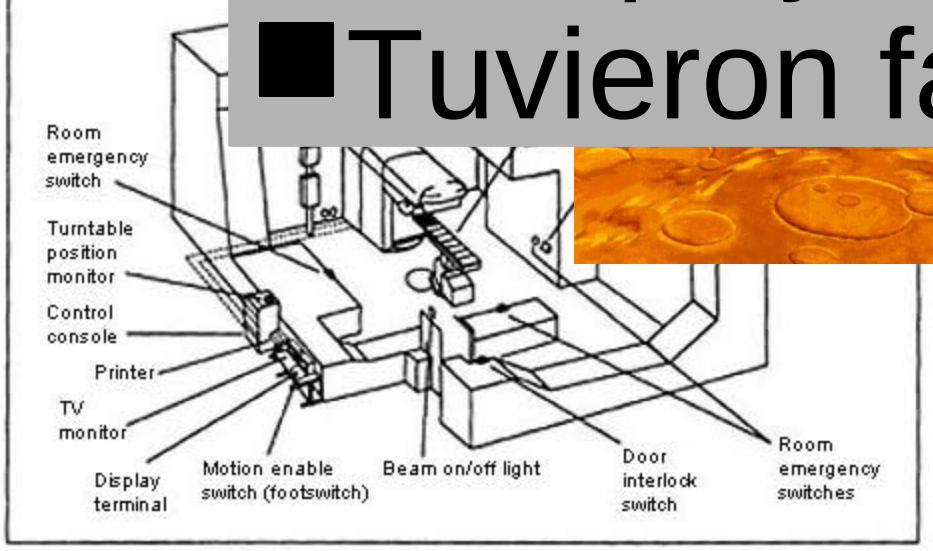
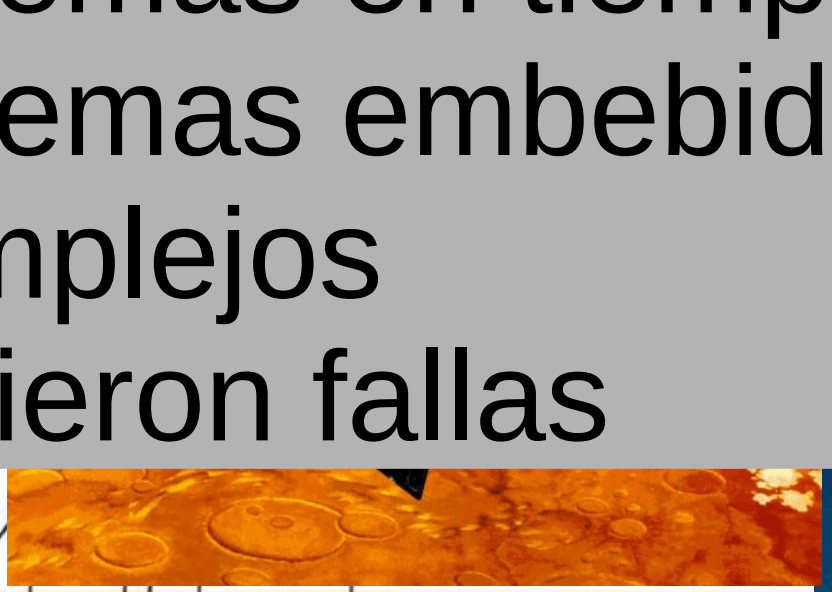
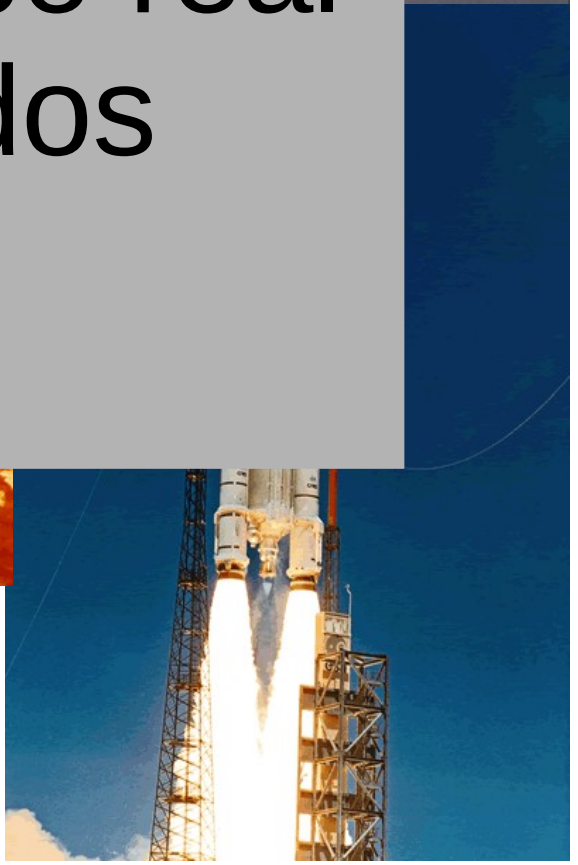


Figure 1. Typical Therac-25 facility





Sistemas de tiempo Real y Sistemas embebidos /2

Problemas específicos de los sistemas de tiempo real

- Tiempos de respuesta estrictos
- Impredictibilidad del comportamiento del entorno
- Necesidad de accion rapida ante errores

Problemas específicos de los sistemas de software

- Atomicidad
- Estancamiento
- Sincronizacion
- Concurrencia
- Condiciones de carrera
- Secciones criticas
- Desbordes de memoria
- Paralelismo
- Tiempos de desarrollo del sistema
- Mantenimiento del sistema

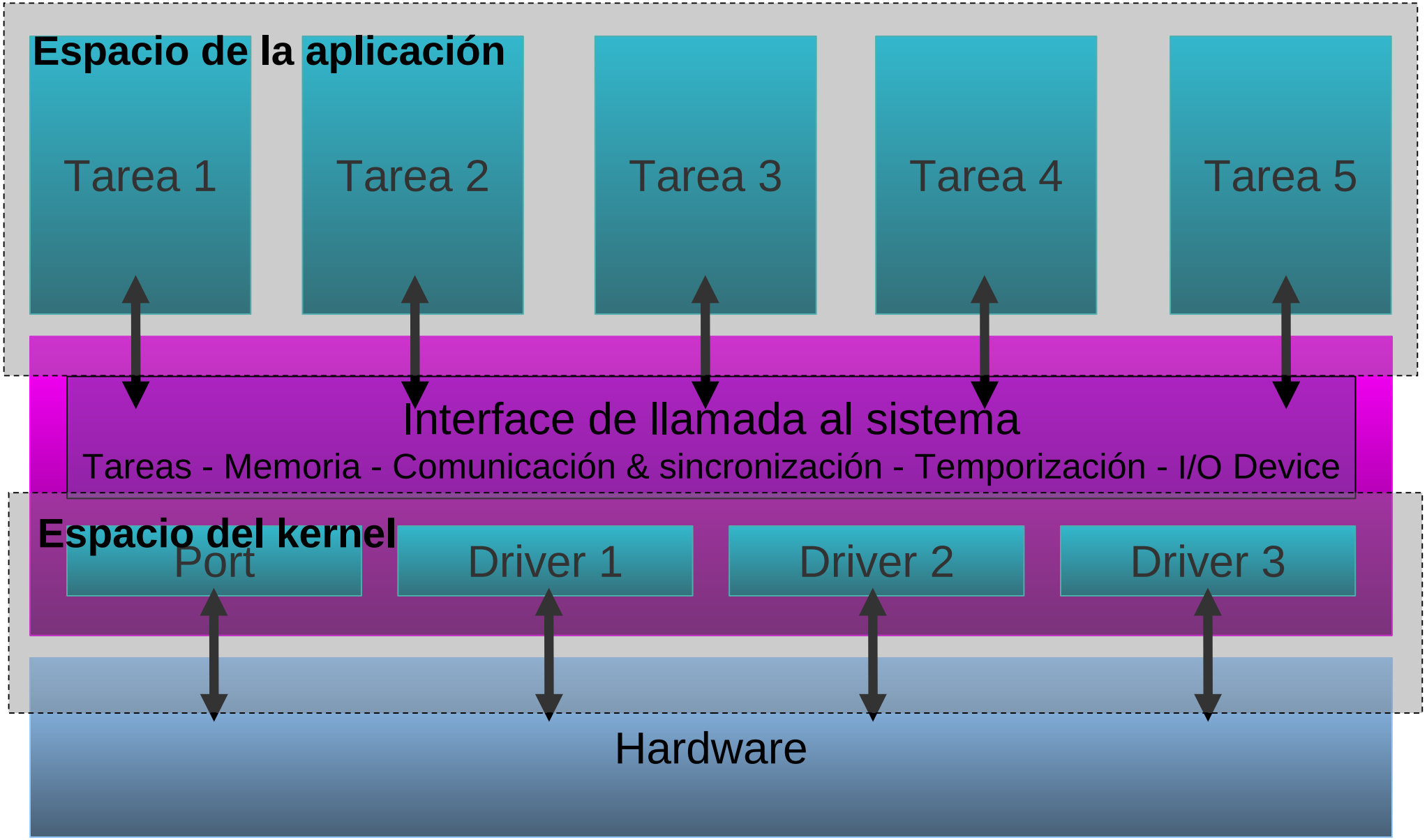
Problema específicos de los sistemas embebidos

- Parte integral de un sistema mas grande
- Heterogeneidad de dispositivos
- Memoria de datos y de programa reducida
- Alto acoplamiento del código con el Hardware
- Baja potencia de computo
- Monoliticos

Introducción a los RTOS embebidos /1

- Variante de un sistema operativo, donde se asegura un resultado computacional en un periodo predecible de tiempo.
- RTOS para sistemas embebidos! No sistemas complejos de tiempo real *
- Dominio total del algoritmo de scheduling
- Acceso rápido al hardware
- Tiempos reducidos de reaccion (ms vs ns)
- Capacidad de priorización
- Generalmente diseñados para funcionar con memoria reducida y baja potencia de computo.
- Suelen ser monolíticos,en una sola imagen binaria se alberga el kernel y las aplicaciones
- Drivers simples de una sola capa,lo que los hace mas rápidos comparado con kernels que usan drivers multicapa.

Introducción a los RTOS embebidos /2



Introducción a los RTOS embebidos /3

Otros RTOSes:

RTLinux
uCLinux
LynxOS
eCos
Symbian OS
BeRTOS
FreeOSEK
OpenRTOS
PICOS18
UNIX-RTR
µnOS
Helium
uCOS-III
Vxworks
Salvo
QNX
ThreadX
EmbOS
Windows CE
INTEGRITY
RTAI
TRON
VelOSity
CMX

Introducción a FreeRTOS

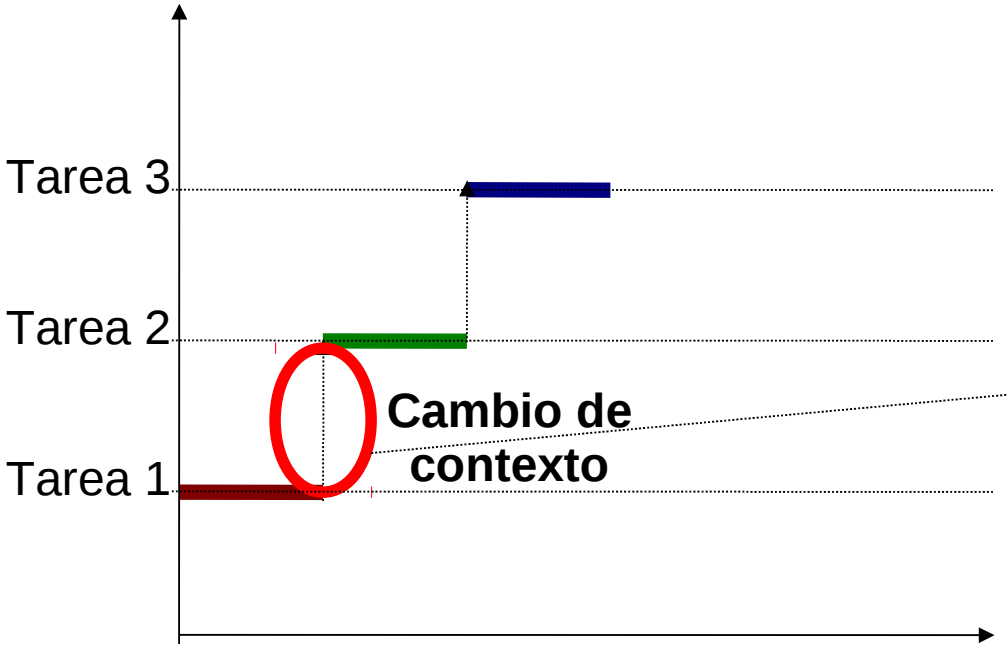
Especificaciones:

- Micro kernel que provee un sistema operativo en tiempo real priorizable, tanto cooperativo, “preventivo” o mixto.
- Pensado para microcontroladores , cuenta con mas de 23 ports avalados para las diferentes arquitecturas y compiladores.
- API totalmente documentada y con ejemplos.
- Escrito en su mayoria en C, es liviano en tamaño de codigo y en uso de memoria RAM.
- Ambiente de desarrollo multiplataforma, de codigo abierto y de libre uso.

Ports /1

- Un port, es una porcion específica del codigo del sistema operativo.
- Se lo conoce tambien como Board Support Package
- Conecta la logica del RTOS con el hardware especifico del procesador.
- Debe tener en cuenta las especificaciones del compilador que se usa.
- El diseño de un port de un RTOS no es trivial.

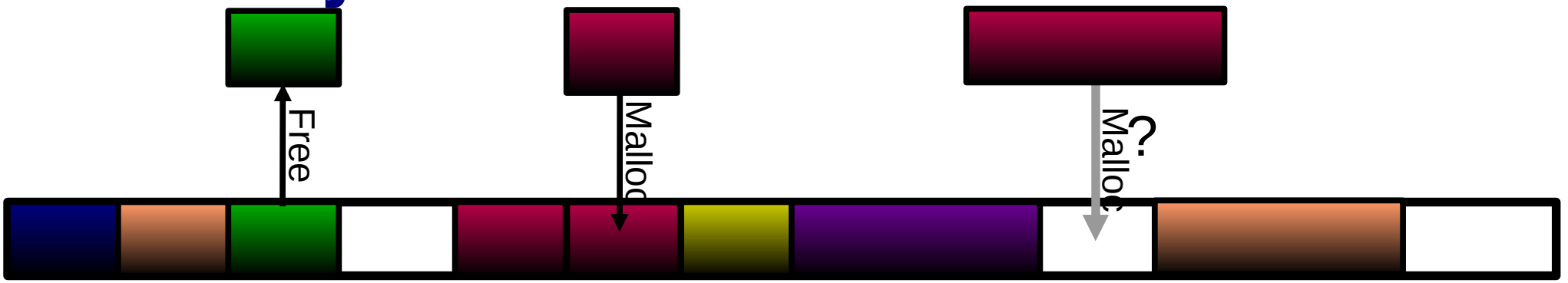
Ports /2



- Especialización de metodos de arranque
- Control del timer de sistema
- Control de la pila de memoria
- Control de las zonas criticas
- Control de los registros del procesador

(BCT)
Bloque de control de la tarea genérico

Manejo de Memoria



- Parte integral del sistema operativo.
- Usado por el RTOS para la creacion/destruccion de tareas, semaforos y colas.
- Soporta multiples modelos:
  - Modelo simple (asigna memoria pero no libera)
  - Modelo común (asigna y libera pero no ordena)
  - Modelo mejorado (reordena)
- Implementacion de FreeRTOS
  - Malloc() → pvPortMalloc()
  - Free() → pvPortFree()

Configuración

FreeRTOSConfig.h

Modifica en tiempo de compilación el modo de ejecucion del RTOS.

- configUSE_PREEMPTION
- configCPU_CLOCK_HZ
- configTICK_RATE_HZ
- configMAX_PRIORITIES
- configMINIMAL_STACK_SIZE
- configTOTAL_HEAP_SIZE
- configMAX_TASK_NAME_LEN
- configUSE_USE_MUTEXES
- configUSE_CO_ROUTINES
- #define INCLUDE_vTaskDelete 1

Control en ejecución del kernel /1

Metodos de distribución del procesador:

- Subdivision en el tiempo
- Basado en eventos

FreeRTOS soporta **AMBOS**

Algoritmos de administracion de tareas:

- Cooperativo (la tarea relega el procesador)
- Preventivo (el OS administra el procesador)

Priorizacion de tareas:

- FreeRTOS permite controlar el numero de prioridades
- La tarea de mas alta prioridad es la que se ejecuta
- Posibilidad de inversion de prioridades en el modelo cooperativo
- FreeRtos permite prioridades iguales
  - Round Robin

Control en ejecucion del kernel /2

Modifican el manejador de tareas del RTOS en tiempo de ejecucion.

- taskYIELD()
- taskENTER_CRITICAL()
- taskEXIT_CRITICAL()
- taskDISABLE_INTERRUPTS()
- taskENABLE_INTERRUPTS()
- vTaskStartScheduler()
- vTaskEndScheduler()
- vTaskSuspendAll()
- xTaskResumeAll()

Control de la tarea /1

- Una **TAREA** es la unidad basica de ejecucion de un RTOS.
- Las tareas se ejecutan con su propio **contexto** independiente de cualquier otra **instancia**, con lo que su consumo de memoria es alto y su tiempo de ejecucion considerable.
- Pueden existir multiples instancias de la misma tarea.
- Las co-rutinas comparten el mismo contexto entre diferentes instancias (consumen una sola pila), son de ejecucion rapida pero tienen restricciones en cuanto a donde y como se pueden ejecutar y solo funcionan en modo cooperativo.
- Si no esta instanciada no ocupa memoria ni tiempo de procesador

Control de la tarea /2

Es interrumpible por:

- IRQ de Hardware
- Cambio de prioridades
- Cambio de contexto del RTOS
- Cambios de estado forzados desde la tarea

APIs que modifican el flujo de control de la tarea e indirectamente del kernel:

- XtaskCreate / VtaskDelete
- VtaskDelay / vTaskDelayUntil
- UxTaskPriorityGet / vTaskPrioritySet
- VtaskSuspend / vTaskResume / vTaskResumeFromISR

Control de la tarea /3

Arranque del RTOS y de una Tarea “Demo” y destruccion:

```
static void vDemo( void *pvParameters )
{
    for ( ; ; )
    {
        vTaskDelay( 1000 );
        vTaskDelete(xHandle);
    }
}

/*Main Program*/
int main( void )
{
    prvSetupHardware();
    xTaskCreate( vDemo, "Demo", STACK_SIZE, NULL, tskIDLE_PRIORITY, xHandle );
    vTaskStartScheduler();
    return 0;
}
```

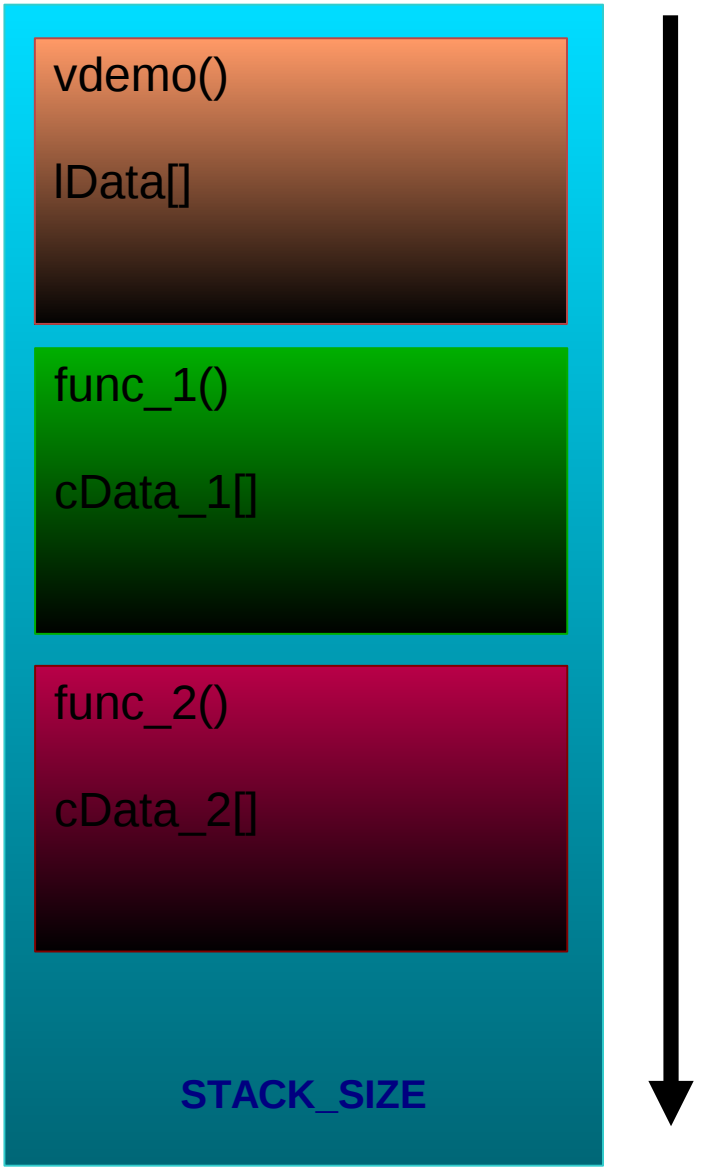
Control de la tarea /4

Uso del Stack de Memoria:

```
static void vDemo( void *pvParameters )
{
    portLONG  lData[1000];
    for ( ; ; )
    {
        func_1();
    }
}

void func_1(void)
{
    portCHAR cData_1[1000];
    func_2();
}

void func_2(void)
{
    portCHAR cData_2[1000];
    process();
}
```



Sincronización de la tarea /1

- Una tarea no es util si no interactúa con otros procesos.
- Se requiere que permita **sincronizar**, **proteger** y **compartir**.
- FreeRTOS ofrece **Semaforos** y **Mutex**.
- Pueden correr en una tarea o en código de interrupciones.
- Los **semaforos** permiten la **sincronización** entre dos tareas, siendo una la que espera (Take) y otra que la dispara o libera(give)
- Los **semaforos** pueden ser **binarios** o **contadores**.
  - Dar incrementa
  - Si esta en su maximo no incrementa
  - Tomar decrementa
  - Si es cero y se lo toma bloquea
- Los **Mutex** son una especializacion de los semaforos binarios, permiten la proteccion para el acceso de un **recurso** comun serializado.

Sincronización de la tarea /2

API's de sincronización:

- VsemaphoreCreateBinary
- VsemaphoreCreateCounting
- XsemaphoreCreateMutex
- XsemaphoreCreateRecursiveMutex
- XsemaphoreTake
- XsemaphoreTakeRecursive
- XsemaphoreGive
- XsemaphoreGiveRecursive
- xSemaphoreGiveFromISR

Sincronización de la tarea /3

Ejemplo de captura de recurso compartido:

- **Creación de recurso:**

```
vSemaphoreCreateBinary(sem1);
```

- **Captura y liberación de recurso (2 procesos iguales):**

```
if (xSemaphoreTake(sem1, (portTickType) 1000 ) != pdTRUE)
{
   return ERROR;
}

doProcess();

xSemaphoreGive(sem1);
```

Sincronización de la tarea /4

Ejemplo de sincronización, productor y consumidor:

Creación de recurso:

```
VsemaphoreCreateCounting(sem1, 0);
```

Consumidor:

```
if (xSemaphoreTake(sem1, (portTickType) portMAX_DELAY ) != pdTRUE)
{
    return ERROR;
}
processData();
```

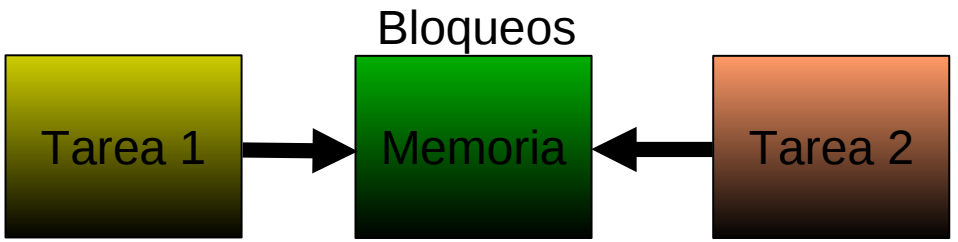
Productor:

```
feedData();
xSemaphoreGive(sem1);
```

Comunicación entre tareas /1

Provee los medios para compartir datos entre tareas aisladas.

- Colas de mensajes:
- Tuberias (Resultado de un proceso es entrada para otro)
  - Puede variar el tamaño del mensaje
  - Stout / stdin
- Mailbox (Pub / sub)
- Semaforos
- Memoria compartida



- A menor nivel de abstraccion mayor probabilidad de errores
- Los recursos del RTOS son caros, usar con criterio.

Comunicación entre tareas /2

Colas de mensajes:

Creacion de una cola:

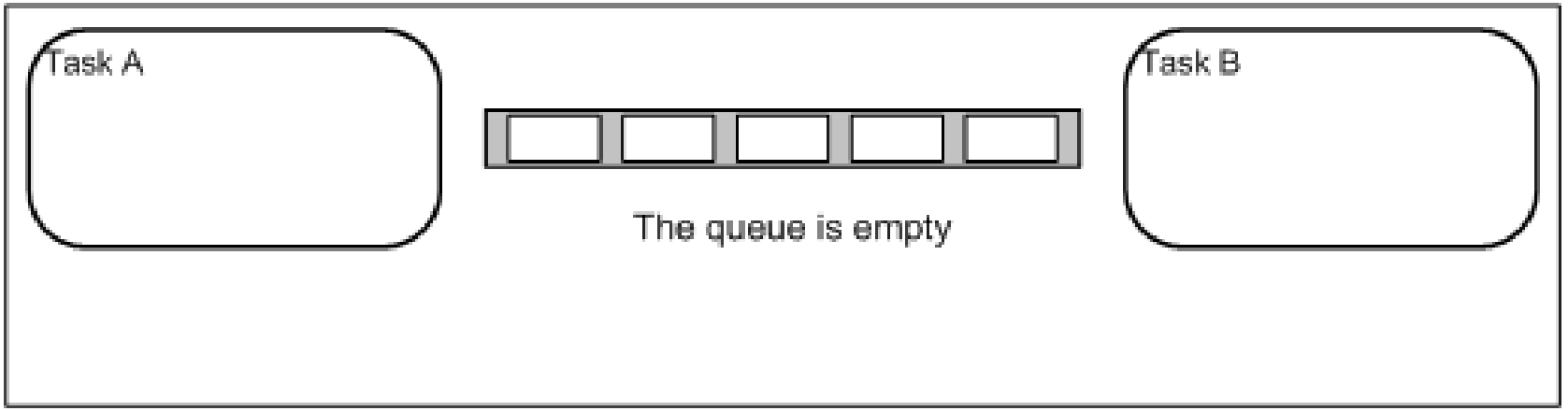
xQueue = xQueueHandle xQueueCreate(lenght, size);

Envio de datos por cola:

xQueueSend(xQueue, itemToQueue, timeOut);

Recepcion de datos por cola:

xQueueReceive(xQueue, itemToDeQueue, timeOut);



Resumen

- **Tarea no instanciada no ocupa tiempo en el manejador (scheduler) ni RAM.**
- **Tarea en demora o en espera de eventos (semaforo o cola) no consume tiempo de procesador (Idle waiting)**
- **Evitar busy waiting!!**

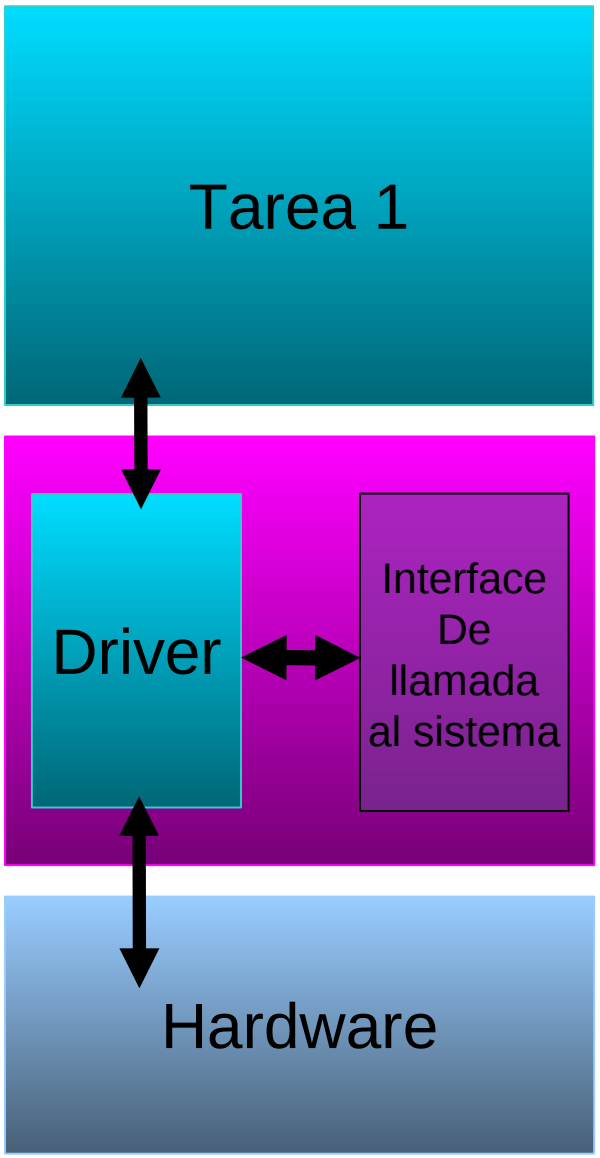
```
While ( !done ) { };
```

- **Crear objetos del RTOS consume tiempo y memoria, usar con criterio!!!**

El por qué del Driver

- Abstraen la logica del dispositivo de la funcionalidad genérica.
- Permite la reusabilidad del driver en diferentes aplicaciones
- Permite la independencia de la aplicacion en diferentes arquitecturas
- Permite sincronizar eventos del hardware con eventos del RTOS
- Permite considerar al procesamiento del hardware como una tarea mas del sistema
- Los periféricos son lentos comparados a la CPU

Partes de un Driver /1



Entre la tarea y el driver se maneja el “que”

•Formado en la mayoria de los casos por los siguientes metodos básicos:

- Init() / Reset()
- Open()
- Close()
- Read()
- Write()

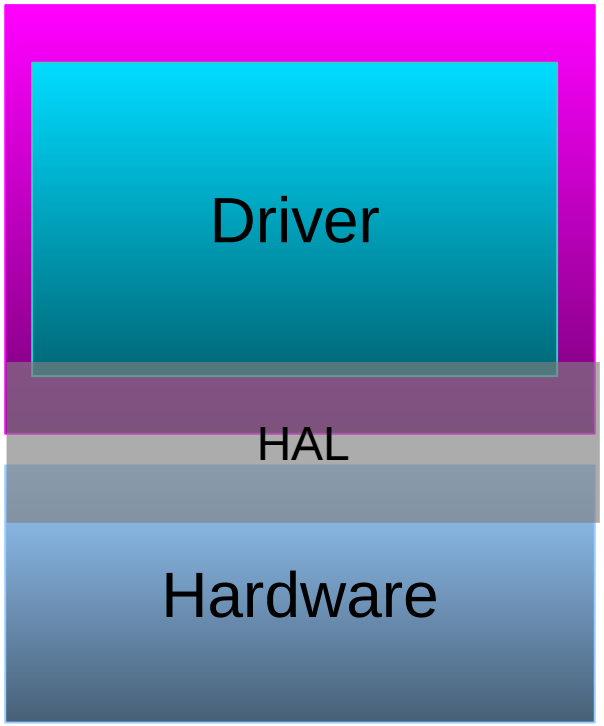
•Metodos persistentes sobre las diferentes arquitecturas

La interfase entre el driver y el hardware maneja el ”como”:

•En la mayoria de los casos esta definida por el uso de:

- I/O de hardware
- Manejo de las interrupciones
- Manejo del DMA
- Manejo de la memoria

Partes de un Driver/2



Capa de Abstracción del hardware (HAL)

Virtualiza el hardware de la plataforma para que diferentes drivers puedan ser portados facilmente en diferentes hardwares.

Puede ser tan simple como define en un archivo de cabeceras (header file) hasta una capa completa de código a la cual se interfacea.

Desarrollo de un driver

Código en espacio de aplicación:

- Llamadas a la interface del driver (APIs)
  - Usan el hilo del programa que lo llama
- Previamente instanciado (como un servicio)
  - Usan su propio hilo
- Pueden ser instanciados y/o removidos del scope del kernel
- La mayoria del procesamiento se encuentra aqui

Código en interrupciones:

- Es irremovible, se ubica en tiempo de compilación
- Es simple y corto
- Solo se toman los datos
- KISS (Keep it simple Silly)

Desarrollo de un driver /1

- **Puntos importantes:**
- Requerimientos para el driver
  - Tiempo de respuesta
  - Capacidades de direccionamiento
- Especificaciones del hardware
  - Encuestado (Polled)
  - Basado en IRQs
  - Basado en DMA
  - Mixtos

Desarrollo de un driver /2

Secciones Criticas:

- Porción de código que debe ser tratada atómicamente
- Una vez entrado en sección critica no se interrumpe
- Generalmente acceden a un recurso de HW
- Deben ser serializables
- Tipos de secciones criticas
  - En espacio de aplicación
  - En espacio del kernel
  - En espacio de interrupciones
- Métodos de protección según severidad
  - Baja: Mutex ()
  - Media: evitar cambio de contexto
  - Alta:
    - Evitar cierto tipo de interrupciones
    - Evitar interrupciones dentro de interrupciones
  - Mas Alta!: Deshabilitar interrupciones globales

Tipos de drivers /1

- Interface humana
  - Entrada
    - Teclados matriciales
    - Pulsadores
    - Touch pads
    - Mouse
  - Salida
    - Pantallas
      - LCD de caracteres
      - LCD Graficas
      - VGA
    - Sonido
      - DACs
      - Codecs
- De comunicaciones
  - Usart
  - SPI
  - I2C
  - I2S
  - USB
- De temporizacion
  - Timers
  - Counters
  - PWM / CCP
- De entrada salida
  - Controladores de memoria
  - Controladores de puertos

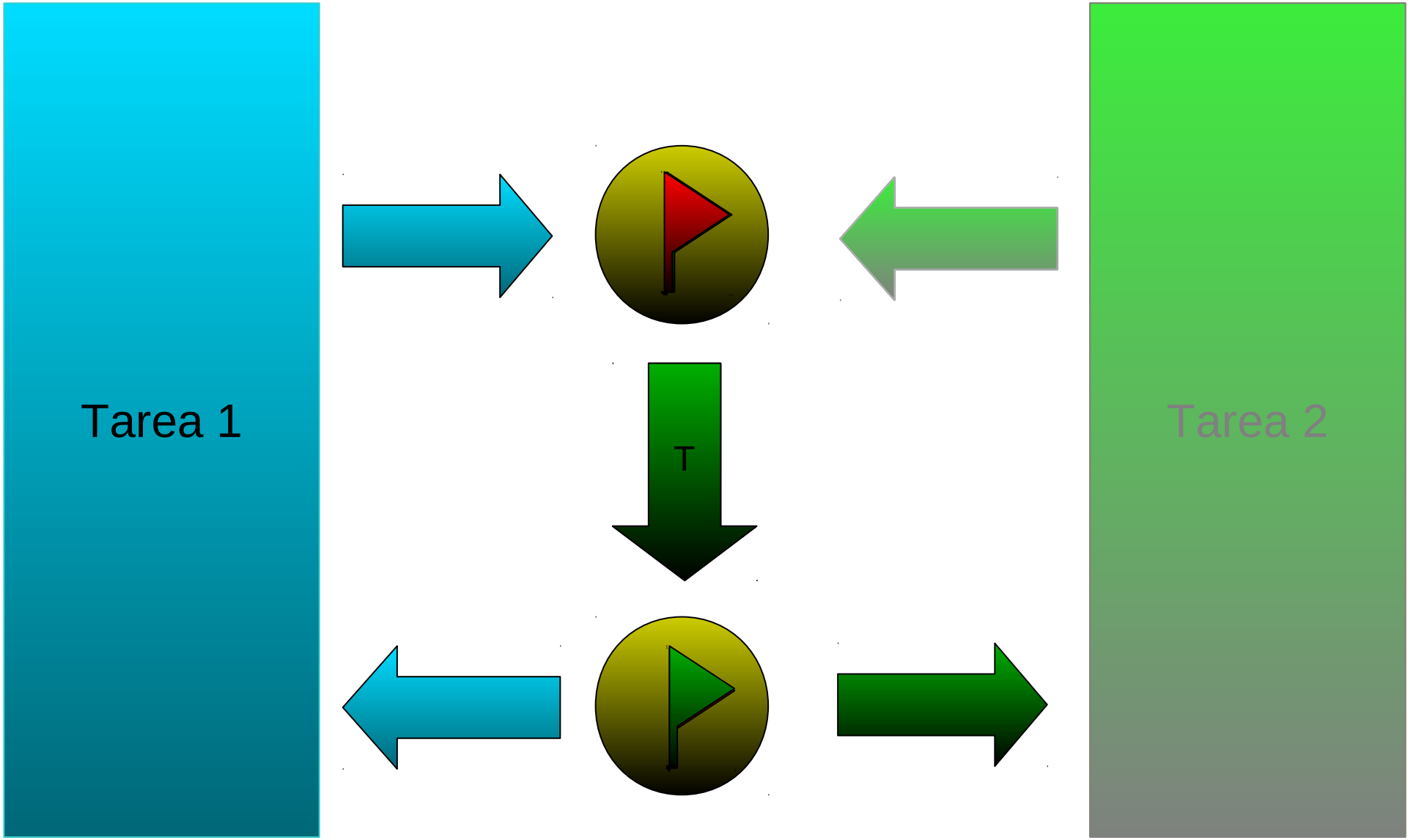
Tipos de drivers /2

Sincrónicos

- Tambien llamados bloqueantes.
- Por diseño la tarea que llama al driver esperara el resultado de la operación por la cual consulto.
- No significa que otras tareas no puedan ocupar el tiempo del procesador.
- Una vez completado se despertara a la tarea en espera y consumira su resultado.
- Son mas simples que otro tipo de drivers.
- Suelen utilizarse semáforos binarios para tal fin.
- Se utilizan al menos 2 semáforos:
  - Uno para garantizar la captura del recurso (mutex)
  - Otro para informar la conclusion de la tarea.

Tipos de drivers /3

Ejemplo driver sincrónico



Tipos de drivers /4

Ejemplo driver sincronico

```
unsigned portCHAR cSendFrame(pI2sSendData pData)
{
    unsigned portCHAR errCode;
    if (xSemaphoreTake(xMutex, (portTickType) xBlockTime ) != pdTRUE) //Intento tomar mutex
    { return ERR_SEM_TIMEOUT;  }
    DoBgProcess(); //Proceso del HW en segundo plano
    if (xSemaphoreTake(xSem, (portTickType) xBlockTime ) == pdTRUE) //Espero que complete
    { errCode = SUCCESS;  }
    else { errCode = ERR_TASK_TIMEOUT; }
    xSemaphoreGive(xMutex);  //Libero Mutex
    return errCode;
}

void vI2sISR(void)
{
    if (completed)
    {
        takeData();
        xSemaphoreGiveFromISR(xSem, &xN);  //Señalo evento de finalización
    }
}
```

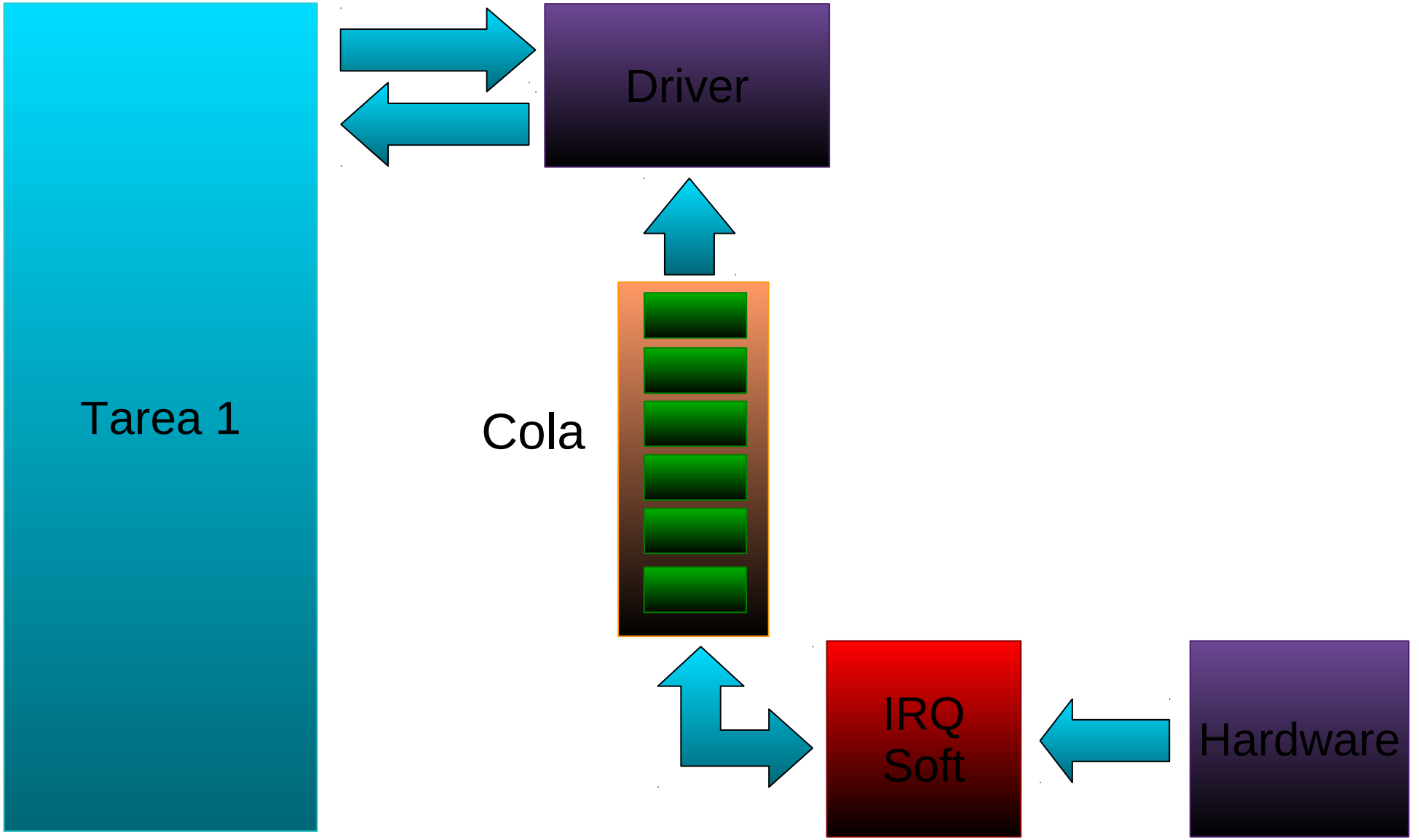
Tipos de drivers /5

Asincrónicos

- Por diseño la tarea que llama al driver puede continuar ejecutando sin esperar el resultado de la operación que pidio.
- Oportunidad para avanzar tareas hasta que la operación este lista.
- Mucho mas complejos de diseñar y a veces innecesarios
- Mas impredecibles en su comportamiento
- Metodos:
  - Poll
  - Señales / Slots
  - Callback

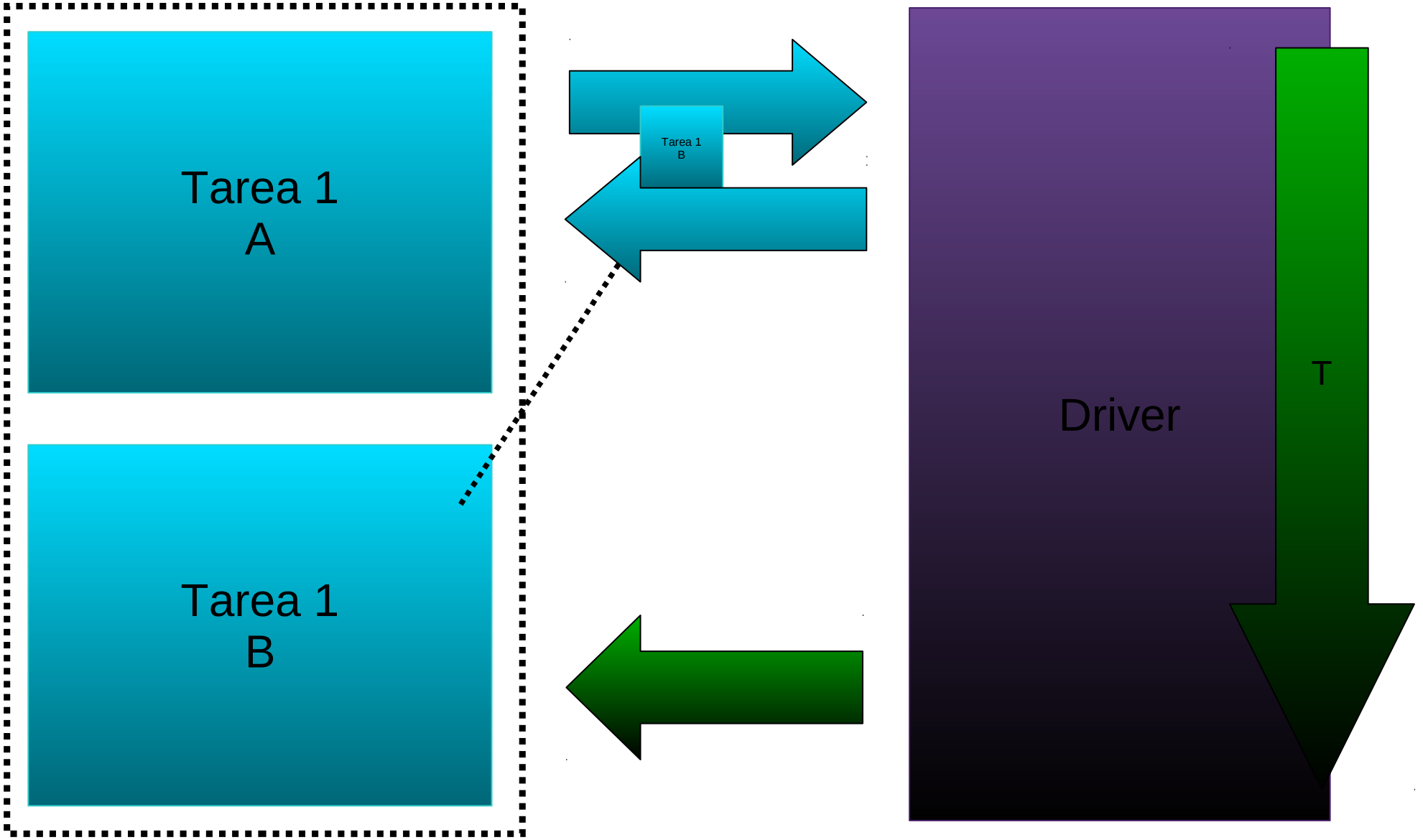
Tipos de drivers /6

Ejemplo de driver asincrónico (Polled):



Tipos de drivers /7

Ejemplo de driver asincrónico (Callback):

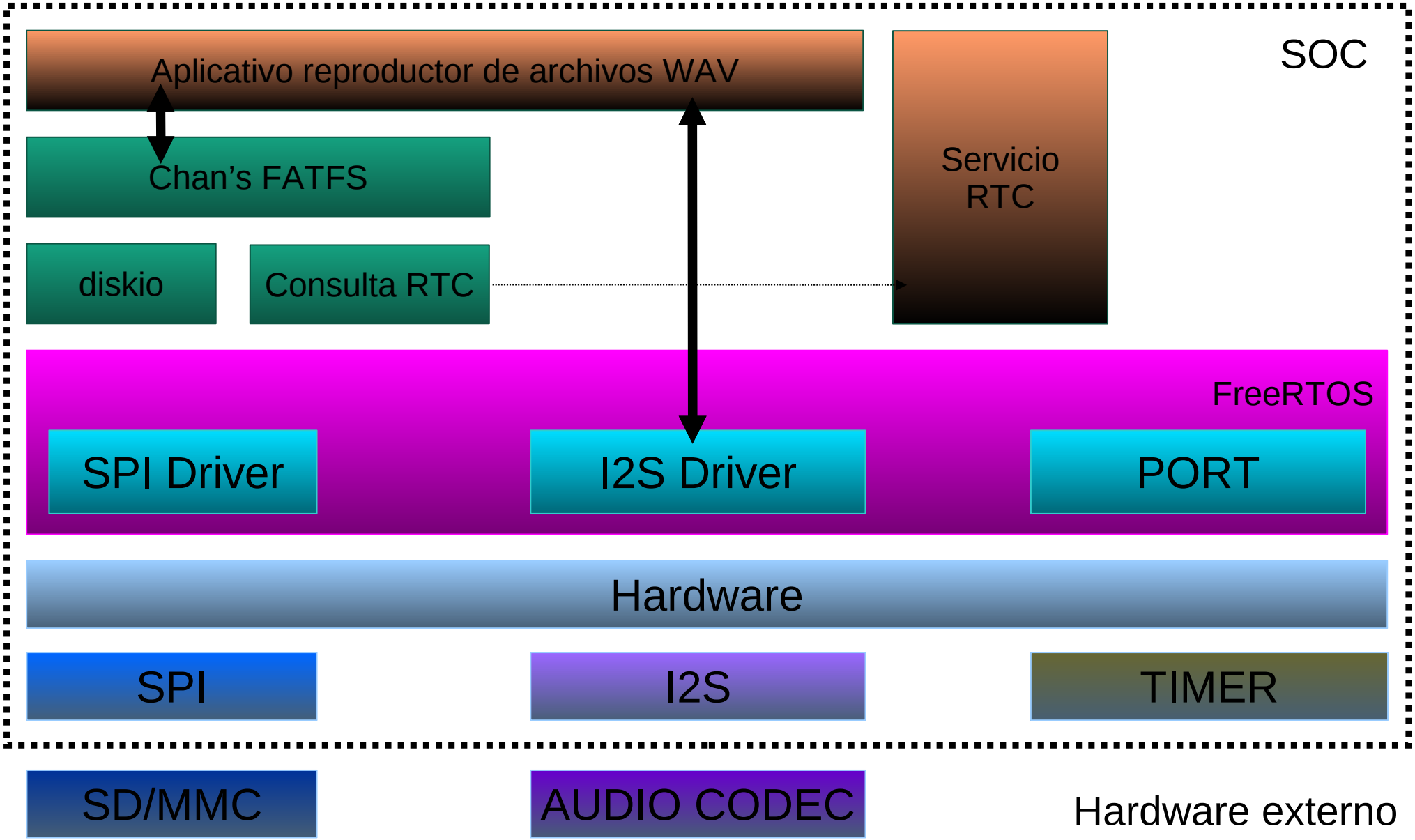


Drivers desarrollados

Lista de drivers desarrollados para FreeRTOS

- Puerto serie
- SPI
- I2S
- Fat
- LCD gráfico
- Timer

Ejemplo de uso – audio player con FreeRTOS



Conclusiones

Los RTOS:

- Consumen mas tiempo de procesador
- Consumen mas memoria
- Reducen la incertidumbre
- Reducen el tiempo de desarrollo
- Mejoran la mantenibilidad
- Mejoran la Abstraccion
- Mejoran la independencia de tareas
- Mejoran la escalabilidad
- Son menos suceptible a errores
- Permiten facilemente la priorizacion
- Facilitan la portabilidad

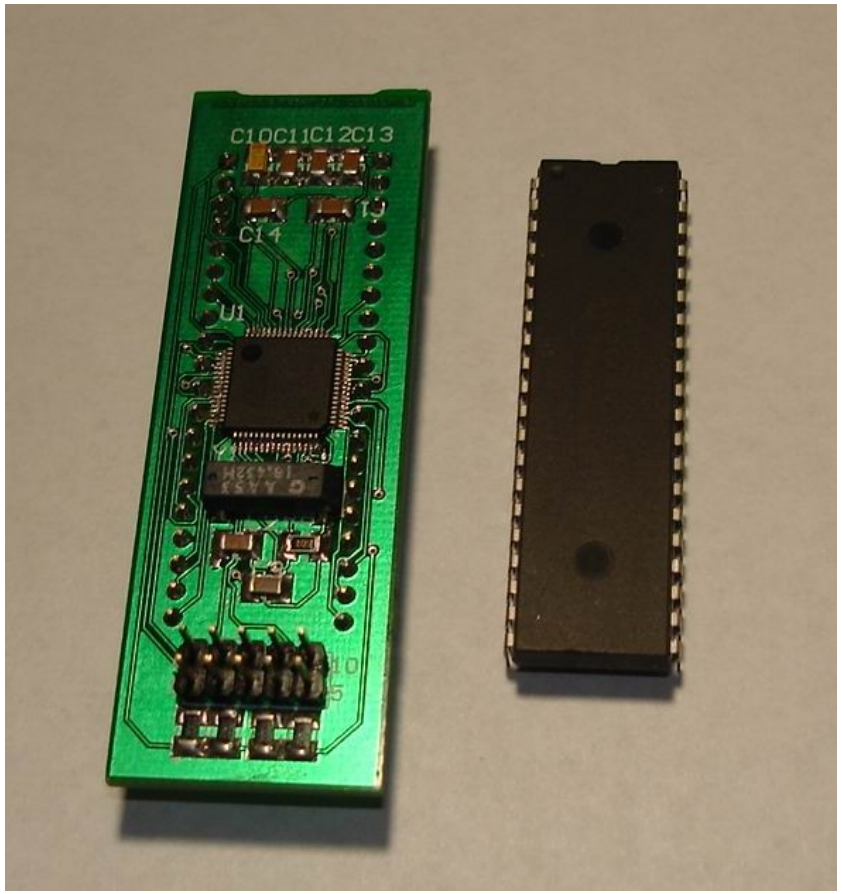
Bibliografía

- Using the FreeRTOS Real time Kernel, a Practical Guide - Richard Barry
- http://freertos.org
- Architecture of Device I/O Drivers" - David Kalinsky, Ph.D.
- Linux Device Drivers, Third Edition - Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
- An embedded Software Primer – David E. Simon
- http://elm-chan.org/fsw/ff/00index_e.html
- http://www.cs.brown.edu/courses/csci1600/handouts/notes/ipc.pdf
- http://sistemasembebidos.com.ar/foro

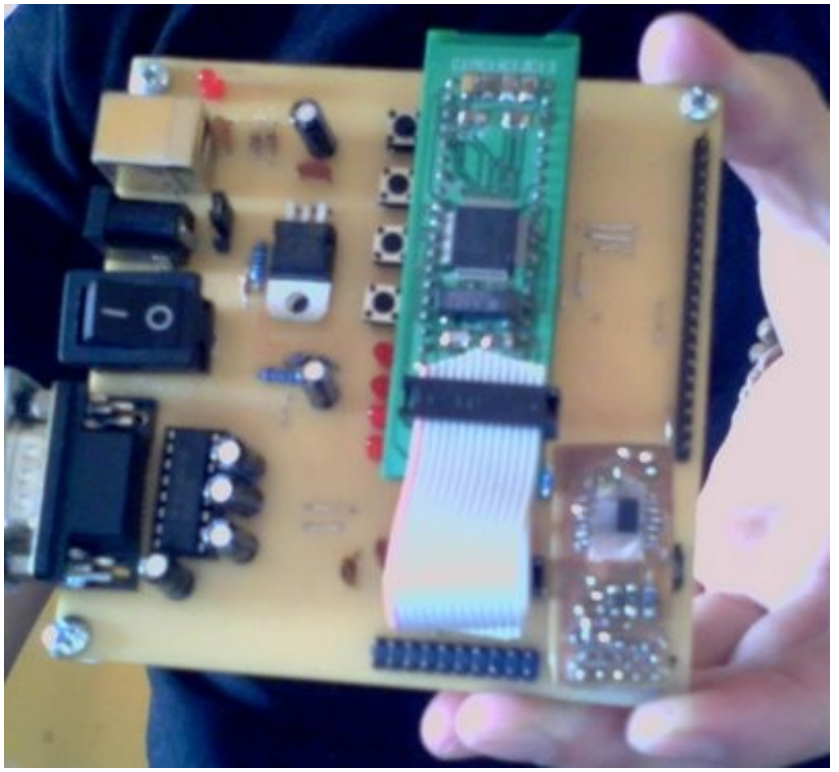
Preguntas

Gracias!

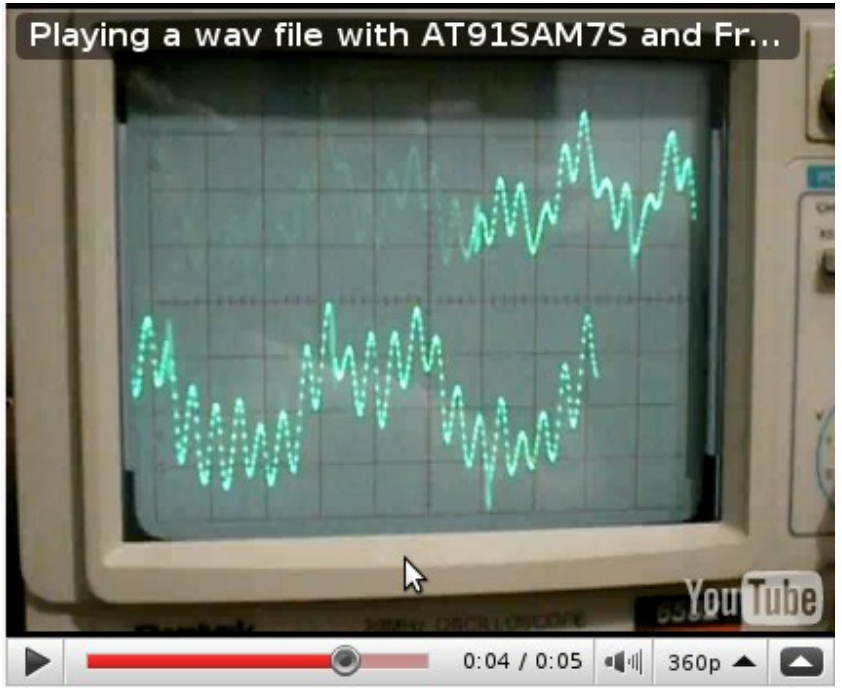
Backup



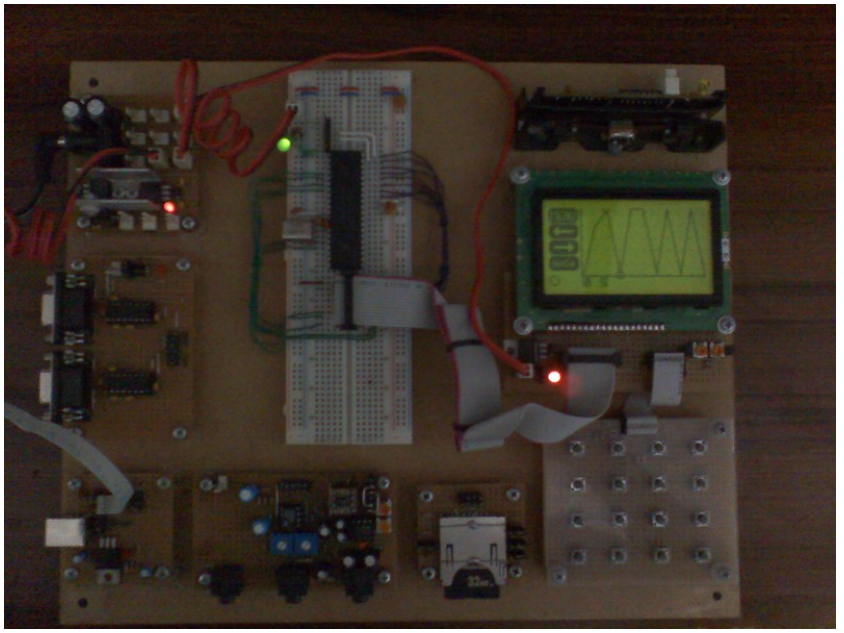
Placa ARM7TDMI adaptada de
LQFP64 a PDIP
(XTAL, USB y JTAG)



Ejemplo de uso para medicion
de acelerometros

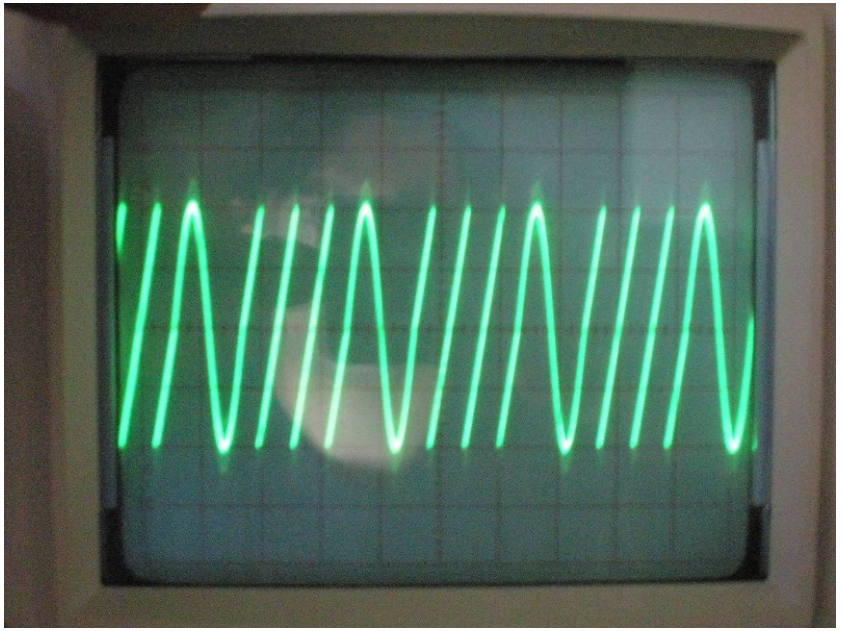


Demo AudioPlayer



Placa casera de desarrollo

Ejemplo Driver de video



Ejemplo Generador de ondas digital

Codigo fuente de los ejemplos