USO DE SISTEMAS OPERATIVOS EN TIEMPO REAL EN SISTEMAS EMPOTRADOS

Alexander Gómez Rojas

Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones
Universidad Industrial de Santander
Bucaramanga
2010

UNIVERSIDAD INDUSTRIAL DE SANTANDER
Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones
Perfecta combinación entre Energía e Intelecto

USO DE SISTEMAS OPERATIVOS EN TIEMPO REAL EN SISTEMAS EMPOTRADOS

Alexander Gómez Rojas

Trabajo de grado para optar por el título de Ingeniero Electrónico

Director
MSc. Jorge Hernando Ramón Suarez

Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones
Universidad Industrial de Santander
Bucaramanga
2010

"A Dios, mis padres, hermanos, amigos
por estar conmigo en cada paso
e iluminar mi mente y fortalecer
mi corazón."

AGRADECIMIENTOS

En primer lugar, a Dios, por llenarme de bendiciones y permitirme llegar hasta este momento que refleja el fruto de esfuerzo, constancia y perseverancia. Este es el primero de muchos exitos que vendrán.

A mis padres Hector Julio y Carmen Tulia, por sus concejos, por el apoyo incondicional y la motivación constante que me ha permitido ser la persona que hoy soy.

A mis hermanos, por su compañia en todos estos años y saber que cuento con ellos en todo momento y por compartir momentos inolvidables.

A mis amigos, por su apoyo incondicional, compañia, por los buenos momentos vividos y los recuerdos agradables también agradezco a Dios por haberlos conocido.

A mis profesores, por sus conocimientos, sabiduria, consejos y por ser una luz que guia.

A mi director, por la colaboración, apoyo, confianza brindados en el desarrollo del presente proyecto.

Contenido

Lista de Figuras

Lista de Tablas

Índice de códigos

RESUMEN

TITULO:
USO DE SISTEMAS OPERATIVOS EN TIEMPO REAL EN SISTEMAS EMPOTRADOS*

AUTOR:
ALEXANDER GÓMEZ ROJAS**

PALABRAS CLAVE:
Sistemas empotrados, sistemas operativos en tiempo real, tareas, colas, semáforos.

DESCRIPCIÓN:

En el presente proyecto se realiza una comparación entre diferentes sistemas operativos frente al método de interrupciones en un sistema empotrado. El sistema empotrado está conformado por un joystick, torre de desarrollo modular y un brazo robótico.
En los primeros capítulos se hace una revisión conceptual de los sistemas empotrados y los sitemas operativos en tiempo real con sus principales características. Se detallan las ventajas, desventajas y la solución que ofrecen los sistemas operativos en tiempo real en la administración de recursos y tiempo del procesador. Se realiza una breve descripción de cada uno de los sistemas operativos utilizados.
En los siguientes capítulos se presenta la implementación de cada uno de los elementos de hardware que conforman el sistema empotrado con las limitaciones físicas de cada componente, se expone la forma de diseño de la aplicación en base a las funciones disponibles en cada sistema operativo en tiempo real. Cada tarea ha sido diseñada cuidadosamente para evitar conflictos en el sistema respetando los tiempos de ejecución y así poder asegurar el correcto funcionamiento de la aplicación. Finalmente se realiza un cuadro comparativo entre los diferentes métodos de programación y se presentan las principales características de interrupciones y los sistemas operativos en tiempo real.

---

* Proyecto de Grado.

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones. Director MSc. Jorge Hernando Ramón Suarez.

SUMMARY

TITLE:
USE OF REAL-TIME OPERATING SYSTEMS IN EMBEDDED SYSTEMS*

AUTHOR:
ALEXANDER GÓMEZ ROJAS**

KEY WORDS:
Embedded systems, real-time operating systems, tasks, queues, semaphores.

DESCRIPTION:

In the present project is a comparison between different operating systems against interruption method an embedded system. The embedded system consists of a joystick, a tower of modular development and a robotic arm.
In the first chapter is a conceptual review of embedded systems and the operating systems in real time with their main characteristics. It details the advantages, disadvantages and the solution offered real-time operating systems in managing resources and processor time. A brief description of each of the operating systems used.
In the following chapter presents the implementation of each of the hardware elements that make up the embedded system with the physical limitations of each component, outlining how the application design based on the features available in each operating system on time real. Each task has been carefully designed to avoid conflicts in the system respect the execution times so we can ensure the proper functioning of the application.
Finally, make a table comparing the different methods of programming and presents the main features of disruption and real time operating systems.

---

* Degree project.

** Physics Mechanical Engineering Faculty. Electric, Electronic and Telecommunications School. Director MSc. Jorge Hernando Ramón Suarez

Introducción

El avance de la industria semiconductora ha hecho posible crear SoC (System on Chip), con millones de transistores en un área pequeña. Para utilizar esta capacidad los sistemas son construidos alrededor de un "core" (memoria-procesador-periféricos), adaptando los procesadores a una funcion especial y dotandolos de periféricos adecuados (rom, ram y software empotrado) para realizar la tarea específica [1].
Los sistemas operativos han venido ganando terreno debido a la importancia y complejidad de aplicaciones en tiempo real. La demanda de sistemas con plataformas en tiempo real se incrementa cada año motivando el desarrollo de aplicaciones [2].

Los sistemas de uso general no utilizan los recursos del sistema de forma adecuada teniendo un rendimiento variable con las tareas que realizan. Existen sistemas de función específica llamados *sistemas empotrados*, que poseen la cualidad de realizar una única tarea, por lo tanto son especializados formando parte de un sistema más grande.

Los sistemas empotrados pueden ejecutar gran cantidad de tareas, para ello necesita un planificador que distribuya los recursos de una manera eficiente. Un sistema operativo en tiempo real es un bloque básico de la mayoría de los sistemas empotrados que proveen esta característica, es decir, el sistema operativo en tiempo real asigna los recursos del sistema a una tarea específica por un determinado tiempo en base a prioridades.

La principal tarea de un sistema operativo en tiempo real es administrar los recursos del microcontrolador entre las diferentes tareas. Una tarea es generalmente un programa escrito en lenguaje C. Estos sistemas tienen la habilidad de usar los recursos necesarios para la ejecución de cada tarea, los más comunes son rom, ram y tiempo del procesador. Tienen una respuesta temporal predecible con unos estimulos del exterior impredecibles, para construir estos sistemas se necesita que todos los componentes (hardware y software) tengan este requisito en respuesta.

Los requisitos básicos de una aplicación en un sistema empotrado son:

- Poco espacio de memoria. Con un código pequeño y optimizado se resuelve este problema.
- Controladores de 8 bits. Para controladores de pequeña escala se crea un código minimalista que sea capaz de realizar las operaciones lógico-aritméticas sin gastar mucho tiempo del procesador. Para ello se tiene en cuenta las siguientes características:
    - Utilizar variables sin signo.
    - Evitar las divisiones.
    - Crear variables locales para evitar el uso escesivo de memoria.
    - Optimizar el código para evitar redundancias y condicionales innecesarios.

Aunque la meta del diseño de sistemas empotrados no es el alto rendimiento, una buena respuesta en el tiempo, un diseño con mínimo consumo de potencia y un rendimiento adecuado hacen que estos sistemas sean balanceados y sean usados en una amplia gama de aplicaciones.

Debido a la importancia y al interés de sistemas de tiempo real y a la demanda de arquitecturas que se incrementan cada año, se ha decidido evaluar un sistema operativo en tiempo real con una aplicación en un sistema empotrado.

OBJETIVOS

General

- Evaluar el uso de sistema operativo en tiempo real *(RTOS)* en sistemas empotrados.

Específicos

- Comparar 3 diferentes arquitecturas de sistemas operativos en tiempo real (MQX, FreeRtos y CircleOs) y escoger la más adecuada para el control de un brazo robotico mediante un joystick.
- Implementar un sistema operativo en tiempo real en el sistema empotrado y validar su funcionamiento con un cuadro comparativo.

ORGANIZACIÓN DEL DOCUMENTO

El documento se encuentra estructurado de la siguiente forma:

En el capítulo 1 se presentan los sistemas empotrados como una introducción a los sistemas operativos en tiempo real, debido a la importancia vez mayor estos sistemas en el desarrollo de aplicaciones con especifícaciones de tiempo real. Se hace una introducción a los sistemas operativos en tiempo real en cuanto a su clasificación y a los conceptos que se manejan como lo son las tarea, colas, semáforos.

En el capítulo 2 se exponen las arquitecturas seleccionadas describiendose detalladamente cada una de ellas. En cada arquitectura se destacan las características propias de cada sistema, las arquitecturas soportadas, la forma de administración de los recursos y la manera en que el planificador dirige los diversos mecanismos de control del sistema operativo.

En el tercer capitulo se explica el diseño e implementación del sistema operativo en tiempo real en el sistema empotrado. Se hace una descripción de cada elemento físico utilizado en el proyecto analizando sus ventajas y limitantes. Después se realiza un análisis del sistema empotrado y se procede a explicar la forma en que se programo el sistema operativo en tiempo real para implementarlo en el sistema empotrado.

El capítulo final se presenta la comparación de los parámetros importantes de cada sistema operativo y el método de interrupciones mediante un cuadro, como memoria, tiempos, lenguaje de programación.
Finalmente se describen las conclusiones, observaciones y recomendaciones para trabajos futuros. Se anexan los códigos de programación de cada sistema operativo.

Capítulo 1

Sistemas Operativos en Tiempo Real

1.1. Sistemas Empotrados

Actualmente no existe una única traducción para la palabra del inglés utilizada "*Embedded System*". Este término ha tenido varias traducciones dependiendo de las áreas de aplicación, los más empleados son "Sistemas Embebidos", "Sistemas Empotrados", "Sistemas Embarcados" y otras definiciones que últimamente están ganando terreno son "Sistemas Incorporados" y "Sistemas Integrados" [3].

Un sistema empotrado, embebido o integrado[1] es un sistema de uso específico, es decir, estan diseñados para realizar un rango específico de tareas y se encuentra construido dentro de un sistema más grande, normalmente en la placa base y dan una funcionalidad añadida a un equipo. Estos elementos "*son una parte diferenciable del objeto pero al mismo tiempo participa intrínsecamente en su construcción, su estructura y funcionalidad*", es decir, se puede identificar claramente en el sistema pero no se puede aislar completamente porque afectaría el funcionamiento total del sistema. Estos dispositivos se encuentran en nuestra vida cotidiana siendo algunos "invisibles". Gps, abs, radio FM, microondas, tv, telefónos, son tan solo algunos ejemplos.

La línea entre sistemas empotrados y los sistemas de propósito general se esta convirtiendo en borrosa a medida que avanza la tecnología, porque cada vez los sistemas empotrados son más robustos en el diseño, potentes en la capacidad de cómputo, flexibles y tienen mayor integrabilidad de periféricos lo que los hace aptos para ejecutar cualquier tarea.

[1] En adelante se referirá a estos sistemas como sistemas empotrados.

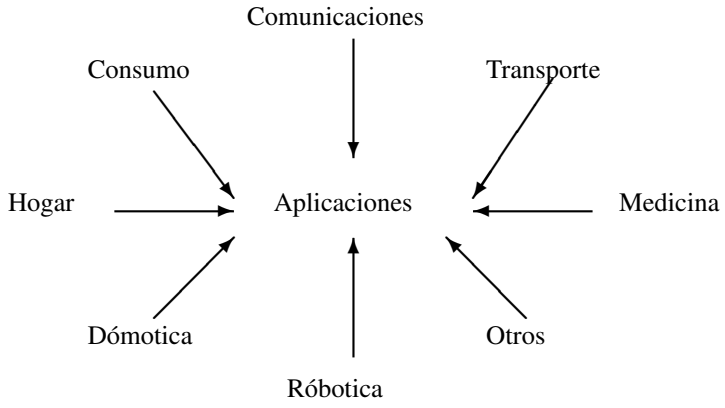


Figura 1.1: Flexibilidad de los sistemas empotrados en diferentes áreas.

Diversas áreas estan enfocadas hacia el desarrollo de aplicaciones en sistemas empotrados para realizar una tarea específica. Entre ellas se encuentran la róbotica, dómotica, hogar, medicina, consumo, transporte y comunicaciones. Fig 1.1.

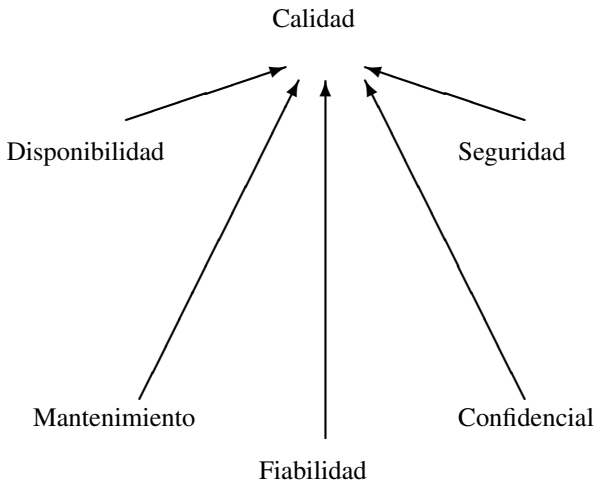


Figura 1.2: Exigencias en las aplicaciones para los sistemas empotrados.

Los sistemas empotrados se diseñan para ser robustos, confiables, seguros, bajo mantenimiento brindando calidad en las aplicaciones desarrolladas. Fig 1.2.

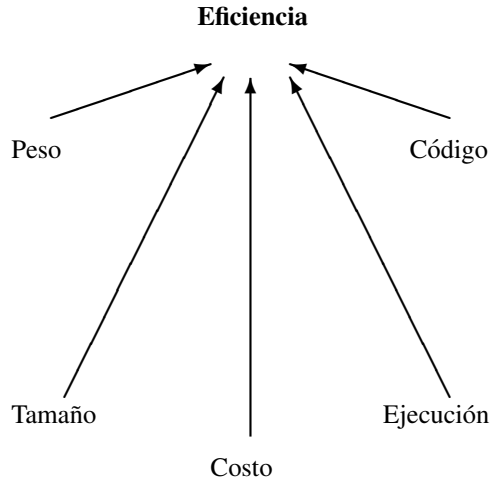


Figura 1.3: Eficiencia de la aplicación en un sistema empotrado

Un factor importante en los sistemas empotrados es la *eficiencia* que debe tener la aplicación en el manejo de recursos para utilizar el mínimo de energía, siendo de esta manera eficientes. La relación entre costo, código, peso, tamaño y ejecución en la eficiencia se puede ver en la fig 1.3.

Los sistemas empotrados se pueden clasificar de varias formas. Una de ellas es la interación con el entorno. Dentro de esta característica podemos encontrar:

- **Sistemas Reactivos**: Son aquellos que siempre interactúan con el exterior, de tal forma que la velocidad de operación del sistema dependerá de la velocidad del entorno exterior.
- **Sistemas Interactivos**: Son aquellos que interactúan con el exterior, de tal forma que la velocidad de operación del sistema dependerá de la velocidad del sistema empotrado.
- **Sistemas Transformacionales**: Son aquellos que no interactúan con el exterior, únicamente toman un bloque de datos de entrada y lo transforma en un bloque de datos de salida, que no es necesariamente del entorno.

1.2. Características de los SE

1.2.1. Procesador

El procesador en los sistemas empotrados no pose alta capacidad de procesamiento, en consecuencia tiene un algoritmo especializado. Si la cantidad de interrupciones es alta interfiere con el procesamiento de las tareas.

Los sistemas empotrados estan diseñados para realizar una tarea específica por eso se debe utilizar procesadores predecibles, es decir, que se puedan medir tiempos de respuesta para que su comportamiento sera el correcto.

1.2.2. Arquitectura

Los elementos básicos de un sistema empotrado son [4]:

Microprocesador: Se encarga de realizar las principales operaciones del sistema empotrado, ejecutando el código para realizar la tarea específica. También dirige la ejecución de los demás elementos.

Memoria: En este dispositivo se encuentra almacenado el código del programa y los datos. La velocidad de transferencia de datos es su principal característica ya que debe ser lo suficientemente rápida para que el microprocesador no pierda tiempo en operaciones de cálculo. Normalmente este tipo de dispositivos son volátiles, es decir pierden los datos almacenados cuando son desconectados de la alimentación.

Cache: Memoria de mayor velocidad que la principal donde se encuentra almacenado código y datos utilizados frecuentemente. Por su alto costo el tamaño de esta memoria es limitado en comparación con la principal.

Disco Flexible: En él se encuentra almacenada información y es de tipo no volátil, puede ser de material magnético lo cual lo hace inviable para entornos con vibraciones mecánicas. Hoy en día existen diferentes soluciones de estado sólido. Tienen una capacidad limitada.

Bios-Rom: BIOS (Basic Input Output System, Sistema Básico de Entrada y Salida), es el código necesario para inicializar el sistema y comunicar los diferentes elementos, el código de la BIOS se encuentra en la ROM.

Chipset: Chip encargado de controlar las interrupciones dirigidas al microprocesador, DMA (Acceso directo a memoria) y al bus ISA.

Entradas al sistema: Son dispositivos que dan la posibilidad al usuario de ingresar datos al sistema (mouse, teclado, puertos), para modificar las caractetísticas del sistema.

Salidas al sistema: Son dispositivos que permiten ver de una forma amigable para el usuario las diferentes opciones que ofrece el sistema, también sirve para observar el correcto funcionamiento del equipo.

1.2.3. Componentes

El microprocesador se encuentra como parte central del sistema, aportando la capacidad de cómputo. Puede tener pantalla gráfica, táctil, led, alfanumérica, etc. Posee puertos RS-32, RS-485, SPI, I^2C, CAN, USB, IP, Wi-Fi, GSM, GPRS, para la comunicación. Existen actuadores añadidos que el sistema se encarga de controlar, estos pueden ser motores eléctricos, relés, PWM (es el más habitual para el control de velocidad en motores de corriente continua).

El módulo E/S a veces es utilizado para digitalizar señales analógicas procedentes de sensores, activar leds, reconocer estados de conmutadores o pulsadores. Existe un único oscilador principal encargado de generar las señales de reloj. Dependiendo del tipo de oscilador se obtiene la frecuencia de operación necesaria para la estabilidad y el consumo de potencia requerida. Los osciladores de cuarzo son los más estables, mientras que los osciladores RC son los de menor consumo. Mediante sistemas PLL se obtienen otras frecuencias con la misma estabilidad que la del oscilador patrón. El módulo de energía se encarga de generará las diferentes tensiones y corrientes necesarias para la óptima operación de los sistemas.

1.2.4. Tiempo real

Tiempo Real es una característica importante en los sistemas empotrados. Para que el sistema sea en *tiempo real* es necesario que ejecute la interrupción antes de la llegada de la siguiente. El procesador sólo puede atender una sola interrupción, por lo tanto es necesario que los controladores se ejecuten lo más rápido posible [5].

Otra característica importante es la *latencia de interrupción* y comprende el tiempo transcurrido desde que se genera una interrupción hasta el momento en que es atendida, el inverso de este tiempo es la *frecuencia de saturación*, si la frecuencia de las señales son mayores a esta frecuencia el sistema será fisicamente incapaz de procesarlas.

1.3. Interrupciones

Las interrupciones son de naturaleza impredecible y es la manera más común de informar al programa sobre algún evento en el exterior como la presencia de datos en un puerto de entrada o un cambio en una variable específica, etc.

Existen varias técnicas de programación para ejecutar un conjunto de tareas en "paralelo[2]", las cuales son [6]:

- Procesamiento secuencial[3] (Bucle de scan).
- Primer plano / segundo plano (Foreground/Background).
- Multitarea cooperativa (No *Preemptiva*).
- Multitarea no cooperativa (*Preemptiva*).

Para evitar la latencia de las tareas en el procesamiento secuencial se puede asociar a interrupciones, dando lugar a un sistema como el mostrado en la figura 1.4.

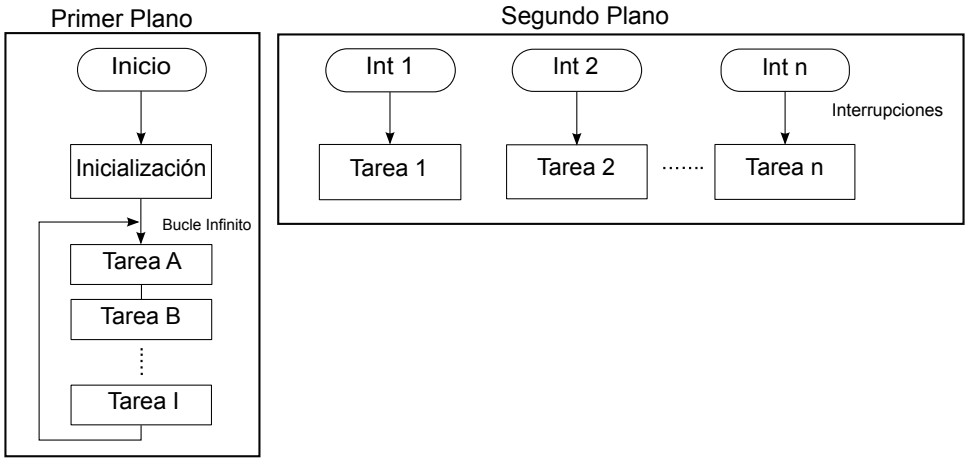


Figura 1.4: Sistema implementado con Foreground / Background [6]

A este tipo de sistemas se les denomina "Tareas en primer/segundo plano" (*Foreground / Background*). El nombre refleja precisamente el modo de implementar este tipo de sistemas. Existen 2 tipos de tareas:

- Un programa principal que se encarga de inicializar el sistema de interrupciones y luego permanece en un bucle infinito. Dentro de este bucle se ejecutan las tareas en primer plano. Estas tareas son TareaA, TareaB . . . TareaI.
- Tareas de segundo plano son las encargadas de gestionar algún suceso externo provocando una interrupción. En la figura 1.4 estas tareas son denominadas Tarea 1, Tarea 2 . . . Tarea n y las interrupciones son nombradas como Int 1, Int 2 . . . Int n.

La principal ventaja de este sistema es la baja latencia conseguida en los sucesos externos asociados a las interrupciones y la ejecución de las tareas que atienden a dichso sucesos (Tarea 1, Tarea 2 . . . Tarea n). Además no necesitan ningún software adicional.

Entre los inconvenientes de este tipo se sistemas se destacan:

[2] Las tareas se ejecutan secuencialmente pero lo hacen tan rápido que parecen que se lo hicierán en paralelo.
[3] La siguiente tarea no se ejecuta sin haberse ejecutado la tarea anterior.

- Se necesita soporte de interrupciones por parte del hardware.
- Esta estructura es válida si cada una de las tareas se puede asociar a una interrupción. Por ejemplo se tienen varias tareas que han de ejecutarse con distintos periodos y solo hay disponible un temporizador, no queda más que recurrir a un sistema operativo en tiempo real, o asociar varias tareas a una misma interrupción, lo cual complica la programación.
- Este sistema es más complejo de programar y de depurar que el procesamiento secuencial, no solo porque hay que gestionar el hardware de interrupciones, sino porque aparecen problemas ocasionados por la concurrencia de tareas y por la naturaleza de las interrupciones.

La latencia en este tipo de sistemas, en una primera aproximación, es igual al tiempo máximo de ejecución de las tareas de interrupción, ya que mientras se está ejecutando una tarea, las interrupciones están inhabilitadas. Por lo tanto, la duración de las tareas de segundo plano deben ser lo más pequeñas posibles.

```
void TareaTemp(void) //Tarea segundo plano
{  variables  ms, seg, min, hor;

   Actualizar variables();
   Imprimir(hor,min,seg,ms);
}

void main() //Tarea primer plano
{  for(;;)
   {

   }
}
```

Código 1.1: Implementación de un reloj.

El código 1.1 implementa la visualización de un reloj. Se ha supuesto que la TareaTemp esta asociada a una interrupción de un temporizador, el cual está configurado para que realize una interrupción cada milisegundo. Ahora se supone que la función *Imprimir*(hora, minutos, segundos, miliseg) tarda en ejecutarse 20 [ms]. Como por defecto existe una interrupción cada 1 [ms], la hora se actualizará cada 20 [ms] que es la duración de la función.

Por lo tanto, para que un sistema primer / segundo plano funcione correctamente, las tareas de segundo plano han de ser muy simples para que el tiempo de ejecución sea lo más corto posible. Para conseguirlo, es necesario dejar los procesos complejos para una tarea de primer plano.

```
variables  ms, seg, min, hor;

void TareaTemp(void) //Tarea segundo plano
{
   Actualizar variables();
}

void main() //Tarea primer plano

{
   for(;;)
```

```
    {
        Imprimir(hor, min, seg, ms);
    }
}
```

Código 1.2: Separación de tareas.

En el código 1.2 la tarea en segundo plano se limita a actualizar las variables y la tarea de primer plano se limita a imprimir la hora. Las variables hora, minutos, segundos y milisegundos ahora son globales porque se comparten entre estas 2 tareas.

Al trasladar los procesos lentos a la tarea de primer plano se soluciona el problema de la latencia, pero apararece un nuevo problema 'la comunicación entre tareas es asincronamente'. En el ejemplo anterior las dos tareas se comunican compartiendo tres variables globales.

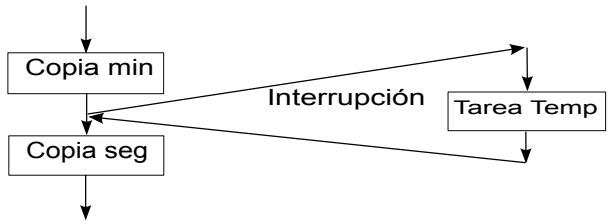


Figura 1.5: Comportamiento del sistema ante una interrupción [6]

El problema se genera cuando se está ejecutando la función *Imprimir* y se ha copiado la variable minuto de la hora actual, pero aún no se ha copiado la variable segundo como se muestra en la figura 1.5. Lo que se ve en la pantalla sería, suponiendo que la interrupción se ha producido en la llamada que imprime la segunda línea:

```
13:13:59
13:13:00
13:14:00
```

El problema que se acaba de presentar se origina por una interrupción dentro de lo que se denomina **zona crítica**. Por zona crítica se entiende toda zona de código en la que se usan recursos compartidos entre dos tareas que se ejecutan de forma asíncrona, como por ejemplo entre una tarea de primer / segundo plano.

Para evitar incoherencias de datos es necesario conseguir que la ejecución de la zona crítica se realize de principio a fin sin ningún tipo de interrupción. Por lo tanto una posible solución sería desabilitar las interrupciones durante la ejecución de la zona crítica.

```
...
    for(;;)
    {
        InhabilitarInterrupciones();
        Imprimir(hor, min, seg, milseg);
        HabilitarInterrupciones();
    }
...
```

Código 1.3: Solución de incoherencias entre datos.

Según lo expuesto si una tarea de segundo plano es compleja entonces es necesario dividirla, dejando la parte más costosa computacionalmente para una tarea asociada de primer plano. Tal como se ilustra en la figura 1.4 las tareas de primer plano se ejecutan secuencialmente, por lo que su duración dependerá de la ejecución de las funciones que se encuentren en el bucle.

1.4. Sistema Operativo

Un sistema operativo de tiempo real[4] (SORT o RTOS -Real Time Operating System-), es un sistema operativo que ha sido desarrollado para aplicaciones de tiempo real. Como tal, se le exige corrección en sus respuestas bajo ciertas restricciones de tiempo. Si no respeta las restricciones de tiempo se dirá que el sistema ha fallado. Para garantizar el comportamiento correcto en el tiempo requerido se necesita que el sistema sea predecible o determinista.

Los sistemas operativos en tiempo real son un mecanismo implementado en microcontroladores para dar flexibilidad a sistemas que tengan especificaciones fuertes en tiempo real ofreciendo un entorno estándar para que el software pueda interactuar con el hardware de manera uniforme. Las aplicaciones pueden ser ejecutadas en cualquier ordenador independiente de sus recursos. Estos sistemas son versátiles porque se adaptan a diferentes necesidades y exigencias. Se pueden reconfigurar para realizar diferentes tareas o cambiar totalmente la aplicación [5] [7].

Los sistemas operativos son clasificados en 3 categorias [8]:

- **Hard real-time**: Si excede el *deadline*[5] los resultados son catastróficos. Un ejemplo es la aviación.
- **Firm real-time**: Si excede el *deadline* provoca una reducción en la calidad del control del sistema con una consecuencia no grave. Un ejemplo son los controladores digitales PID.
- **Soft real-time**: Si excede el *deadline* se puede recuperar el control del sistema. La reducción en la calidad del sistema es aceptable. Aplicaciones de multimedia son ejemplos de esta catergoría.

Existen varios sistemas operativos los cuales proveen una solución para diferentes arquitecturas y herramientas de desarrollo, usan el mínimo de memoria, son sencillos y vienen preconfigurados. También proveen una amplia documentación y son una excelente opción para cumplir los requisitos de la mayoria de aplicaciones.

- eCos
- velOSity
- LynxOS
- embOS
- integrity
- OS-9
- RTEMS
- TRON

1.4.1. Características

Existen dos formas de diseño de los sistemas operativos:

- El sistema cambia cuando un evento necesita el servicio, estos sistemas están diseñados por eventos.
- El cambio de tareas se realiza por medio de interrupciones de un temporizador y eventos, estos sistemas están diseñados por compartición de tiempo. Se gasta más tiempo en cambios innecesarios, pero genera una mejor idea de multitarea.

En los sistemas operativos existen prioridades y pueden ser: Baja, Media y Alta, o numeradas -por ejemplo entre 0 y 62- donde 0 es la más baja prioridad y 62 es la más alta. Se suelen emplear algoritmos para la asignación de prioridades de las tareas. Las prioridades pueden ser fijas o variables y están definidas al principio de la ejecución del sistema.

[4] Sistema que interactua activamente con el medio con una dinámica conocida en relación con sus entradas, salidas y restricciones.
[5] Tiempo límite de ejecución de una tarea.

1.5. Características de los SO

1.5.1. *Preemptivo*

Un kernel *preemptivo* es usado cuando la sensiblidad del sistema es importante. La mayoría de los sistemas operativos en tiempo real son *preemptivos*. La tarea con mayor prioridad disponible se ejecutará en el procesador suspendiendo las tareas de menor prioridad.

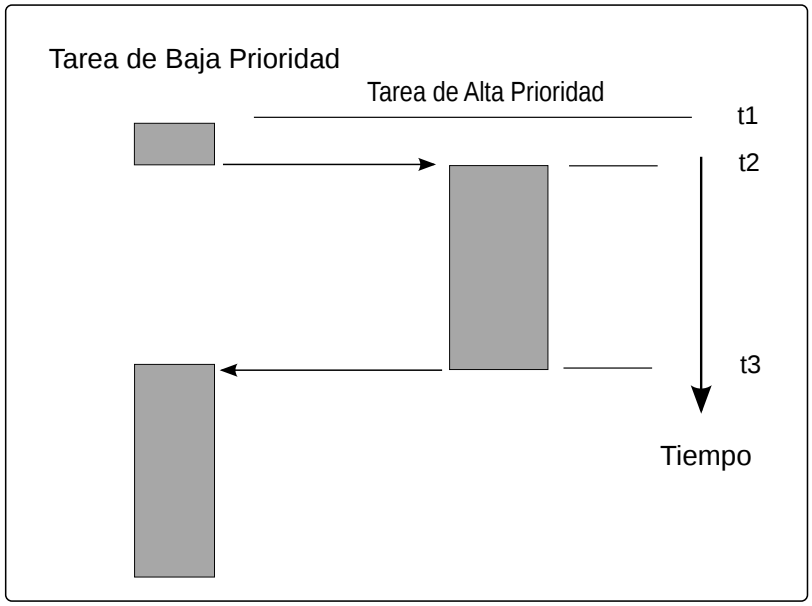


Figura 1.6: Kernel Preemptivo [5]

En la figura 1.6 se observa el comportamiento de un sistema con kernel *preemptivo*. En el tiempo t_1 la tarea de baja prioridad se encuentra en el estado ejecutándose en el procesador, en el tiempo t_2 aparece una tarea de mayor prioridad obligando un cambio de contexto. La tarea de baja prioridad se bloquea y el procesador ejecuta la tarea de alta prioridad. Cuando la tarea de alta prioridad termina de ejecutarse se realiza un cambio de contexto y permite seguir con la tarea de baja prioridad.

Con un kernel *preemptivo* la ejecución de la tarea de más alta prioridad es determinística y se puede conocer cuando obtiene el control del procesador.

1.5.2. No *Preemptivo*

Un kernel no *preemptivo* requiere que cada tarea termine la función antes de ceder el control del procesador. Una ventaja es que existe baja latencia de interrupción. El tiempo de respuesta puede ser más baja que los demás sistemas porque el tiempo de la tarea depende de la longitud de la tarea.

Otra de las ventajas de kernel no *premptivo* es la baja necesidad de acceder a recursos protegidos mediante semáforos porque cada tarea toma el control del procesador y no permite que otras tareas interfieran.

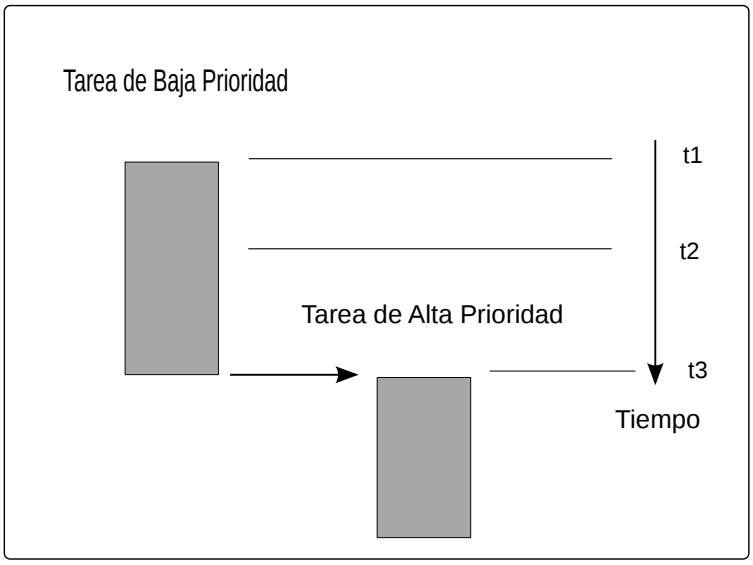


Figura 1.7: Kernel No Preemptivo [5]

En la figura 1.7 se observa el comportamiento de un sistema con kernel no *preemptivo*. En el tiempo t_1 la tarea de baja prioridad se esta ejecutándose en el procesador. En el tiempo t_2 aparece una tarea de mayor prioridad pero no se realiza ningún cambio de contexto hasta que la tarea de baja prioridad termine, esto ocurre hasta el tiempo t_3 donde ocurre un cambio de contexto y la tarea de alta prioridad se ejecuta en el procesador.

1.5.3. Co-rutinas

El uso de co-rutinas estan diseñadas para pequeños procesadores que tienen restricciones en el uso de memoria RAM. Solo permiten los siguientes estados:

- **Ejecutándose:** Cuando una co-rutina se esta ejecutando actualmente en el procesador.
- **Lista:** Cuando está disponible para ejecutarse (no esta bloqueada ni suspendida) pero no esta actualmente en el procesador y puede ser por:
  - Otra co-rutina tiene igual o mayor prioridad a la que se esta ejecutando actualmente.
  - Una tarea se esta ejecutando en el procesador.
- **Bloqueadas:** Se encuentra en este estado porque esta esperando un evento temporal o externo.

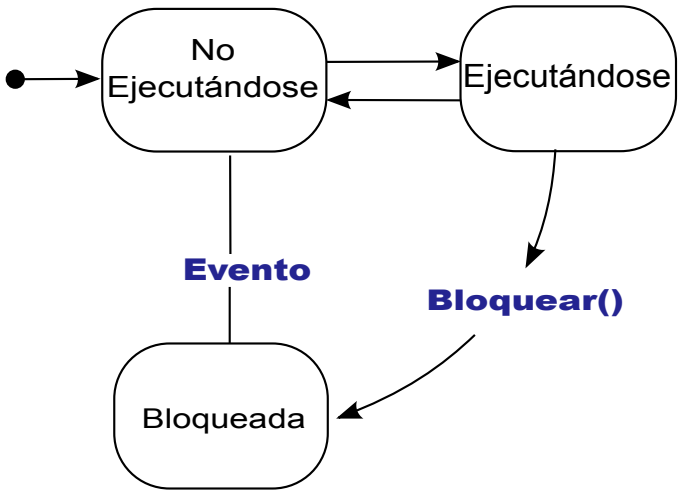


Figura 1.8: Estados válidos de las co-rutinas [7]

1.5.4. Tareas

Las tareas son pequeños programas implementados en lenguaje C con autonomía propia. Estan definidas en un lazo infinito.

Cualquier aplicación puede consistir en muchas tareas. Si en el sistema empotrado existe un procesador, entonces sólo será posible que una tarea se ejecute en cualquier instante de tiempo, esto implica que existen 2 estados primarios: *ejecutándose* y *no ejecutándose*. Fig 1.9

Cuando una tarea se encuentra en el estado *ejecutándose* su código esta actualmente en el procesador y cuando se encuentra en el estado *no ejecutándose* esta "dormida" y su estado ha sido guardado hasta que el programador decida pasarla al estado *ejecutándose*.

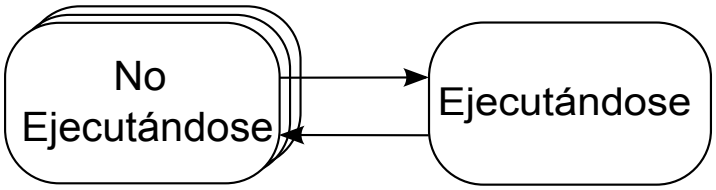


Figura 1.9: Estados primarios de una tarea [7]

```
void Tarea(Variables de entrada)
{
  Declaracion de variables
```

```
    for(;;)
     {
       Tarea();
     }

  La implementacion anterior nunca abandona del lazo
  infinito pero si por alguna circunstancia sale la
  tarea se debe destruir.

  Destruir(Tarea)
  }
```

Código 1.4: Estructura básica de una tarea.

La tarea que esta actualmente en el procesador se ejecutará por una fracción de tiempo (tick). La duración del tick es variable y depende de los requisitos de la aplicación. El scheduler[6] se ejecutará al final de cada tick y permitirá seleccionar la proxima tarea que pasará al estado activo. Si existe sólo una tarea y no se encuentra bloqueada ni suspendida, entonces será ejecutada en el procesador de forma continua. Para generar los ticks se implementará una interrupción periódica. Fig 1.10.

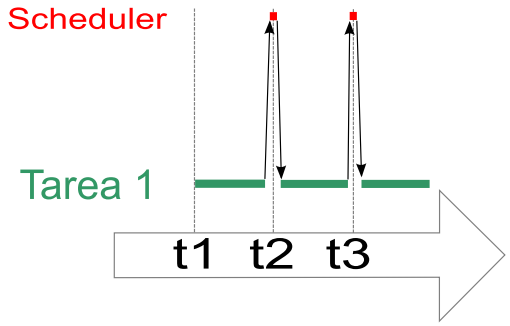


Figura 1.10: Ejecución del scheduler para elegir la próxima tarea a ejecutar [7]

Cuando se crean las tareas se asigna una prioridad que puede ser cambiada después de iniciar el scheduler. La transición entre un estado y otro se dice que la tarea ha cambiado de contexto. El máximo número de tareas depende de las restricciones de memoria y la manera de asignación de las prioridades del sistema operativo. El scheduler siempre ejecutará la tarea con la más alta prioridad disponible.

En la figura 1.11 se puede ver el comportamiento en el tiempo de 2 tareas con distintas prioridades. Las tareas se crean simultaneamente donde la tarea 1 tiene mayor prioridad que la tarea 2. La tarea 1 se encuentra en el estado *ejecutándose* y la tarea 2 en el estado *no ejecutándose* por las prioridades asignadas cuando se crearón y permanece en este estado durante en intervalo de tiempo t_1 y t_2. Al finalizar el tiempo t_2 la tarea 1 se bloquea permitiendo que el scheduler cambie de contexto. La tarea 1 estará en el estado *no ejecutándose* durante el intervalo de tiempo t_2 hasta t_3. Al finalizar el tiempo t_3 la tarea 1 es desbloqueada y al tener mayor prioridad pasa inmediatamente al estado *ejecutándose*.

Dos o más tareas pueden tener la misma prioridad, en este caso se alternarán entre el estado *ejecutándose* y *no ejecutándose*. Fig 1.12 (La operación del programador no ha sido graficada).

[6] Planificador

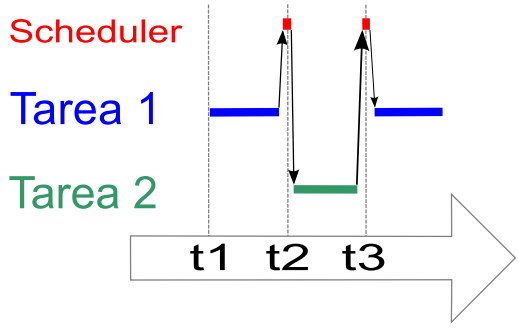


Figura 1.11: Secuencia de ejecución entre 2 tareas distintas prioridades [7]

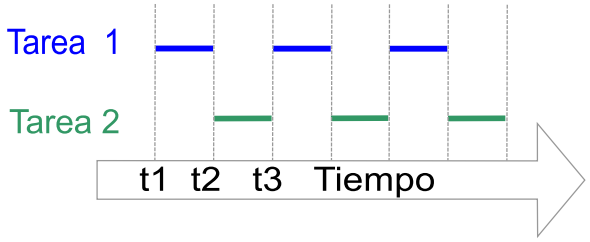


Figura 1.12: Ejecución de 2 tareas de igual prioridad [7]

Si una tarea tiene la máxima prioridad siempre estará en el estado *ejecutándose* y no permitirá que las tareas de baja prioridad pasen a este estado.

A veces se necesita que la tarea que se encuentra en el estado *ejecutándose* en el procesador deba esperar un evento externo o temporal, pasando al estado *no ejecutándose* y permitiendo que las tareas de baja prioridad cambien de contexto. Por lo tanto hay que definir subestados en el estado *no ejecutándose* y dependiendo la espera que tenga cada tarea se tienen los siguientes estados. Fig 1.13:

Estado bloqueado: Una tarea se encuentra en este estado si esta esperando un evento y no se encuentra ejecutándose. Las tareas pueden entrar en este estado de 2 formas:

1. **Evento Temporal** : El evento tiene un periodo de tiempo definido de expiración y al terminar cambia al estado listo para ejecutarse, por ejemplo la tarea espera en el estado bloqueado por 100 milisegundos.
2. **Evento de sincronización** : El evento es originado por otra tarea o una interrupción y no tienen un periodo definido, por ejemplo permanece en el estado bloqueado hasta que llegue un dato a una cola.

Es posible que una tarea bloqueada salga de este estado por ambos tipos de eventos, por ejemplo una tarea espera que pasen 100 milisegundos o la llegada de un dato a una cola. La tarea se desbloqueará si llega un dato antes de 100 milisegundos o 100 milisegundos después si no llega ningún dato.

Estado Suspendido: Indica que una tarea no está disponible para ejecutarse y entrará en este estado cuando explícitamente se llame con el comando correspondiente para suspender la tarea y volverá a estar disponible para el ejecutarse cuando se reanude explicitamente.

Estado listo: En este estado la tarea está esperando turno para ser ejecutada y no se encuentra en los subestados bloqueado o suspendido pero su prioridad es igual o más baja que la que está siendo ejecutada actualmente en el procesador.

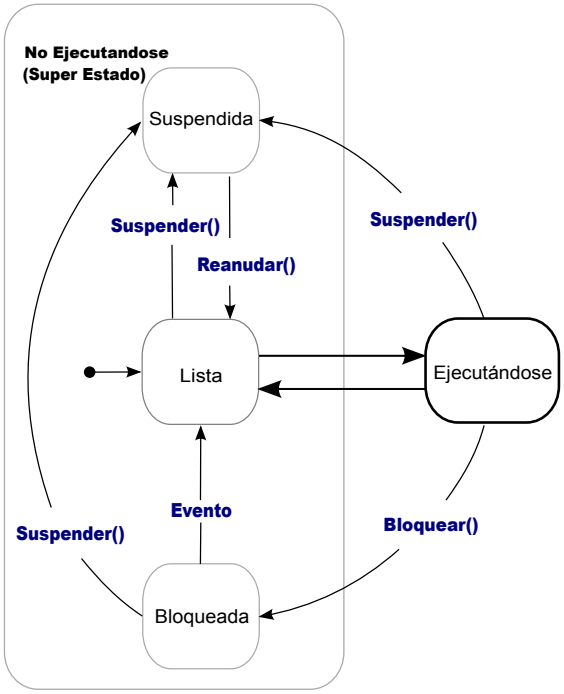


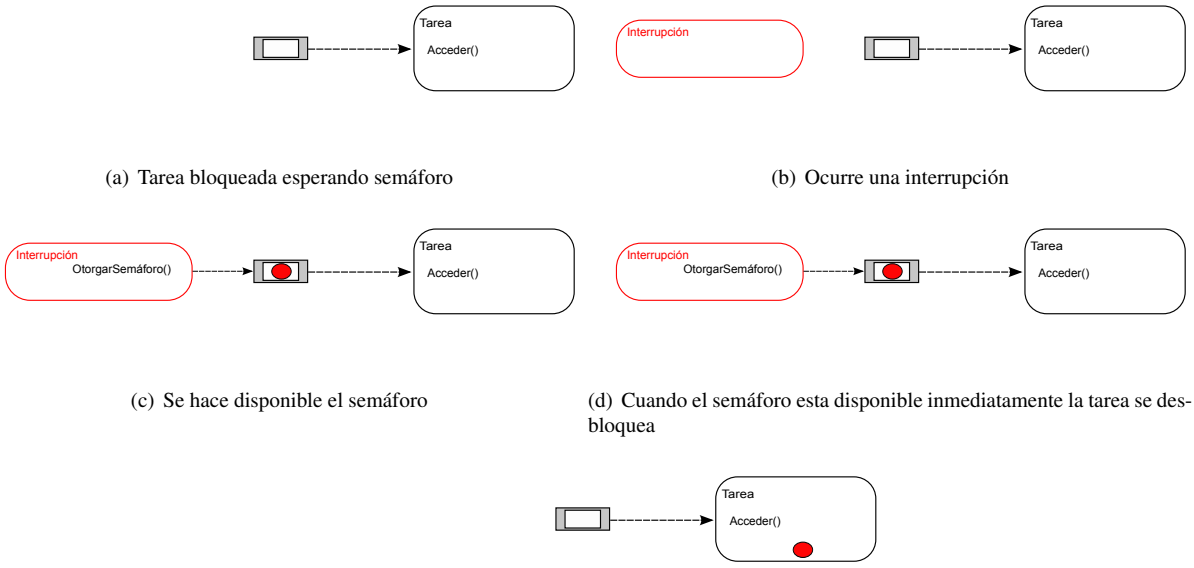
Figura 1.13: Estados completos [7]

Un punto crítico es el tiempo de respuesta que necesita poner en la cola una nueva tarea preparada y establecer el estado de la tarea de más alta prioridad.

1.5.5. Semáforos

Para que el sistema no ejecute accesos simultáneos a un recurso es necesario implementar un mecanismo de control. Los semáforos son usados para sincronizar varias tareas dentro del sistema. Cuando una tarea quiera utilizar un recurso protegido por un semáforo pregunta por el estado del recurso que va a utilizar, si está ocupado la tarea quedará en espera hasta que se libere, de lo contrario lo toma y bloquea el semáforo realizando su operación y una vez terminado libera el semáforo permitiendo que otras tareas puedan hacer uso del recurso. Fig 1.14.

Los semáforos permiten bloquear funciones por un tiempo específico. El tick indica el máximo número de fracciones de tiempo que una tarea estará bloqueada cuando pregunte por el semáforo y no se encuentre disponible. Si más de una tarea es bloqueada por el mismo semáforo entonces la tarea con la más alta prioridad será desbloqueada la próxima vez que este disponible. Se puede pensar que un semáforo es una cola con un solo ítem. Sin embargo la cola puede tener solo dos estados, estar llena o vacia. Tareas e interrupciones que usan una fila no les importa si esta bloqueada, solo quieren sabler si esta llena o vacia.



(a) Tarea bloqueada esperando semáforo

(b) Ocurre una interrupción

(c) Se hace disponible el semáforo

(d) Cuando el semáforo esta disponible inmediatamente la tarea se desbloquea

(e) Toma el semáforo y lo hace No disponible

Figura 1.14: Descripción del uso de un semáforo para sincronizar una tarea con una interrupción [7]

Los semáforos y los mutexes[7] son similares pero con algunas diferencias: los mutexes incluyen un mecanismo inherente de prioridad que los semáforos no poseen. Esto hace que los semáforos usen mejor la sincronización (entre tareas o tareas e interrupciones).

Deadlock o Abrazo Mortal

Un abrazo mortal es una situación en la cual dos tareas estan esperando los recursos protegidos de la otra. Por ejemplo, se ha supuesto que dos tareas acceden a dos variables compartidas y que cada variable esta protegida por un semáforo.

En el código no se ha mostrado el programa principal por simplicidad. Los argumentos de la función *AccederSem* son:

- Semáforo que protege un recurso.
- Tiempo de espera si el semáforo no esta disponible.

Con el argumento *Wait_Forever* la tarea se desbloqueará la próxima vez que el semáforo este disponible.

El problema de utilizar tiempo infinito puede ser muy grave si se comete el error mostrado en el código 1.5. Se supone que los semáforos estan libres y se ejecuta la Tarea1 la cual tomará el semáforo *Sem1* y tendrá acceso al recurso *V1*. Ahora se realiza un cambio de contexto y se ejecuta la Tarea2 accediendo al recurso protegido por el semáforo *Sem2* y seguidamente tratará de acceder al recurso protegido por el semáforo *Sem1* que no se encuentra disponible, por lo tanto

[7] Abreviatura de Mutual Exclusion (Exclusión mutua)

la Tarea2 se bloqueará indefinidamente permitiendo que se realize un cambio de contexto. La Tarea1 pasa a un estado activo y trata de obtener el segundo semáforo que no esta disponible porque la Tarea2 lo ha tamado. De esta manera ninguna tarea podrá ejecutarse de nuevo.

La forma de evitar que se produzca un abrazo mortal es pedir los semáforos en el mismo orden, no obstante en programas complejos es mejor usar semáforos sin tiempo infinito.

```
Variables V1, V2;

CrearSem  Sem1; //Se crea el semaforo 1
CrearSem  Sem2; //Se crea el semaforo 2

void Tarea1(void)
{  //Accede al sem1 y si no esta disponible la
   //tarea se bloquea hasta que se encuentre disponible

   AccederSem(Sem1,Wait_Forever);
   AccederSem(Sem2,Wait_Forever);

   V2=V1+2;//Se realiza la operacion de ejemplo

   CederSem(Sem1);
   CederSem(Sem2);
}

void Tarea2(void)
{
   //Accede al sem2 y si no esta disponible la
   tarea se bloquea hasta que se encuentre disponible

   AccederSem(Sem2,Wait_Forever);
   AccederSem(Sem1,Wait_Forever);

   V1=V2+2;//Se realiza una operacion de ejemplo

   CederSem(Sem1);
   CederSem(Sem2);
}
```

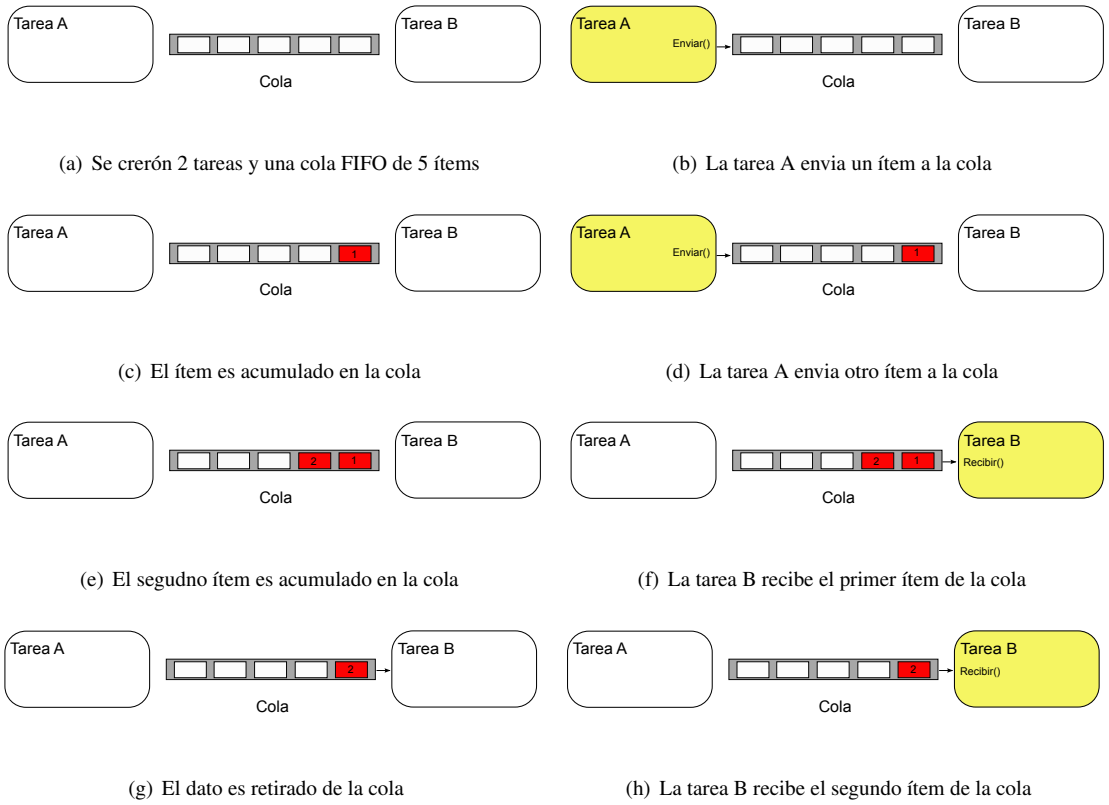
Código 1.5: Ejemplo de un abrazo mortal.

1.5.6. Colas

Las colas son una forma primaria de comunicación entre tareas. Pueden ser usadas para enviar mensajes entre tareas o interrupciones y tareas. En la mayoría de los casos son usados las colas FIFO[8], sin embargo los datos pueden enviarse al inicio o al final de la cola.

Las colas pueden contener 'ítems' de tamaño y número fijo, las cuales son definidas en el momento de crearlos. Los 'items' son colocados en la cola como una copia y no por referencia. En la figura 1.15 se puede ver el comportamiento de una cola.

[8]First In Fist Out, Primero en Entrar Primero en Salir



(a) Se crerón 2 tareas y una cola FIFO de 5 ítems

(b) La tarea A envia un ítem a la cola

(c) El ítem es acumulado en la cola

(d) La tarea A envia otro ítem a la cola

(e) El segudno ítem es acumulado en la cola

(f) La tarea B recibe el primer ítem de la cola

(g) El dato es retirado de la cola

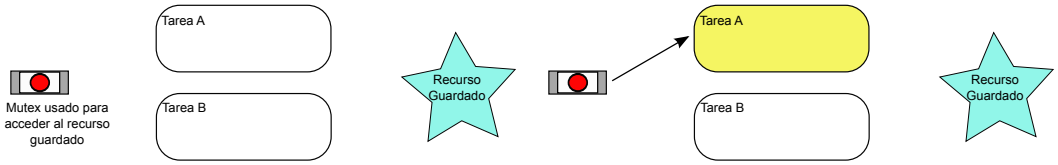
(h) La tarea B recibe el segundo ítem de la cola

Figura 1.15: Descripción del comportamiento de una cola entre dos tareas [7].

1.5.7. Exclusiones Mutuas

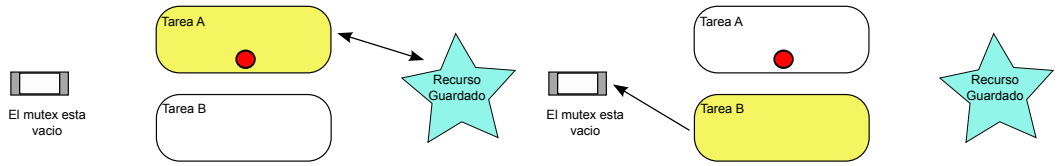
Las exclusiones mutuas son semáforos que incluyen un mecanismo inherente de prioridad. Cuando se usa una exclusión mutua, esta actua tomando el recurso protegido. Cuando una tarea desea acceder al recurso primero se debe 'obtenerlo'. Cuando ha finalizado de utilizar el recurso este se debe 'cederlo'.

Las exclusiones mutuas usan el mismo acceso que los semáforos que permiten bloqueo por un tiempo específico de las tareas. El bloque de tiempo indica el número máximo de fracciones de tiempo que una tarea debería entrar en el estado *bloqueado* cuando hace un intento de 'tomar' el recurso y no se encuentra inmediatamente disponible. Las exclusiones mutuas poseen prioridad, esto significa que si una tarea de alta prioridad intenta obtener un recurso guardado mientras está siendo usado por una tarea de baja prioridad, entonces la prioridad de la tarea es mantenida temporalmente bloqueando la tarea de baja prioridad.
Este mecanismo esta diseñado para permitir que una tarea de alta prioridad este en el estado bloqueado el menor tiempo posible, y también minimizar la 'inversión de prioridad' que ha ocurrido.



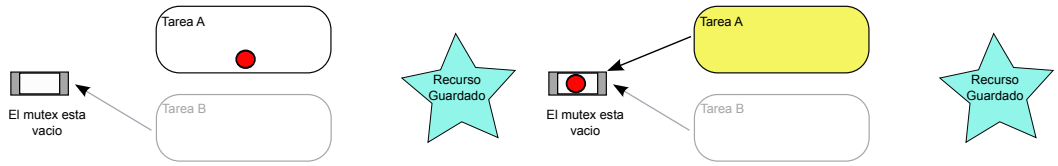
(a) Para acceder al recurso primero debe "tomarlo"

(b) La tarea A toma el acceso al recurso



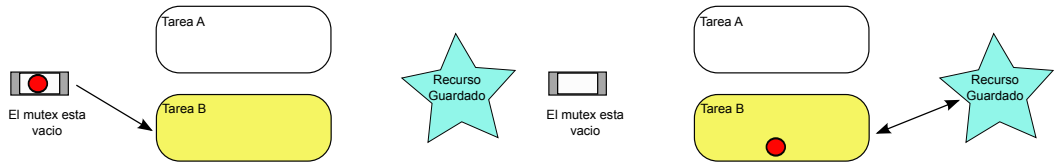
(c) El semáforo es exitosamente obtenido. Ahora la tarea A puede acceder al recurso guardado

(d) La tarea B intenta acceder al recurso, pero primero debe obtenerlo



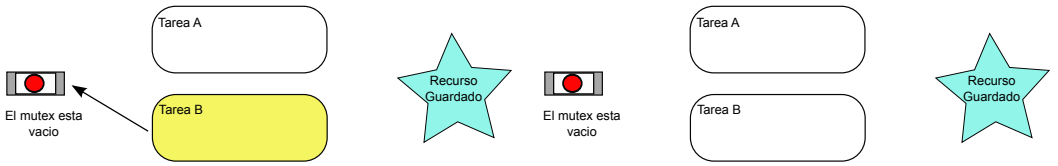
(e) El semáforo no esta disponible, entonces la tarea B se bloquea a la espera del semáforo

(f) Cuando la tarea A ha finalizado con el recurso guardado retorna el semaforo



(g) Y desbloquea la tarea B

(h) La tarea B ahora puede exitosamente obtner el semáforo y acceder al recurso guardado



(i) Después de terminar cede el semáforo

(j) El semáforo queda disponible para otras tareas

Figura 1.16: Descripción el comportamiento de las exclusiones mutuas [7]

Capítulo 2

Arquitecturas

La mayoría de los sistemas operativos estan escritos en lenguaje C, siendo un lenguaje de programación fácil de manejar y ampliamente conocido. Tiene un scheduler que administra las tareas, ya sea que esten listas para ejecutarse, bloqueadas o suspendidas.

Los sistemas operativos en tiempo real seleccionados fuerón [9] [8]:

- **FreeRTOS**: Sistema operativo de código abierto, escrito en C, de fácil programación.
- **MQX**: Sistema operativo propio de Freescale, presenta varias funciones en el momento de la programación, como logs, kernel, colas, semaforos, mutex, watchdog.
- **uC/OS II**: Sistema operativo portable, escalable y de tiempo real, escrito en C y adaptado a diferentes arquitecturas

2.1. MQX

MQX es un sistema operativo en tiempo real desarrollado por Precise Software Technologies Inc[1] y vendido a ARC International, Embedded Access[2] Inc y Freescale[3].

MQX ha sido diseñado para uni-procesador, multi- procesador y procesadores distribuidos. Freescale adopto esta plataforma de software para sus procesadores ColdFire y PowerPC. En comparación con la distribución original de MQX, la distribución MQX de freescale ha sido simplificada para uso de aplicaciones.

Hasta el momento el entorno de desarrollo por defecto para freescale MQX RTOS y otros componentes de MQX es CodeWarrior Development Studio. Nuevas versiones podrán ser soportadas por otros entornos de desarrollo. Con Codewarrior se tiene una vista trasparente a las tareas e información del sistema operativo, esto incluye lista tareas, colas, semáforos, mutexes, eventos, mensajes entre colas, mensajes por encuesta, bloques de memoria, estructura de datos y kernel, estructura de inicialización y manejo de interrupciones.

MQX es una arquitectura basada en componentes con funciones extenedidas, incluye 25 componentes - 8 del nucleo y 17 opcionales-. Fig 2.1

[1] www.pstinc.com/
[2] www.embedded-access.com
[3] www.freescale.com

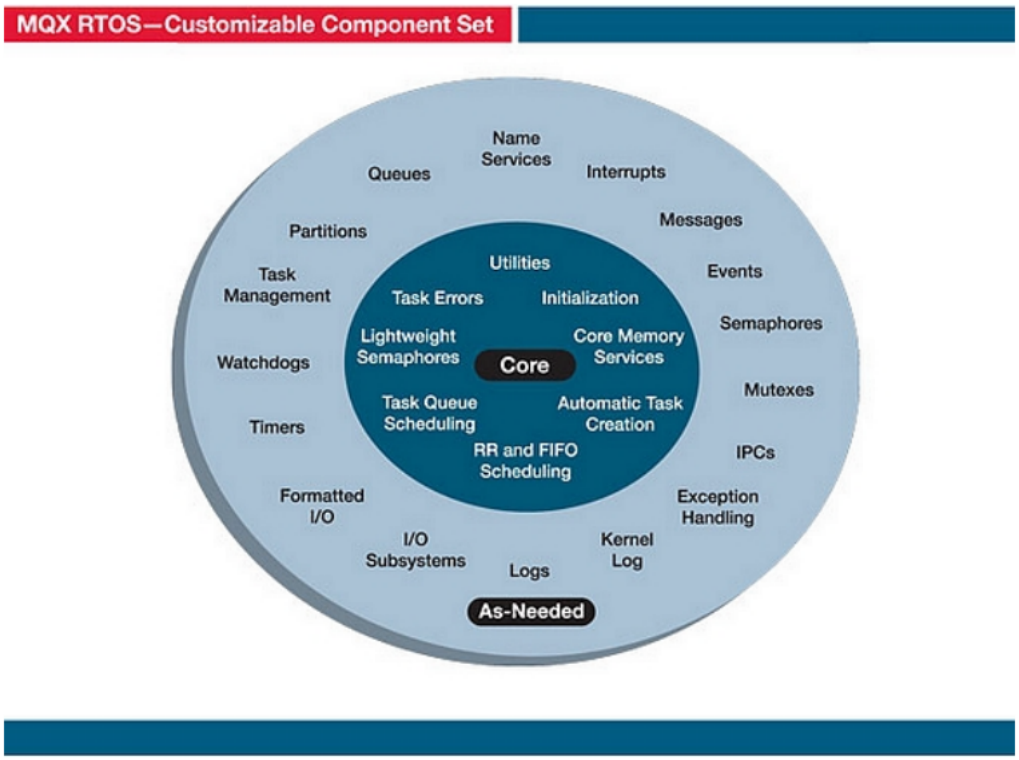


Figura 2.1: Componentes de MQX [10]

MQX otorga una misión critica en áreas como la aviación, aeroespacial, médica y comunicación. Es implementado como una libreria C con un código optimizado en asembler para el cambio de contexto y el manejo de interrupciones.

Posee las siguientes características:

Peqeña densidad de código: Ha sido diseñado para un eficiente manejo de velocidad y tamaño en sistemas empotrados. Se obtiene un verdadero rendimiento en tiempo real con cambios de contexto e interrupciones de bajo nivel.

Arquitectura basada en componentes: Provee un nucleo con una completa funcionalidad con servicos adicionales.

Componentes completos y ligeros: Los componentes claves estan incluidos en 2 versiones, la completa y la ligera, para un control del tamaño.

Tiempo real, basado en prioridades: Las tareas se ejecutan en orden de su prioridad, si una tarea de alta prioridad se convierte en lista para ejecutarse, se puede con un pequeño y corto intervalo de tiempo tomar el control del procesador sobre cualquier tarea de baja prioridad que se este ejecutando. siendo esta tarea ininterrupida mientras permanezca con la más alta prioridad hasta que termine.

Scheduler: MQX provee de un desarrollador rápido para crear o mantener un sistema progamado eficiente y el manejo de interrupciones.

Se tienen los siguientes opciones en MQX

2.1.1. Programador

El programador compila con POSIX.4 (extenciones de tiempo real) y soporta:

- FIFO.
- Round robin.
- Programador explícito.

2.1.2. Manejo de memoria

Para asignar o liberar bloques de memoria de tamaño variable, *MQX* provee un servicio en el nucleo similar a **malloc()** y a **free()** el cual la mayoría de las librerias C proveen.

- Manejo de memoria con bloques de tamaño fijo: La partición es un componente rápido, localización de memoria determinístico, el cual reduce la fragmentación de la memoria y conserva el recurso.
- Control de cache: MQX incluye la función de controlar la instrucción de cache que algunas CPU tienen.
- Controlando un MMU : Para algunas CPU, se debe inicializar la unidad de manejo de memoria (Memory Managment Unit) antes de permitir el cache. MQX permite inicializar, habilitar o desabilitar un MMU y añadir memoria a una región.
- Memoria de poco peso: Si una aplicación tiene requerimientos de tamaño de código y datos restringidos, entonces memoria de poco peso debe ser usada, es una interface de funciones para pequeños tamaño de código y datos, como resultado se pierde robustez y es más lenta.

2.1.3. Sincronización de Tareas

Entre las tareas de sincronización se tiene los siguientes elementos:

Eventos: Soporta manejo dinámico de objetos. La rutina de servicio de interrupción puede usar eventos para sincronizar de una manera simple la información.

Semáforos ligeros: Se pueden usar los semáforos para sincronizar tareas. Se puede usar un semáforo para guardar un recurso o para implementar un mecanismo de señalamiento entre tareas. No poseen prioridad inherente, y pueden ser estrictos o no estrictos. También son nombrados como semáforos rápidos.

Exclusiones mutuas: Proveen una exclución mutua a las tareas cuando intentan acceder a un recurso protegido. Se pueden acceder por medio de encuesta, FIFO, proteccion de prioridad, prioridad inherente.

Mensajes: Las tareas se pueden comunicar con otras mediante el envío de mensajes a una cola de mensajes que es abierta por otras tareas. Cada tarea abre su propia cola de mensajes. Un mensaje tiene identificación única el cual MQX asigna cuando la fila es creada. Solo la tarea que fue creada los mensajes puede abrirlos pero cualquier tarea puede enviar mensajes a cualquier cola de mensajes si conoce la ID.

Interrupciones y manejo de excepciones: MQX soporta completamente las interrupciones. La rutina de servicio a interrupciones (ISR) es una aplicación que puede permitir cualquier nivel de interrupción. ISR no es una tarea, es una rutina de alta velocidad y pequeña que reacciona rápidamente a interrupciones de hardware. Esta escrita en lenguaje C.

2.2. FreeRTOS

FreeRTOS^{TM} es un software portable, de código abierto, realmente libre, con mini-kernel de tiempo real. La descarga es gratis y libre para usar el sistema operativo que puede ser usado sin ningún requerimiento de exponer el código fuente del propietario en aplicaciones comerciales [11].

FreeRTOS permite configurar el kernel de manera cooperativa, preemtiva o híbrida, con un alto nivel de confianza e integratibilidad del código soportando 23 arquitecturas. Esta diseñado para ser simple, pequeño y facil de usar. Tiene una estructura predominante escrita en lenguaje C, soportando tareas y co-rutinas. No tiene restricción para el número de tareas creada y el número de prioridades que puedan ser usadas.

Colas, semáforos contadores y recursivos, mutexes son componentes usados para la sincronización y comunicación entre tareas o tareas e interrupciones. *FreeRTOS* se puede ejecutar en herramientas de desarrollo libre para todas las arquitecturas con el código fuente empotrado libre para modificar.

Cada arquitectura incluye una aplicación pre-configurada de las características del kernel, para acelerar el aprendizaje y permitir el desarrollo de aplicaciones avanzadas. Existen dos clases de sistemas operativos:

OpenRTOS^{TM}**:** Es una licencia comercial con una versión soportada de FreeRTOS que incluye todas las funciones profesionales con componentes USB, TCP/IP y archivos del sistema.

SafeRTOS^{TM}**:** Es la versión que ha sido certificada para uso seguro en aplicaciones críticas. Es un producto compatible con **IEC 61508**[4]

	FreeRTOS	OpenRTOS
Es libre?	Si	No
Puede usarse comercialmente?	Si	Si
Las ganancias son libres?	Si	Si
Tiene el código abierto para cambiar el kernel?	Si. Web	No
Puedo ofrecer el código a los usuarios de mi aplicación?	Si	No
Puedo proveer garantia del código?	No	Si
Tiene protección legal?	No	Si

Tabla 2.1: Características de FreeRTOS

2.2.1. Descripción

FreeRTOS tiene las siguientes características:

- Provee una solución para diferentes arquitecturas y herramientas de desarrollo.
- Es fiable. Tiene confianza garantizada por las actividades realizadas por SafeRTOS
- Esta bajo continuo desarrollo.
- Tiene un mínimo de ROM y RAM. Tipicamente ocupa una región entre 4k y 9k bytes.
- Es muy simple, consta de 3 archivos escritos en lenguaje C.
- Es realmente libre para uso en aplicaciones comerciales.

[4]Es un estándar internacional de reglas aplicadas a la industria y su intención es ser una base fundamental aplicable en la seguridad de la industria. `http://www.iec.ch/functionalsafety/`

- Es portable, con plataforma de desarrollo.
- Es muy estable con una gran base de usuarios.
- Contiene un ejemplo pre-configurado para cada puerto. No necesita modificar el proyecto, solo se necesita descargar y compilar.
- Tiene una amplia documentación.
- Es escalable y simple para usar.

2.2.2. Arquitecturas Soportadas

Altera, Atmel, Freescale, Fujitsu, Luminary Micro / Texas Instrument, Microchip, Nec, Nxp, Renesas, Silicon Labs, St, Ti, Xilinx, X86 son algunas arquitecturas soportadas por FreeRTOS. Al ser compatible con Freescale es posible utilizar el software CodeWarrior para el desarrollo del diseño con todo el potencial que posee para los microcontroladores de la familia HCS12, ColdFire V1, ColdFire V2.

2.3. uC/OS II

uC/OS II es un software completamente portable, ROMable, escalable, tiempo real, preemptivo, escrito en C y contiene una pequeña porción de lenguaje asembler adaptado a diferentes arquitecturas de procesadores [12]. Miles de personas alrededor del mundo usan uC/OS II en sus aplicaciones como:

- Aviónica.
- Equipo/dispositivos Médicos.
- Equipo de Comunicación.
- Teléfonos, PDAs.
- Industria de Control.
- Automovilismo.
- Un amplio rango de aplicaciones en sistemas empotrados.

2.3.1. Características

Código Fuente: El código fuente está ordenado de una forma consistente, clara, documentada, y organizada. Conocer el código fuente no es suficiente, se necesita conocer como trabajan las diferentes piezas.

Portable: uC/OS II esta escrito en lenguaje C con código en lenguje asembler para tarjetas con microprocesadores específicos.

ROMable: uC/OS II esta diseñado para aplicaciones empotradas por lo tanto se puede incluir a uC/OS II como parte del producto con las herramientas adecuadas (Compilador C, Linker, Asembler).

Escalable: uC/OS II esta diseñado para que se pueda usar solo los servicios necesarios en la aplicación, lo cual significa que el producto final puede usar unos pocos servicios del sistema operativo reduciendo el tamaño de memoria necesaria (RAM, ROM).

Premptivo: uC/OS II tiene un kernel totalmente premptivo que siempre ejecuta la tarea con más alta prioridad que no se encuentre en el estado *no ejecutándose*.

Multitarea: Puede manejar hasta 64 tareas, sin embargo, existen 8 tareas reservadas al sistema operativo, teniendo hasta 56 tareas. Cada tarea tiene prioridad única por lo tanto existen 64 prioridades.

Determinístico: El tiempo de ejecución de las funciones y los servicios del sistema operativo es determinístico, se puede conocer cuanto tiempo tarda en realizar una función o servicio.

Servicios: uC/OS II provee varios servicios como semáforos, exclusiones mutuas, eventos, mensajes, colas, particiones de memoria de tamaño fijo, administración de tareas, funciones de manejo de tiempo.

Provee toda la documentación necesaria para el soporte y el uso de uC/OS II en sistemas críticamente seguros. Actualmente está implementado en en alto nivel para:

- Aquellos dispositivos certificados para aviónica DO-178B. [5][6]
- Dispositivos médicos.
- Sistemas nucleares y de transporte SIL3/SIL4 IEC.

uC/OS II combina una curva de aprendizaje extremadamente corta con un uso fácil.

2.3.2. uC/OS II

uC/OS II	
Quien debería usar RTOS?	Desarrolladores que quieran mejorar el tiempo en los proyectos con sistemas empotrados, y los que quieran un código más limpio, robusto.
Procesadores Soportados	Ver cuadro 2.3
Máximo ROM (No escalable)	24 Kbytes
Minima ROM (Escalable)	6 Kbytes
Número de Servicios del Kernel	10 diferentes usando 80 API
Modelo de multitarea	Premptivo
Ejecución del código	Tareas, ISR
Objetos Dinámicos	Estaticos y dinamicos
Movimiento de datos	Mensajes (Ilimitado) Colas (Ilimitado)
Semáforos	Si (Ilimitados)
Exclusiones Mutuas- Con prioridad inherente	Si (Llamando prioridad)
Partición de Memoria	Si
Administración de tiempo	Si (Ilimitados)

Tabla 2.2: Características de uC/OS II [12]

[5] Estándar internacional de avionica para software criticamente seguro.

[6] www.ldra.com/do178b.asp?gclid=CPuMz_qSlqQCFVkJ2godO0Zu4g

2.3.3. Procesadores Soportados

Compañia	Arquitectura
Actel	Cortex-M1
Aletra	Nios II, Cortex-M1
Analog Devices	AduC7xxx (ARM7), ADSP-21xx, Blackfin 5xx, SHARC
ARM	ARM7, ARM9, ARM11, Cortex-M1, Cortex-M3
Atmel	SAM7 (ARM7), SAM9 (ARM9), AVR, AVR 32
Freescale	9S08, 9S12, Coldfire, PowerPC, I.MX
Fujitsu	FR50
Infineon	TriCore, 80C16x
Intel	80x86
Lattice	Micro32
Luminary Micro	Cortex-M2
Microchip	PIC24, dsPIC33, PIC31 (MIPS)
MIPS	R3000,R4000
NEC	78Kx, V850
NXP	ARM7, ARM9, Cortex-M3
Remesas	H8, M16C, M32C, R32C, SH
Samsung	ARM7, ARM9
ST	80C16x, STR7 (ARM7), STR9 (ARM9), STM32 (Cortex-M3)
TI	MSP430, TMS320, TMS470 (ARM7)
Toshiba	Cortex-M3
Xilinx	MicroBlaze, PowerPC
ZILOG	Z80, eZ80

Tabla 2.3: Procesadores Soportados por uC/OS II [12]

Capítulo 3

Diseño e Implementación

Dos aspectos importantes que se deben tener en cuenta al realizar un proyecto son el Hardware y el Software. El Hardware aporta los elementos físicos que se pueden tocar, aunque algunos no se pueden ver porque estan de un circuito integrado o un microcontrolador. La mayoría del hardware permite interactuar con el sistema mediante dispositivos de (salida, entrada, almacenamiento, procesamiento). El Software aporta la forma de utilizar los recursos lógicos para la creación del proyecto. Estos aspectos son compelementarios en el momento de analizar la manera en que se implementará cada uno de los recursos disponibles.

En el desarrollo del presente proyecto se utilizó un brazo robótico, un joystick y la torre Mcf51cn128 para crear un sistema empotrado. Los diferentes movimientos del brazo robótico será controlado por el joystick mediante el microcontrador.

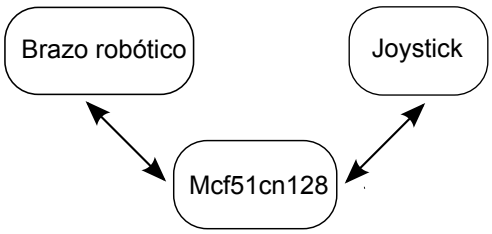


Figura 3.1: Esquema de la interacción del hardware

El joystick permite realizar el movimiento del brazo robótico con 5 grados de libertad. En la figura 3.1 no se incluyerón los elementos que permiten adecuar las señales entre estos tres elementos.

3.1. Hardware

3.1.1. Mcf51cn128

El sistema de "Torre" de Freescale es una plataforma de desarrollo modular para controladores de 8, 16, 32 bits que permite el desarrollo rápido realizando prototipos avanzados. Las características de los múltiples módulos de desarrollo proveen bloques de construcción avanzados para el microcontrolador [13].

Características

- Los módulos proveen una forma fácil de usar con un hardware modular.
- Módulos de periféricos intercambiables - serial, memoria, lcd -.
- Hardware de código abierto y con especificaciones éstandar que permite el desarrollo de módulos adicionales para ser añadir funcionalidad.
- Posee un hardware BDM[1] permitiendo que cada módulo tenga una herramienta de depuración.

Figura 3.2: Torre de Freescale [10]

La torre de demostración y evaluación TWR-MCF51CN es una plataforma reconfigurable que permite en menor tiempo el desarrollo de prototipos. Fig 3.2
La torre tiene 2 ranuras con conectores a cada lado que se insertan en el elevador. Cada módulo se puede insertar en cualquier ranura del elevador ejerciendo presión. Fig 3.3

Figura 3.3: Sistema modular implementado [10]

[1] Background Debug Mode. Es una interface que permite depurar el código en sistemas empotrados.

Las características de la TWR-MCF51CN estan en la figura 3.4 donde se observan periféricos como adc, I^2C, Rtc, Spi, Sci, Tpm, Kbi, Mtim8, memoria Flash y SRam.

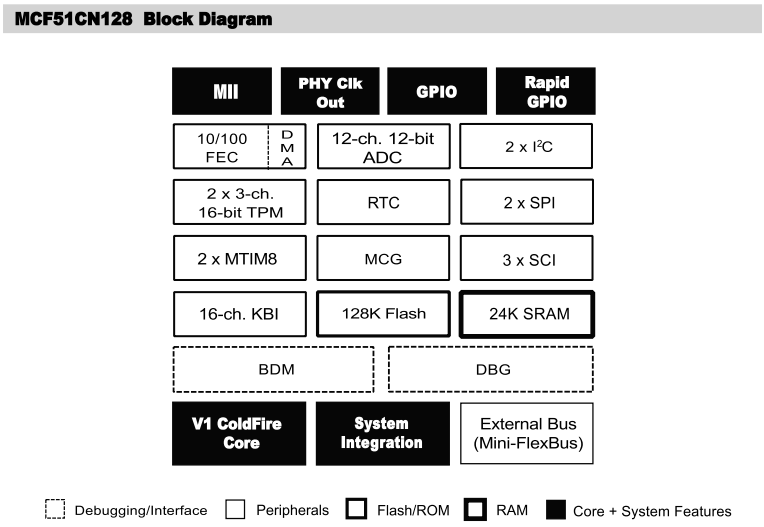


Figura 3.4: Características del sistema modular [10]

3.1.2. Joystick

Un joystick[2] es un dispositivo de control de varios ejes que se usa en juegos de computadora, grúas y simuladores de avión.

Los joystick vienen en diferentes formas y varios elementos incorporados en la palanca siendo su principal característica convertir señales, movimientos o eventos del exterior analógicos forma digital.

En las clases de joystick estan los digitales y los analógicos. Los joystick digitales tienen interruptores en la base con 2 estados (encendido/apagado) para obtener la posición de la palanca mientras que los analógicos usan potenciómetros para leer el estado de la palanca de cada eje siendo más precisos que los digitales.

Los principales componentes de este dispositivo se pueden ver en la figura 3.5.

[2]Joy = Juego, Stick = Palanca

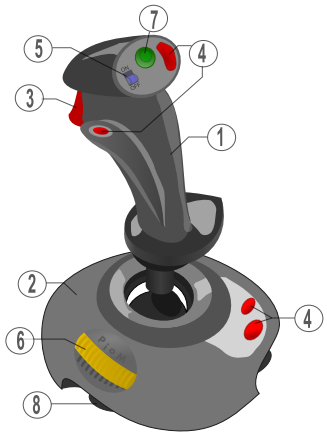


Figura 3.5: Componentes de un Joystick

1. Palanca
2. Base
3. Botón de disparo
4. Botones adicionales
5. Interruptor de autodisparo
6. Rueda
7. Botón direccional
8. Gomas

Para realizar el movimiento del brazo se utilizó un joystick Genius Flight 2000 F-23. Fig 3.6[3].

Figura 3.6: Joystick Flight 2000 F-23

Con las siguientes características (los números corresponden a la figura 3.7):

- 2 ejes de moviento de la palanca (x, y).
- 4 botones individuales (1, 2, 3, 4).
- 4 switches de direccion (9).

[3] www.geniusnet.com

- 2 ruedas que simulan los ejes z,w (10, 11).
- 4 botones auxiliares (5, 6, 7, 8)

Figura 3.7: Características del Joystick utilizado para mover el brazo robótico

Para conocer la posición de los ejes de la palanca y de las ruedas se utilizan potenciómetros con resistencias de valor 100 [kΩ] no lineales.

El valor de la resistencia del potenciómetro cuando esta sin movimiento depende de la posición de una ranura que se encuentra en la parte inferior de la base y permite seleccionar un valor de tensión de 0 - Vdd, donde Vdd es la tensión de alimentación del potenciómetro. El valor tensión seleccionado fue :

$$V_r = \frac{Vdd}{2} \tag{3.1}$$

para tener el mismo rango de excursión hacia arriba y abajo.

El diagrama de conexión de los potenciómetros se puede observar en la figura 3.8

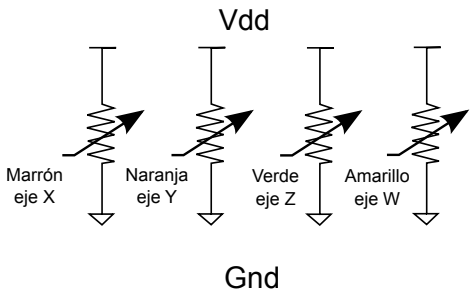


Figura 3.8: Esquema de conexión de los potenciómetros.

Donde Vdd es la tensión de alimentacion del microcontrolador y Gnd es tierra. Cada potenciómetro tiene asignado un color que corresponde al eje. En el dispositivo existen 8 botones (4 individuales y 4 direccionales) y vienen conectados según la figura 3.9:

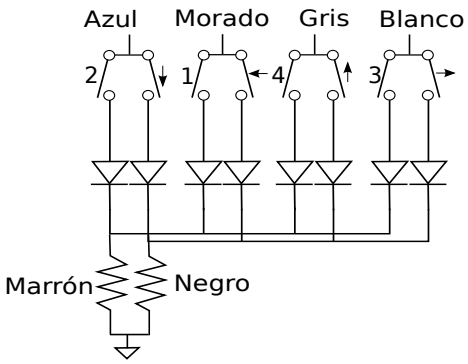


Figura 3.9: Conexión interna de los botones.

Los colores corresponden a los conectores que salen del joystick. Los números corresponden a la codificación de los botones individuales y las flechas corresponden a los botones direccionales.

3.1.3. Brazo Robótico

La estructura del brazo puede ser tan diversa la imaginación de los fabricantes lo deseen pero por lo general se distinguen 4 partes fundamentales:

- Pedestal
- Cuerpo
- Brazo
- Antebrazo

Debido a la estructura de las articulaciones y al número de ellas, el brazo del robot puede llegar a alcanzar ciertos puntos del espacio, pero nunca puede llegar todos los puntos. Al conjunto de puntos en el espacio que el brazo puede llegar a alcanzar se denomina **campo de acción** y es una característica propia de cada robot.

Las articulaciones pueden ser giratorias o moverse linealmente. El número de elementos del brazo y sus articulaciones determinan una característica propia. Al número de movimientos independientes entre sí se le denomina **grados de libertad**.

La mayoría de los brazos robóticos tienen una base para sostenerse sobre una plataforma rígida, un cuerpo donde se integran los motores que lo hacen mover y la estructura que permite el movimiento del brazo en las diferentes direcciones.

Figura 3.10: Brazo robótico

El brazo robótico utilizado 3.10[4] tiene 5 motores DC independientes lo que permite el movimiento del brazo en 5 direcciones o 5 grados de libertad. Tiene 2 movimientos giratorios, 2 movimientos lineales y 1 movimiento de apertura y cierre correspondientes a la pinza. Fig 3.11

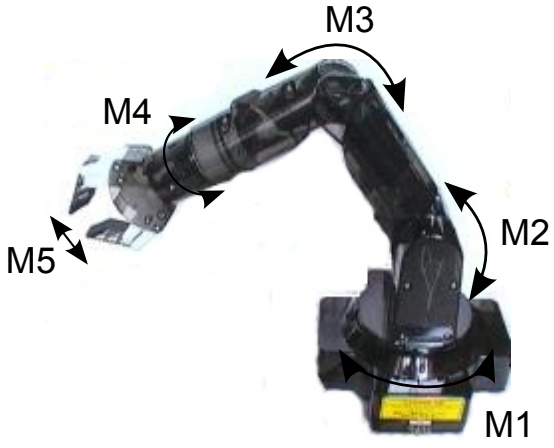


Figura 3.11: Movimientos que realiza el Brazo

Puentes H

[4] www.owirobot.com

Un puente H es un circuito electrónico que permite a un motor de corriente directa girar en ambos sentidos. Son ampliamente usados en robótica. El término H proviene de la representación gráfica del circuito. Esta consitiuido con 4 interruptores mecánicos o de estado sólido.

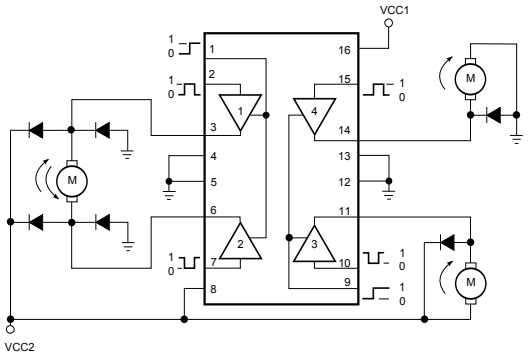


Figura 3.12: Dibujo simplificado de un puente H

En el manejo de los 5 motores de continua del brazo se utilizarón 3 circuitos integrados de referencia 4GDSTIM L293NE, con las siguientes características son:

- VCC_1 hasta 36 [V].
- VCC_2 hasta 36 [V].
- V_I hasta 7 [V].
- V_O desde -3 hasta $VCC_2 + 3$ [V].
- I_O hasta ± 1 [A].

Optoacopladores

Un optoacoplador es un dispositivo que funciona como interruptor de luz. La luz emitida por un diodo led satura un fototransistor y se utiliza para aislar electrícamente dispositivos sensibles. En la figura 3.13 se observa la conexión típica.

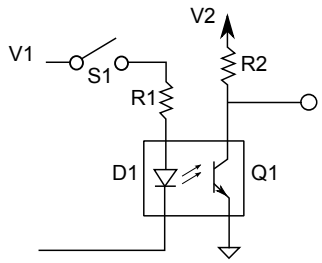


Figura 3.13: Esquema del funcionamiento de los optoacopladores

La resistencia R1 limita la corriente que pasa por el diodo led (D1). El valor de esta resistencia es 300 [Ω] obtener un valor de corriente 100 [mA] con una tensión en los pines de salida del microcontrolador de 3.0 [V] y según la ecuación Ec. 3.2:

$$I = \frac{V}{R} = \frac{3\ [V]}{300\ [\Omega]} = 100\ [mA] \tag{3.2}$$

La resistencia R2 se utiliza como *pull up* y tiene un valor de 50 [kΩ] para permitir los valores lógicos necesarios para el correcto funcionamiento de los puentes H.

Los optoacopladores que se utilizarón fuerón L0831 LTV4N25 y tiene las siguientes características generales:

- Voltaje de aislamiento 5300 V_{RMS}.
- Compatible con las Familias Lógicas.
- Capacitancia de entrada y salida <0,5 pF.

3.2. Software

En la ejecución del sistema operativo se crearón 5 tareas, dos de las cuales son temporales (se eliminan después de cumplir su función), y tres permanecen ejecutándose indefinidamente. También se crearón 2 colas para guardar los datos, una temporal y otra permante.

En la figura 3.14 se puede observar un esquema general del diseño realizado. En la parte superior se encuentran las primeras tareas en ser creadas (Calibrar X, Calibrar Y). Estas tareas utilizan una cola de un ítem (Queue)para recibir el dato de la conversión del adc. La cola Queuef permite almacenar el valor del adc después de oprimir el botón aceptar (botón 2 del joystick). Cuando los ejes han sido calibraods se eliminan las tareas de calibración para liberar memoria y se crean 3 tareas (X, Y, Botones) y una exclusión mutua que permite el acceso al adc.

Debido al tiempo de establecimiento inverso de los diodos por el orden de miliseguntos y con una frecuencia de operación del microcontrolador de 50 [MHz] se diseñarón las tareas de la siguiente manera para un correcto funcionamiento.

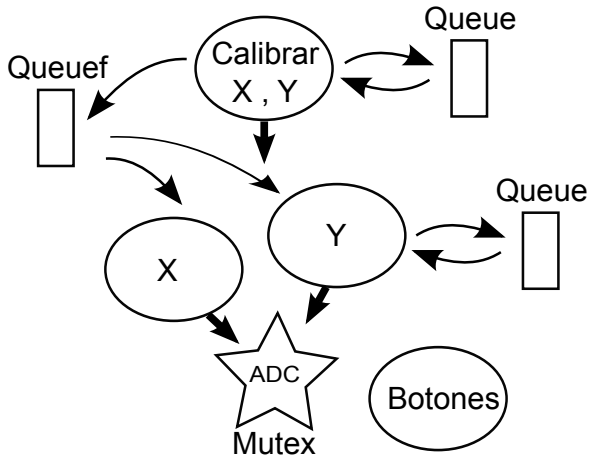


Figura 3.14: Esquema general del diseño

3.2.1. Tareas Calibrar X, Y

La primera tarea que se crea en el sistema operativo es *Calibrar X* y permite la calibración del eje **x** del joystick adquiriendo el valor de tensión del potenciómetro mediante el adc1 cuando el brazo se encuentra sin movimiento. Esta tarea permanece en el estado *ejecutándose* hasta que se oprima el botón N 2. Como no existen más tareas creadas, estará en este estado sin suspenderse ni bloquearse.

Después de presionar el botón aceptar, el valor de tensión leido por el adc1 en el eje **x** se envia a una cola donde se guarda hasta que sea tomado por una tarea que se crea más adelante. Al cumplir la función de calibración crea la tarea *Calibrar Y* y después se elimina. La función de borrado de la tarea de la memoria la realiza el sistema operativo mediante el llamado del comando respectivo.

La tarea *Calibrar Y* permite calibrar el eje **y** del joystick adquiriendo el valor de tensión del potenciómetro mediante el adc2 cuando el brazo se encuentra sin movimiento. Al no existir ninguna tarea ni bloqueada ni suspendida estará en el estado *ejecutándose* hasta que se oprima el botón aceptar. Al oprimir este botón el valor de tensión leido por el adc2 en el eje **y** es enviado a la cola donde se encuentra guardado el valor del eje **x**.

Al finalizar de calibrar el eje **y** se crean 3 tareas y después se elimina la tarea actual. Se crean las tareas **X**, **Y** y **Botones**.

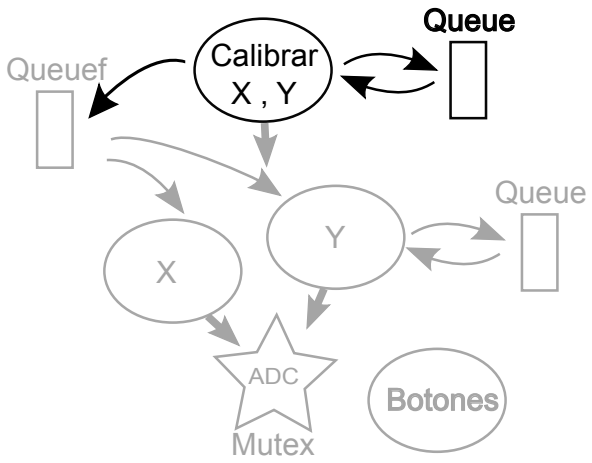


Figura 3.15: Diagrama de creación de las tareas

3.2.2. Tarea Botones

Debido los diodos presentes en la codificación de los botones la tarea **Botones** se debe bloquear un tick (1 [ms]) y verificar si algún botón del joystick es presionado. La secuencia seguida es colocar en alto el valor del color azul mientras que el valor de los demás colores se encuentran en bajo y se espera un tick mientras que los diodos se estabilizan en un estado (corto o abierto) para leer el valor en las resistencias *pull-down*. Si el valor leido en las resistencias (Marron, Negro Fig 3.9) es bajo entonces no se ha presionado ningún botón. Dependiendo en cual resistencia se haya leido el valor en alto y el color que se encuentre en alto es posible saber que botón fue presionado. Los colores que se colocan en valor alto son Azul, Morado, Gris, Blanco respectivamente.

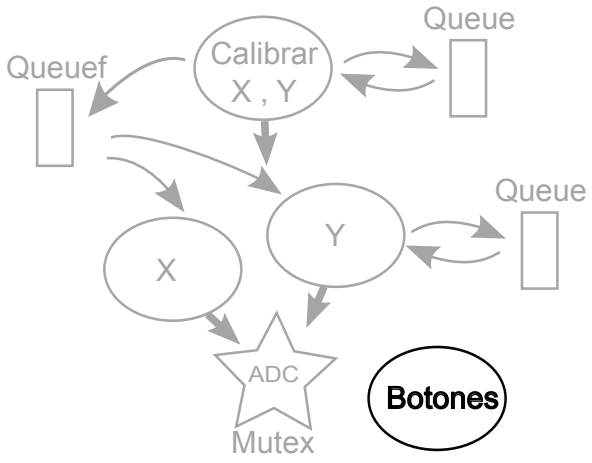


Figura 3.16: Tarea Botones

Al tener mayor prioridad sobre las demás tareas es la primera que se encuentra en el estado *ejecutándose* y se desbloquea periódicamente.

```
void Botones(void *pvParameters)
{
  Azul();
  Bloquearms();
  ComprobarValorResistencias();
  Mover();

  Morado();
  Bloquerms();
  ComprobarValorResistencias();
  Mover();

  Gris();
  Bloquearms();
  ComprobarValorResistencias();
  Mover();

  Blanco();
  Bloquearms();
  ComprobarValorResistencias();
  Mover();
}
```

Código 3.1: Tarea Botones.

3.2.3. Tareas X, Y

Las tareas **X**, **Y** son de menor prioridad que la tarea Botones, por lo tanto se ejecutarán cuando no exista ninguna tarea en el estado *ejecutándose* con mayor priodidad. El primer paso es coger el dato guardado en la cola por las tareas Calibrar Y y X y realizar un *offset* para no realizar ninguna acción mientras no existe movimiento del joystick.

El segundo paso es adquirir el permiso para la exclusión mutua y acceder al recurso protegido que esta conformado con una cola y el adc. En la cola se guarda el valor convertido por el adc y se compara si es mayor que el valor tomado mas el offset.Cuando la tarea **X** cede el permiso de la exclusión mutua es tomada por la tarea **Y** y realiza el mismo procedimiento. Fig 3.17

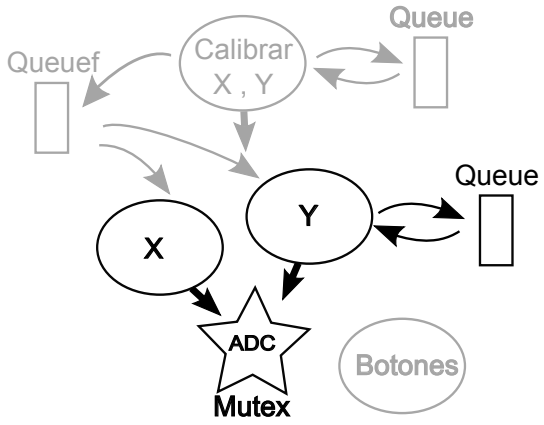


Figura 3.17: Tareas X, Y

```
Hay permiso para acceder a la exclusion mutua?
   {
     eje;//Envia la orden de convertir el eje
     if(Existe un dato en la cola?)
     {
       datorecibido>xalto?
         Mover motor derecha;

       datore<xbajo?
         Mover motor izquierda;
     }
     Ceder permiso exclusion mutua();
   }
```

Código 3.2: Adquieriendo la exclusión mutua.

Capítulo 4

Resultados del diseño

En este capitulo se realiza la comparación de resultados obtenidos con la aplicación de la metodología tradicional mediante interrupciones y la aplicación de un sistema operativo en tiempo real. El sistema operativo MQX no fue tenido en cuenta en la comparación debido a la que no fue posible encontrar documentación sobre el manejo de puertos entrada / salida.

4.1. Interrupciones

La programación mediante foreground / background se realizó siguiendo los mismos pasos que la programación en los sistemas operativos. Se crearón funciones que representaban las tareas y desarrollan el mismo procedimiento para conocer la posición de los ejes x e y.

4.1.1. Memoria

Según el reporte realizado por CodeWarrior en el momento de la compilación, el tamaño de la memoria utilizada es 2.292 [Kbytes].

4.1.2. Tiempo

Para tener la misma frecuencia de los sistemas operativos se utilizó un oscilador externo de 25 [MHz] y un PLL duplicarla y llegar a 50 [MHz].

Para realizar las interrupciones temporales de 1 [ms] requeridas para el correcto funcionamiento de los botones del joystick se utilizó el contador de tiempo real (RTC) que viene en los periféricos del microcontrolador.

Como en cualquier programación Foreground/Background existen dos tareas (primer/segundo) plano.

```
void main(void)
{ static unsigned int x,y,yalto,ybajo,xalto,xbajo;

  Inicializacion de perifericos();

  calibrar(x); calibrar(y);
```

```
    for(;;)
    {  ejey; //Se convierte eje y del joystick
      if(datoadc<(ybajo))
        {MotorDerecha;}
      else
        {NoMover;}

      if(datoadc>(yalto))
        {MotorIzquierda;}
      else
        {NoMover;}

      ejex; //Se convierte eje x del joystick
      if(datoadc<(xbajo))
        {MotorAbajo;}
      else
        {NoMover;}

      if(datoadc>(xalto))
        {MotorArriba;}
      else
        {NoMover;}
    }
}
```

Código 4.1: Tarea que se ejecuta en primer plano

En el código 4.1 se observa la tarea que se implemento en primer plano. Primero se declaran las variables que se van a utilizar y se inicializan los periféricos (adc, puertos de entrada/salida, rtc). Las siguientes funciones son *Calibrar* donde el sistema toma el dato mientras el joystick se encuentra sin movimiento y crea un offset. El offset creado es el mismo de los sistemas operativos.

En el bucle infinito se realiza una conversión del valor de los potenciómetros y se compara con las variables (x,y)alto, (x,y)bajo para conocer si existe un cambio en la posición de la palanca, si la condición es verdadera envia el comando para mover el motor respectivo.

```
num =0;
void rtc()
{
  LimpiarBanderaRTC();
  switch(num)
  {
    case 0: if(Ne==1){CerrarPinza;}else{NoMover;}
            if(Ma==1){BajarBrazo;}else{NoMover;}
            PTDD=Mo;num++;break;

    case 1:
            if(Ma==1){SubirBrazo;}else{NoMover;}
            PTDD=Gr;num++;break;

    case 2: if(Ne==1){AbrirPinza;}else{NoMover;}
```

```
            if(Ma==1){BrazoIzq;}else{NoMover;}
            PTDD=B1;num++;break;

    case 3:
            if(Ma==1){BrazoDer;}else{NoMover;}
            PTDD=Az;num=0;break;
}
```

Código 4.2: Tarea que se ejecuta en segundo plano

El código 4.2 fue implementado como tarea de segundo plano y asociado a una interrupción. El código genera una secuencia que recorre los *case* empezando por el número cero (0) y siguiendo los números uno (1), dos (2) y tres (3). Al iniciar el sistema la variable *num* se hace cero obligando al sistema a entrar al código *case 0:*. La comparación de los valores altos se hace sobre las resistencias Ne (negro) y Ma (marrón). Si la condición es verdadera envia el comando para mover el motor respectivo, de lo contrario no realiza ninguna acción. Despue's se cambia el valor del puerto y se aumenta la variable *num* para que la condición verdadera sea *case 1:*. El valor de los puertos van cambiando y la variable *num* aumenta hasta el número tres para recorrer los *case 1:, case 2: y case 3:*. En esta última comparación la variable se le vuelve el número cero para volver a comenzar.

4.2. FreeRTOS

FreeRTOS permite deshabilitar las funciones que no se utilizan en la programación mediante el archivo de configuracion *FreeRTOSConfig.h*. Para tener un óptimo tamaño de memoria se modificarón los siguientes parámetros:

```
#define configUSE_PREEMPTION 1
#define configUSE_TICK_HOOK 1
#define configCPU_CLOCK_HZ ( 50000000UL )
#define configTICK_RATE_HZ ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 256 )
#define configTOTAL_HEAP_SIZE ( ( size_t ) ( 13 * 1024 ) )

#define configMAX_TASK_NAME_LEN ( 12 )
#define configUSE_TRACE_FACILITY 1
#define configUSE_MUTEXES 1
#define configCHECK_FOR_STACK_OVERFLOW 1
#define configMAX_PRIORITIES ( ( unsigned portBASE_TYPE ) 6 )
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_vTaskDelay 1
#define INCLUDE_uxTaskGetStackHighWaterMark 1
```

Código 4.3: Parámetros modificados para el sistema operativo FreeRTOS

Los demás parámetros se deshabilitarón colocando un valor en cero.

4.2.1. Memoria

Según el reporte realizado por CodeWarrior en el momento de la compilación, el tamaño de la memoria utilizada es 8.031 [Kbytes].

4.2.2. Tiempo

FreeRTOS utiliza un oscilador externo de 25 [MHz] con un PLL integrado en el microcontrolador para aumentar la frecuencia a 50 [MHz].

Para realizar las interrupciones temporales correspondientes a los ticks el sistema operativo utiliza el contador de tiempo real (RTC) que viene en los periféricos del microcontrolador. Los ticks ocurren cada 1 [ms].

```
static void prvSetupTimerInterrupt( void )
{ // Prescale by 1 - ie no prescale.
RTCSC |= 8;
// Compare match value.
RTCMOD = portRTC_CLOCK_HZ / configTICK_RATE_HZ;
// Enable the RTC to generate interrupts -
        interrupts are already disabled
when this code executes.
RTCSC_RTIE = 1;
}
```

Código 4.4: Forma de realizar la interrupción de los ticks

4.3. uC/OS II

uC/OS II es un software escalable, es decir, se puede utilizar solo los servicios necesarios. En base a este hecho se modifico el *os_cfg.h* con los siguientes parámetros para optimizar el tamaño de la memoria:

```
#define OS_APP_HOOKS_EN              1
#define OS_LOWEST_PRIO              20
#define OS_MAX_QS                    4
#define OS_MAX_TASKS                10
#define OS_SCHED_LOCK_EN             1
#define OS_TICK_STEP_EN              1
#define OS_TICKS_PER_SEC          1000
#define OS_TASK_TMR_STK_SIZE       128
#define OS_TASK_STAT_STK_SIZE      128
#define OS_TASK_IDLE_STK_SIZE      128
#define OS_TASK_CREATE_EN            1
#define OS_TASK_DEL_EN               1
#define OS_TASK_SUSPEND_EN           1
#define OS_Q_EN                      1
#define OS_SEM_EN                    1
#define OS_TMR_EN                    1
```

Código 4.5: Parámetros modificados para el sistema operativo uC/OS II

4.3.1. Memoria

Según el reporte realizado por CodeWarrior en el momento de la compilación, el tamaño de la memoria utilizada es 14.762 [Kbytes].

4.3.2. Tiempo

uC/OS II no utiliza oscilador externo de 25 [MHz] para tener la frecuencia, sino utiliza un multiplicador de frecuencia (PLL) para elevarla de 9 a 50 [MHz].

Para generar la interrupción de cada tick uC/OS II utiliza un timer con un periodo de 1 [ms].

```
static void  OSTickISR_Init (void)
{
  CPU_INT32U  bus_freq;

  bus_freq  = BSP_CPU_ClkFreq();
  bus_freq /= 2;
  OSTickCnts = (CPU_INT16U)(bus_freq / (8 * OS_TICKS_PER_SEC));
  SCGC1 |= SCGC1_TPM1_MASK;
  TPM1SC  = 0;
  TPM1MOD = 0;
#if (OS_TICK_TPM1_CH == 0)
  TPM1C0SC = TPM1C0SC_MS0A_MASK |
             TPM1C0SC_CH0IE_MASK;
  TPM1C0V  = TPM1CNT + OSTickCnts;
#endif
#if (OS_TICK_TPM1_CH == 1)
  TPM1C1SC = TPM1C1SC_MS1A_MASK |
             TPM1C1SC_CH1IE_MASK;
  TPM1C1V  = TPM1CNT + OSTickCnts;
#endif
#if (OS_TICK_TPM1_CH == 2)
  TPM1C2SC = TPM1C2SC_MS2A_MASK |
             TPM1C2SC_CH2IE_MASK;
  TPM1C2V  = TPM1CNT + OSTickCnts;
#endif
  TPM1SC |= (1 << TPM1SC_CLKSx_BITNUM) |
            (3 << TPM1SC_PS_BITNUM);
}
```

Código 4.6: Forma de realizar la interrupción de los ticks

4.4. Cuadro Comparativo

	Software implementado		
Característica	Interrupciones	FreeRTOS	uC/OS II
Autor		Richard Barry	Jean J. Labrosse
Memoria [Kbytes]	2.292	8.031	14.762
Ventajas	Memoria	Flexibilidad	Flexibilidad
Scheduler	No aplica	*Preemptivo*	*Preemptivo*
Escalabe	No aplica	Si	Si
Creación de tareas (us)	No aplica	153.16	208.76
Cambio de contexto (us)	No aplica	557.34	698.94
Acceso a recursos (us)	20.56	23.53	31.87
Lenguaje de programación	C	C	C
Licencia	No aplica	Totalmente Libre	Libre para educación
Arquitectura Soportada (Coldfire V1)	Si	Si	Si
Amplia Documentación	Si	Si	Si

Tabla 4.1: Cuadro comparativo

Las principales características de la programación mediante Foreground / Background y la programación de los sistemas operativos en tiempo real se resumen en el cuadro 4.1. En este cuadro se presentan aspectos como autor, memoria, tipo de licencia, lenguaje de programación, documentación. Todos los sistemas operativos tienen la frecuencia de oscilación.

Una de las grandes diferencias entre los tres métodos de programación es el tamaño de memoria usada siendo una de los recursos que necesita ser considerado en los sistemas empotrados. Otra diferencia entre los sistemas operativos es que FreeRTOS es muy simple y para pequeños proyectos puede ser eficiente, sin embargo para grandes proyectos puedan necesarias funciones extras que vienen incluidas en uC/OS II.

La mayoría de los sistemas incluyen una parte que tiene un sección crítica y otra que no lo es. Si la parte no crítica es mejor utilizar sistemas operativos que contengan varias funciones especializadas para realizar estas tareas, pero si por el contrario la parte no crítica no es grande es mejor utilizar sistemas operativos que tengan funciones genéricas para poder reutlizarlas y asi tener un código minímo.

Una parte importante considerada en los proyectos es la económica. En proyectos comerciales la facilidad de usar licencias no existe. En estos casos se puede usar FreeRTOS sin necesidad de pagar por el uso del sistema operativo, mientras que uC/OS II no se puede.

Para pequeños sistemas empotrados con limitado tamaño de memoria FreeRTOS probablemente será el sistema operativo elegido, pero si tiene un sistema con gran tamaño de memoria o memoria expandible uC/OS II sera el sistema operativo que mayor posibilidad tenga de expandirse.

Conclusiones

- Un sistema operativo en tiempo real permite mayor flexibilidad en el diseño y programación de aplicaciones en comparación con el método tradicional de interrupciones, porque tiene funciones optimizadas permitiendo una mejor distribución del tiempo del procesador.
- En la comparación de los 3 sistemas operativos en tiempo real se observarón las diferencias de cada sistema y la manera de administrar los recursos del sistema empotrado, así como la adaptación del sistema operativo a los requisitos de la aplicación.
- Una característica importante en el desarrollo del proyecto es la parte de implementación de la aplicación debido a que presenta componentes físicos en la mayoría mecánicos presentando variables limitantes, siendo necesaria una caracterización del sistema para conocer los límites de la aplicación.
- Los sistemas operativos en tiempo real utilizados en los sistemas empotrados utilizan tiempo del procesador para que el *scheduler* determine cual tarea realiza un cambio de contexto y pasa al estado ejecutándose. El tiempo usado por el sistema operativo para realizar las decisiones depende de la frecuencia del sistema empotrado, entre mayor sea la frecuencia del sistema menor será el tiempo utilizado por el *scheduler*.

  Este tiempo puede ser minímo en comparación con la duración de las tareas y el cambio de contexto se realiza lo suficientemente rápido para no interferir con el desarrollo normal de la aplicación.
- Los sistemas empotrados ofrecen un entorno uniforme, estándar y reconfigurable en software porque se puede realizar un cambio parcial o total en el sistema con mínimas modificaciones en el hardware y la aplicación se puede ejecutar en cualquier sistema empotrado independiente del tamaño de sus recursos.

Recomendaciones para trabajos futuros

- Una mejora a implementar en el proyecto es cambiar la manera de codificación de los botones reemplazando los diodos por interruptores o implementado un un circuito que reduzca el tiempo de recuperación inversa.
- Los motores utilizados no ofrecen la posibilidad de realimentación, no permitiendo conocer la posición exacta del brazo. Un cambio en este sentido será agregar componentes externos de realimentación o cambiar los motores por otros elementos que ya lo tengan incorporado como son los servomotores.
- El sistema TWR-MCF51CN128 tiene integrado el componenete TCP/IP permitiendo realizar comunicación con el computador a través de Internet. Con base a esto se puede realizar el control del brazo robótico remotamente mediante una página web.

Bibliografía

[1] Q. Zhou and K. J. Balakrishnan, "Test cost reduction for soc using a combined approach to test data compression and test scheduling," *Proceeding Desing, Automation, and Test in Europe (DATE)*, p. 39, 2007.

[2] A. K. B. Salem and S. B. Saoud, "Rtos evolution and hardware microkernel implementation benefits," *African Phisical Review Special Issue*, pp. 10–12, 2008.

[3] L. F. Añover, "Embedded systems: Una lanza por «sistemas empotrados»," *ec.europa.eu/translation/bulletins/puntoycoma/105/pyc1053_es.htm*, Visitada en Octubre 2010.

[4] S. Heath, "Embedded system design," *Series for Design Engineers*, no. ISBN 0 7506 5546 1, p. 422, 2003.

[5] J. J. Labrosse, "Micro c/os-ii," *The Real-Time Kernel*, no. ISBN-13: 978-1-57820-103-7, p. 606, 2002.

[6] J. D. M. Frías, "Sistemas empotrados en tiempo real," *Una introducción basada en FreeRTOS y en microcontroladores Coldfire*, vol. Primera Edición., p. 181, Febrero 2009.

[7] R. Barry, "Using the freertos real time kernel," *A Practical Guide*, p. 185, 2010.

[8] S. R. Reddy, "Selection of rtos for an efficient design of embedded systems," *International Journal of Computer Science and Network Security*, pp. 29 –37, Junio 2006.

[9] S.-L. TAN and T. N. B. Anh, "Real-time operating system (rtos) for small (16-bit) microcontroller," *IEEE International Symposium on Consumer Electronics*, p. 5, 2009.

[10] "Freescale," *www.freescale.com*, Visitada en Octubre 2010.

[11] FreeRTOS, "A portable, open source, royalty free, mini real-time kernel," *www.freertos.org*, Visitada en Octubre 2010.

[12] uC/OS II, "A scalable, portable, preemptive, real-time deterministic.," *www.micrium.com*, Visitada en Octubre 2010.

[13] "Sistema torre," *www.freescale.com/webapp/sps/site/overview.jsp?code=TOWER_HOME*, Visitada en Octubre 2010.

Anexo A

Código Interrupciones

```
#include <hidef.h>
#include <stdio.h>
#include "derivative.h"
#include "7Seg.h"
#include "adc.h"
#include "extal.h"
#include "rtc.h"
#include "Botones.h"

unsigned int datoadc;
unsigned char num=0;

void main(void)
{
  static unsigned int x,y,yalto,ybajo,xalto,xbajo;
  seg_init();
  adc_init();
  extal_init();
  EnableInterrupts;

  x=calibrar(1); y=calibrar(0);

  xalto=x+offset; yalto=y+offset;
  xbajo=x-offset; ybajo=y-offset;

  PTDD=Az; rtc_init();

  for(;;)
  {
    ejey;
    if(datoadc<(ybajo))
      {M2d=1;}
    else
      {M2d=0;}

    if(datoadc>(yalto))
      {M2i=1;}
    else
      {M2i=0;}

    ejex;
    if(datoadc<(xbajo))
      {M1b=1;}
```

```
    else
      {M1b=0;}

    if(datoadc>(xalto))
      {M1r=1;}
    else
      {M1r=0;}
  }
}

interrupt VectorNumber_Vadc
void adc()
{
  datoadc=ADCR;
}

interrupt VectorNumber_Vrtc
void rtc()
{
  RTCSC_RTIF=1;
  switch(num)
  {
    case 0: if(Ne==1){M5c=1;}else{M5c=0;}
            if(Ma==1){M3b=1;}else{M3b=0;}PTDD=Mo;num++;break;
    case 1:
            if(Ma==1){M3r=1;}else{M3r=0;}PTDD=Gr;num++;break;
    case 2: if(Ne==1){M5a=1;}else{M5a=0;}
            if(Ma==1){M4i=1;}else{M4i=0;}PTDD=Bl;num++;break;
    case 3:
            if(Ma==1){M4d=1;}else{M4d=0;}PTDD=Az;num=0;break;
  }
}
```

Anexo B

Código uC/OS II

```
#include <includes.h>
#include <stdio.h>
#include "adc.h"
#include "7Seg.h"
#include "Botones.h"

static void Calibrarx(void *p_arg);
static void Calibrary(void *p_arg);
static void x(void *p_arg);
static void y(void *p_arg);
static void Botones(void *p_arg);

OS_STK   CalibrarxStk[APP_TASK_START_STK_SIZE],
         CalibraryStk[APP_TASK_START_STK_SIZE],
         xStk[APP_TASK_START_STK_SIZE],
         yStk[APP_TASK_START_STK_SIZE],
         BotonesStk[APP_TASK_START_STK_SIZE];

OS_EVENT   *CommQ,*CommQf;
void *CommMSG[1],*CommQfMSG[2];
OS_EVENT *DispSem;

void   main (void)
{   BSP_Init();
  #if (OS_TASK_STAT_EN > 0)
       OSStatInit();
  #endif
   BSP_IntDisAll();
   seg_init();   adc_init();     OSInit();

  CommQ=OSQCreate(&CommMSG[0],1);
  CommQf=OSQCreate(&CommQfMSG[0],2);
  DispSem=OSSemCreate(1);

   OSTaskCreate(Calibrarx,(void *)0,
                (OS_STK *)&CalibrarxStk[APP_TASK_START_STK_SIZE-1], 5);

     OSStart();
}

static void Calibrarx(void *p_arg)
{ CPU_INT08U err;
  CPU_INT16U most;
```

```
  void   *datore;

  (void)p_arg;
  PTDD=Az;

  while(Ma==0)
  { ejex;
    datore=OSQPend(CommQ,0,&err);
    if(datore!=NULL)
      { most=(unsigned int)datore;
        mostrar_datos(most);
      }
  }
  OSQPost(CommQf,(void *)datore);

  OSTaskCreate(Calibrary,(void *)0,
               (OS_STK *)&CalibraryStk[APP_TASK_START_STK_SIZE-1],6);

  OSTaskDel(5);
}

static void Calibrary(void *p_arg)
{ CPU_INT08U err;
  CPU_INT16U most;
  void   *datore;

  (void)p_arg;
  PTDD=Az;

  while(Ma==0)
  { ejey;
    datore=OSQPend(CommQ,0,&err);
    if(datore!=NULL)
      { most=(unsigned int)datore;
        mostrar_datos(most);
      }
  }
  OSQPost(CommQf,(void *)datore);

  OSTaskCreate(x,(void *)0,
               (OS_STK *)&xStk[APP_TASK_START_STK_SIZE-1],9);

  OSTaskCreate(y,(void *)0,
               (OS_STK *)&yStk[APP_TASK_START_STK_SIZE-1],8);

  OSTaskCreate(Botones,(void *)0,
               (OS_STK *)&BotonesStk[APP_TASK_START_STK_SIZE-1],7);

  OSTaskDel(6);
}

static void x(void *p_arg)
{ CPU_INT08U err,xalto,xbajo;
  CPU_INT08U temp;

  void *datore;
  (void)p_arg;

  datore=OSQPend(CommQf,0,&err);
  xalto=(unsigned int)datore+offset;
  xbajo=(unsigned int)datore-offset;
```

```
  while(DEF_TRUE)
  {   OSSemPend(DispSem,0,&err);

      ejex;
      datore=OSQPend(CommQ,0,&err);
      temp=(unsigned int)datore;

      if(temp>xalto)
        {M1r=1;}      else      {M1r=0;}

      if(temp<xbajo)
        {M1b=1;}      else      {M1b=0;}

    OSSemPost(DispSem);
    OSTimeDlyHMSM(0, 0, 0, 2);
  }
}

static void y(void *p_arg)
{ CPU_INT08U err,yalto,ybajo;
  CPU_INT08U temp;

  void *datore;
  (void)p_arg;

  datore=OSQPend(CommQf,0,&err);
  yalto=(unsigned int)datore+offset;
  ybajo=(unsigned int)datore-offset;

  while(DEF_TRUE)
  {   OSSemPend(DispSem,0,&err);

      ejey;
      datore=OSQPend(CommQ,0,&err);
      temp=(unsigned int)datore;

      if(temp>yalto)
        {M2i=1;}      else      {M2i=0;}

      if(temp<ybajo)
        {M2d=1;}      else      {M2d=0;}

    OSSemPost(DispSem);
    OSTimeDlyHMSM(0, 0, 0, 3);
  }
}

static void Botones(void *p_arg)
{ CPU_INT08U time;
  (void)p_arg;

  while(DEF_TRUE)
  {
    PTDD=Az;
    OSTimeDlyHMSM(0, 0, 0, time);

    if(Ne==1)
      {M5c=1;}else{M5c=0;}

    if(Ma==1)
      {M3b=1;}else{M3b=0;}
```

```
      PTDD=Mo;
      OSTimeDlyHMSM(0, 0, 0, time);

      if(Ma==1)
        {M3r=1;}else{M3r=0;}

      PTDD=Gr;
      OSTimeDlyHMSM(0, 0, 0, time);

      if(Ne==1)
        {M5a=1;}else{M5a=0;}

      if(Ma==1)
        {M4i=1;}else{M4i=0;}

      PTDD=Bl;
      OSTimeDlyHMSM(0, 0, 0, time);

      if(Ma==1)
        {M4d=1;}else{M4d=0;}
   }
}

interrupt VectorNumber_Vadc
void adc()
{ CPU_INT16U i;
  i=ADCR;   OSQPost(CommQ,(void *)i);
}
```

Anexo C

Código FreeRTOS

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "hardware.h"
#include "Botones.h"
#include "sci.h"
#include "7Seg.h"
#include "adc.h"
#include "irq.h"
#include "extal.h"

xQueueHandle xQueue,xQueuef;
xSemaphoreHandle xSemaphore;

void Calibrarx(void *pvParameters);
void Calibrary(void *pvParameters);
void x(void *pvParameters);
void y(void *pvParameters);
void Botones(void *pvParameters);

int main( void )
{ seg_init();
  adc_init();
  extal_init();
  EnableInterrupts;

  xQueue=xQueueCreate(1,sizeof(int));
  xQueuef=xQueueCreate(2,sizeof(int));
  xSemaphore=xSemaphoreCreateMutex();

  if(xQueue!=NULL && xQueuef!=NULL && xSemaphore!=NULL)
    {
      xTaskCreate(Calibrarx,"Calibrar", configMINIMAL_STACK_SIZE,
      NULL, 0,NULL );

      vTaskStartScheduler();
    }
}

void Calibrarx(void *pvParameters)
{ (void)pvParameters;
  int datore;
```

```
  PTDD=Az;
  while(Ma!=1)
  { ejex;
    if(xQueueReceive(xQueue,&datore,portMAX_DELAY))
      {mostrar_datos(datore);}
  }
  xQueueSend(xQueuef,&datore,0);

  xTaskCreate(Calibrary,"Calibrary", configMINIMAL_STACK_SIZE,
  NULL, 0,NULL );
  vTaskDelete(NULL);
}

void Calibrary(void *pvParameters)
{ (void)pvParameters;
  int datore;

  PTDD=Az;
  while(Ma!=1)
  { ejey;
    if(xQueueReceive(xQueue,&datore,portMAX_DELAY))
      {mostrar_datos(datore);}
  }
  xQueueSend(xQueuef,&datore,0);

  xTaskCreate(y,"y", configMINIMAL_STACK_SIZE,NULL,1,NULL);
  xTaskCreate(x,"x", configMINIMAL_STACK_SIZE,NULL,0,NULL);
  xTaskCreate(Botones,"Botones",configMINIMAL_STACK_SIZE,
  NULL,5,NULL);

 vTaskDelete(NULL);
}

void x(void *pvParameters)
{ (void)pvParameters;
  int datore;
  unsigned int xalto,xbajo;

  xQueueReceive(xQueuef,&datore,portMAX_DELAY);
  xalto=datore+offset;
  xbajo=datore-offset;

  for(;;)
  { if(xSemaphoreTake(xSemaphore,portMAX_DELAY)==pdTRUE)
      { ejex;
        if(xQueueReceive(xQueue,&datore,portMAX_DELAY))
          { if(datore>xalto)
              {M1r=1;}          else          {M1r=0;}

            if(datore<xbajo)
              {M1b=1;}          else          {M1b=0;}
          }
       xSemaphoreGive(xSemaphore);
      }
  }
}

void y(void *pvParameters)
{ (void)pvParameters;
  int datore;
  unsigned int yalto,ybajo;
```

```
  xQueueReceive(xQueuef,&datore,portMAX_DELAY);
  yalto=datore+offset;
  ybajo=datore-offset;

  for(;;)
  { if(xSemaphoreTake(xSemaphore,portMAX_DELAY)==pdTRUE)
    { ejey;
      if(xQueueReceive(xQueue,&datore,portMAX_DELAY))
        { if(datore>yalto)
            {M2i=1;}          else          {M2i=0;}

          if(datore<ybajo)
            {M2d=1;}          else          {M2d=0;}
        }
      xSemaphoreGive(xSemaphore);
    }
  }
}

void Botones(void *pvParameters)
{ (void)pvParameters;

  for(;;)
  { PTDD=Az;
    vTaskDelay(time);
    if(Ne==1)
      {M5c=1;}else{M5c=0;}
    if(Ma==1)
      {M3b=1;}else{M3b=0;}

    PTDD=Mo;
    vTaskDelay(time);
    if(Ma==1)
      {M3r=1;}else{M3r=0;}

    PTDD=Gr;
    vTaskDelay(time);
    if(Ne==1)
      {M5a=1;}else{M5a=0;}
    if(Ma==1)
      {M4i=1;}else{M4i=0;}

    PTDD=B1;
    vTaskDelay(time);
    if(Ma==1)
      {M4d=1;}else{M4d=0;}
  }
}

interrupt VectorNumber_Vadc
void adc()
{ unsigned int f;
  f=ADCR;
  xQueueSend(xQueue,&f,0);
}
```