EXAMEN FINAL ORDINARIO 2021:

1. Responde a las siguientes preguntas:
   a. **¿Qué característica extra incorporan los sistemas *Super-Loop* para ser considerados como *Foreground-Background*?**

La prioridad inversa que permite asignar prioridades dinámicas a las tareas en tiempo real y garantizar que las tareas críticas se ejecuten en primer plano con la máxima prioridad y, así, evitar bloqueos o retrasos. Las tareas que no suponen una alta criticidad se ejecutarán en segundo plano sin afectar la ejecución de las tareas prioritarias.

   b. **¿Cómo se llama la técnica de monitoreo de eventos en sistemas *Super-Loop*?**

Polling, que se realiza continuamente para determinar si ha ocurrido alguna acción que requiera atención. Esto ocurre puesto que no se emplean interrupciones.

   c. **¿Qué es una función *non-reentrant*? Explica si este tipo de funciones pueden ser usadas en aplicaciones multi-hilo (*RTOS*).**

Una función *non-reentrant* es aquella que no puede ser compartida por más de un hilo o tarea, a menos que exista algún método de exclusión (secciones críticas o mutex), por miedo a que los datos queden corruptos durante operaciones de RW/WR.

Este tipo de funciones no deben usarse en aplicaciones multi-hilo, del tipo *RTOS*, puesto que pueden generar problemas de concurrencia y posibles condiciones de carrera. Dado que los hilos pueden ejecutarse concurrentemente y en cualquier orden, no se puede garantizar que las funciones no reentrantes mantengan la consistencia de los datos o recursos compartidos. Para garantizar que múltiples hilos no se ejecuten en una función *non-reentrant*, se pueden emplear mecanismos de bloqueo, como semáforos o mutex, para evitar problemas de concurrencia.

   d. **¿Una función *re-entrant* puede emplear variables globales? Explica que recurso del *RTOS* utilizarías para tratar variables globales con seguridad.**

No es recomendable que una función *re-entrant* emplee variables globales, ya que son compartidas por todas las funciones y pueden ser modificadas por múltiples hilos simultáneamente. Si este tipo de funciones utilizan estas variables puede ocurrir una condición de carrera y, así, provocar resultados incorrectos o inesperados. Por lo tanto, se recomienda usar variables locales, por lo que cada instancia de la función tendrá sus propias variables y no se producirán conflictos de concurrencia.

En un *RTOS*, un recurso usado para tratar las variables globales de manera segura es el mutex, que se utiliza para proporcionar exclusión mutua, es decir, solo un hilo puede tener acceso a las variables globales protegidas por el mutex en un momento dado. Cuando un hilo necesita acceder a las variables globales, debe adquirir el mutex asociado antes de acceder a las variables y luego liberarlo una vez que haya terminado. Esto garantiza que solo un hilo pueda acceder a las variables globales protegidas por el mutex, evitando condiciones de carrera y manteniendo la consistencia de los datos.

e. Explica el concepto de *Real-Time*, en el ámbito de los *RTOS*.

Un sistema en *Real-Time* es aquel en el que la exactitud del cómputo realizado no sólo depende de la exactitud lógica del cómputo, si no también del instante de tiempo en el que se produjo. Si las restricciones de tiempo no se cumplen, se dice que el sistema ha fallado.

Un *RTOS* está diseñado para procesar aplicaciones que requieran tiempos precisos con un alto grado de fiabilidad y tiempos predecibles y determinísticos. Para ser considerado como *Real-Time*, el sistema operativo debe saber el tiempo máximo de ejecución de cada operación crítica:

- Procesamiento de las interrupciones y excepciones internas del sistema.
- Secciones críticas: Proteger estas zonas ante posibles *context switch* o interrupción.
- Mecanismo de planificación del kernel (*Scheduling*): *Preemptive* (expropiativo), cooperativo, modalidad Round-Robin.
- Objetos de kernel: Queue, semáforos binarios, mutex…

f. Expón, en detalle, los conceptos erróneo que suelen tenerse sobre las *RTAs*.

- La computación en tiempo real es equivalente a procesar líneas de código a alta velocidad. La computación en tiempo real se refiere a la capacidad de una aplicación para cumplir con restricciones temporales específicas, es decir, realizar tareas dentro de plazos establecidos. No se trata solo de procesar líneas de código rápidamente, sino de garantizar que las tareas se completen dentro de los límites de tiempo requeridos.

- En las RTA, se obtiene mayor rendimiento, es decir, mayor cantidad de procesos por unidad de tiempo. El rendimiento de una RTA se mide en función de si las tareas se completan dentro de los plazos establecidos, no en la cantidad de procesos realizados por unidad de tiempo. En las RTAs, la prioridad se centra en cumplir con los plazos de ejecución, incluso si eso significa procesar menos tareas.

- Disponer de más procesadores, memoria RAM e interfaces de bus veloces permite transformar el sistema a una RTA. Aunque tener recursos hardware potentes puede mejorar el rendimiento de un sistema, no es suficiente para convertirlo en una RTA, ya que no solo se requieren recursos de hardware adecuados, sino también una planificación y diseño correctos para garantizar el cumplimiento de los plazos.

g. ¿A qué estructura de datos común pertenecen los siguientes miembros y para qué es empleada en los *RTOS*? ¿En código, cómo se puede acceder a estos miembros?

Estos miembros pertenecen a un TCB (*Task Control Block*), que es una estructura usada para mantener información y control sobre cada tarea en el *RTOS*.

```
OS_TICK        TimeQuanta;
OS_CPU_USAGE   CPUUsage;
OS_PRIO        Prio;
CPU_STK_SIZE   StkSize;
```

- **TimeQuanta** es un miembro que representa el valor del quantum de tiempo (tiempo máximo que una tarea puede ejecutarse antes de ser interrumpida y ceder el control a otras) asignado a una tarea.

- **CPUUsage** es un miembro que almacena información sobre el uso de la CPU de una tarea específica. Este valor puede indicar el porcentaje de tiempo de CPU que la tarea ha usado en comparación con el tiempo total de ejecución.

- **Prio** es un miembro que representa la prioridad de una tarea, utilizada para determinar el orden de ejecución de las tareas. Las tareas con prioridad más alta se ejecutan antes que las tareas con una prioridad más baja.

- **StkSize** es un miembro que indica el tamaño de la pila asignado a una tarea. Esta pila es usada por las tareas para almacenar variables locales, registros y otros datos durante su ejecución.

Para acceder a estos miembros, primero es necesario crear una instancia de la estructura TCB y, luego, acceder a los miembros de esta manera:

```
TCB tcb;

tcb.TimeQuanta = 10;
tcb.CPUUsage = 80;
tcb.Prio = 3;
tcb.StkSize = 4096;
```

h. ¿Qué diferencias existen entre la planificaciones *Preemptive* y *Cooperative*, en el ámbito de los *RTOS*? Explica en qué casos seleccionarías una modalidad frente a la otra.

En un *RTOS* cooperativo, las tareas ceden voluntariamente el control al kernel para que se produzca un switch a otra tarea. Este proceso de ceder el control puede ser porque la tarea haya finalizado o mediante una llamada al kernel. Se utiliza cuando:

- Se puede confiar en que las tareas cedan el control en distintos momentos oportunos.
- Las tareas tienen requisitos de comunicación o sincronización intensivos y necesitan interactuar cooperativamente.
- Se desea un mayor grado de determinismo y control en la asignación de tiempo de CPU, ya que las tareas pueden controlar cuándo ceden el control.

En un *RTOS preemptive*, el sistema operativo tiene el control total sobre el cambio de tareas, es decir, el sistema operativo puede interrumpir una tarea en ejecución y cambiar a otra tarea de mayor prioridad en cualquier momento. Se utiliza cuando:

- Se necesita garantizar el cumplimiento de plazos estrictos en tareas críticas.
- Se requiere una respuesta rápida y predecible a eventos en tiempo real.
- Se desea evitar que una tarea monopolice los recursos del sistema y se asegure una distribución equitativa del tiempo de CPU entre las tareas.

i. ¿Cuál es la diferencia básica entre el scheduling de tipo *Timed Cyclic* y *Simple Cyclic*?

En el *Timed Cyclic*, se especifica un tiempo de inicio y un intervalo de tiempo fijo para cada tarea recurrente. Las tareas se programan para ejecutarse periódicamente en distintos momentos.

En el *Simple Cyclic*, las tareas recurrentes se programan para ejecutarse en ciclos consecutivos de tiempo. No se especifican tiempos de inicio precisos ni intervalos de tiempo fijos. Las tareas se asignan en función de orden y la disponibilidad en cada ciclo de planificación.

j. ¿Para qué se utiliza la modalidad *Round-Robin* en un sistema *Preemptive*? Explica cuáles son los requisitos mínimos para trabajar en esta modalidad y qué se consigue al variar el *Time-Quanta* asociado con esta configuración.

La modalidad *Round-Robin* en un sistema *Preemptive* se usa para asignar el tiempo de CPU entre las tareas en un ciclo circular. Cada tarea recibe una asignación de tiempo llamada "Time-Quanta" o *quantum*.

Los requisitos mínimos para trabajar en esta modalidad son:

- El sistema operativo puede interrumpir una tarea en ejecución y cambiar a otra tarea de mayor prioridad.
- Cada tarea debe tener una prioridad asignada para establecer el orden en el que se programarán las tareas.
- Se debe especificar un *Time-Quanta*, que es el tiempo máximo que cada tarea puede ejecutarse antes de que se realice un cambio de contexto y se le dé la oportunidad a la siguiente tarea de la misma prioridad.

En función de cuánto varíe el *Time-Quanta*:

- Si se aumenta, cada tarea tendrá más tiempo de CPU antes de que se realice un cambio de contexto.
- Si se reduce, cada tarea tendrá menos tiempo de CPU antes de que se realice un cambio de contexto.

k. ¿Para qué se utiliza la técnica de '*yielding*' en los *RTOS* y en qué scheduling se aplica? Expón qué método se usa para evitar usar esta técnica.

La técnica de *yielding* se utiliza para permitir que una tarea ceda el control voluntariamente al sistema operativo, es decir, que una tarea decide renunciar a su tiempo de ejecución antes de que se complete su tiempo asignado.

Esta técnica se aplica en la planificación cooperativa, donde las tareas deben cooperar y ceder el control al sistema operativo. En esta planificación, se debe garantizar que todas las tareas tengan la oportunidad de ejecutarse y evitar bloqueos o inanición de tareas.

2. Tenemos un *PIC32* trabajando con una frecuencia de CPU de 24MHz. Por otro lado, se ha configurado un *Timer* hardware de 32-bit, cuya señal de reloj es ½ de la frecuencia de la CPU. A partir de ese *Timer* hemos generado el *System-Tick* del *RTOS*, cargando un valor particular en su registro asociado. Sabiendo que la función *OSTimeDly(20)* provoca una temporización de 40 milisegundos, ¿qué *System-tick* hemos configurado en nuestro sistema, expresado en Hz?

$$\mathrm{F_{Timer}} = \frac{\mathrm{F_{CPU}}}{2} = \frac{24\mathrm{MHz}}{2} = 12\mathrm{MHz}$$

$$\mathrm{T_{Tick}} = \frac{40\mathrm{ms}}{20} = 2\mathrm{ms}$$

$$\mathrm{F_{Tick}} = \frac{1}{2\mathrm{ms}} = 500\mathrm{Hz}$$

3. En *RTOS*, y en particular a sus objetos de kernel:
   a. **¿Qué diferencias existen entre los semáforos binarios y los mutex? Razona en qué situaciones valorarías más el uso de uno frente al otro.**

El semáforo es un mecanismo de señalización, ya que sus operaciones indican si un proceso está adquiriendo o liberando el recurso. En cambio, el mutex es un mecanismo de bloqueo, por el que un proceso necesita bloquear el objeto de mutex y, al liberar un proceso de recurso, debe desbloquear el objeto de mutex.

El valor de la variable semáforo puede modificarse mediante cualquier proceso que adquiera o libere recursos. Mientras que el bloqueo adquirido en el objeto de exclusión solo puede liberarse mediante el proceso que ha adquirido el bloqueo en el objeto de exclusión.

El semáforo es una mejor opción en caso de que haya múltiples instancias de recursos disponibles. Mientras que si únicamente hay un recurso compartido, el mutex es más adecuado.

   b. **¿Por qué razón, en estos sistemas, a la hora de sincronizar una tarea con una ISR o una tarea con otra tarea, se opta por el uso de un Semáforo y no un simple flag declarado como variable global? ¿Cuál es la ventaja que se busca?**

El uso de semáforos en lugar de simples *flags* declarados como variables globales ofrece una mayor flexibilidad, control y capacidad de sincronización en *RTOS*. Los semáforos permiten una gestión más eficiente de los recursos compartidos, evitan bloqueos y esperas activas, y proporcionan una forma estructurada de control de acceso y sincronización entre tareas y/o ISRs.

Los semáforos pueden gestionar prioridades en el acceso a recursos compartidos para evitar situaciones de inanición o bloqueo. Además, proporcionan mecanismos para evitar bloqueos y esperas activas. Por ejemplo, si una tarea intenta adquirir un semáforo que ya está en uso, la tarea puede bloquearse y ponerse en espera hasta que el semáforo se libere.

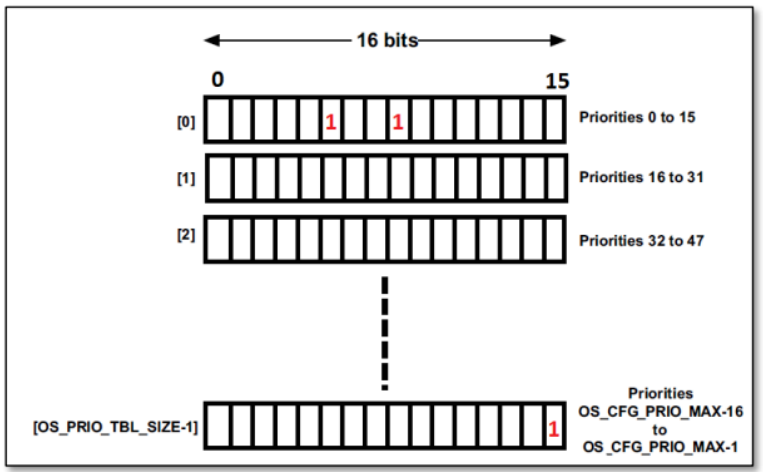
   c. **A nivel funcional, ¿qué similitud tienen los *Message Queues* y los semáforos? Además, ¿Qué ventaja adicional conlleva el uso de *Message Queues* frente a estos?**

Los *Message Queues* y los semáforos son capaces de sincronizar y comunicar entre tareas. Sin embargo, la ventaja adicional del uso de *Message Queues* frente a los semáforos es que pueden transmitir y recibir mensajes.

   d. **Una empresa ha decidido, por primera vez, hacer uso de los *RTOS* y acaba de desarrollar un algoritmo que genera un valor de vital importancia para asegurar el correcto funcionamiento de una aeronave. ¿Por qué recomendarías a esta empresa usar *Message Queues* en lugar de usar una simple variable global, a la hora de transferir dicho valor entre tareas e ISRs?**

Los *Message Queues* proporcionan un mecanismo de sincronización y orden en la transmisión de mensajes y garantizan que los mensajes se transmitan en el orden en que se enviaron. Además, pueden manejar mensajes de diferentes tamaños, lo que permite una mayor flexibilidad en la transmisión de datos. También existe cierta seguridad, ya que solo la tarea o ISR destinataria tiene acceso al mensaje en la cola, evitando el acceso no autorizado o la manipulación de los datos por parte de otras entidades del sistema. Si en el futuro se requiere agregar nuevas funcionalidades, las *Message Queues* ofrecen una mayor escalabilidad, es decir, no hace falta modificar el algoritmo para agregar nuevas tareas para interactuar con la *Message Queue*.

4. Dada la siguiente gráfica:



a. ¿Con qué finalidad emplea el *RTOS* esta estructura?

Esta estructura se utiliza para almacenar las tareas listas para ejecutarse y es propia de un planificador de tareas. Esta lista contiene las tareas que han sido activadas y están en espera de ser asignadas al procesador para su ejecución.

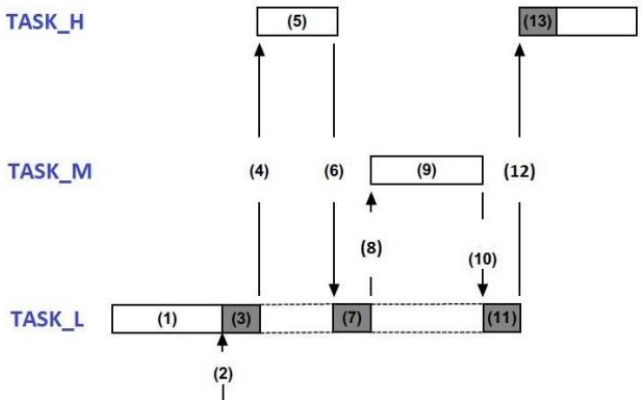
b. En la gráfica anterior vemos varios '1' situados en diferentes Slots. ¿Qué se está indicando al sistema con esta información? (Suponed que el resto de casillas son 'NULL').

Los '1' en los diferentes slots indican que esas tareas están listas para ejecutarse. Mientras que las casillas 'NULL' representan slots que no contienen tareas listas en ese momento. El sistema interpreta esta información para determinar qué tareas deben ser seleccionadas para la ejecución en función de su prioridad.

c. ¿Qué expresa el último '1' (OS_CFG_PRIO_MAX - 1), situado en el array final? ¿Qué peculiaridad tiene respecto al resto de posibles '1' representados en la tabla?

El último '1' en el array se indica que se ha alcanzado la prioridad más alta dentro del rango de prioridades disponibles.

5. En el siguiente gráfico se muestran los estados actuales de 3 tareas gestionadas por un *RTOS* con *scheduler* de tipo *preemptive*. Se pide explicar, en detalle, lo que ocurre en cada punto indicado en la gráfica y contestar a las tres preguntas situadas debajo de la misma. Tened en cuenta las siguientes consideraciones:
   - Task_H y Task_L son las únicas tareas que quieren usar el mismo y único recurso compartido.
   - Las zonas marcadas en gris corresponden a zonas donde se está usando el recurso.



(1): TASK_L toma el control de la CPU y comienza a ejecutarse, ya que es la única tarea que estaba lista para ser ejecutada. Task_H y Task_M tienen prioridades superiores, pero permanecen a la espera de su turno.

(2): TASK_L toma el mutex, porque se comienza a usar el recurso compartido.

(3): TASK_L toma el recurso compartido.

(4): TASK_H se adelanta a TASK_L, ya que se trata de una tarea de mayor prioridad.

(5): Entra en escena la TASK_H que, como tiene mayor prioridad que TASK_L, pues se ejecuta primero.

(6): El mutex se apropia de TASK_L. µC/OS-III eleva la prioridad de TASK_L a la de TASK_H.

(7): Se está usando el recurso compartido en TASK_L.

(8): TASK_L ya ha terminado con el mutex. µC/OS-III reduce la prioridad de TASK_L. Luego, TASK_H se reanuda y ahora posee el mutex.

(9): TASK_H está usando el recurso compartido, gracias a que lo ha adquirido con el mutex.

(10): TASK_H libera el mutex.

(11): Se está usando el recurso compartido en TASK_L.

(12): TASK_H ha finalizado.

(13): TASK_M desea el recurso compartido, pero no lo va a poder obtener hasta que no se adquiera el mutex.

a. En lo referente al fenómeno de *inversión de prioridades* y a la gráfica anterior, ¿se está produciendo o se está evitando dicho fenómeno? Razona la respuesta.

Se está produciendo una inversión de prioridades, puesto que cuando aparece la TASK_L se ejecuta, adquiriendo el mutex en un momento dado y, a continuación, se "invierte la prioridad", es decir, es el turno de TASK_H de realizar lo mismo. Entonces, se invierten las prioridades entre las tres tareas existentes: TASK_H, TASK_M y TASK_L.

b. ¿Qué objeto de kernel se está usando, un semáforo binario o un mutex?

Un mutex, ya que las diferentes tareas quieren acceder a un recurso compartido. Entonces, si el mutex está libre, la tarea lo adquiere y puede acceder al recurso. Si el mutex está siendo utilizado por otra tarea, la tarea que lo intenta adquirir se bloqueará hasta que el mutex esté disponible.

c. ¿Cómo se llama la zona existente entre los puntos (6) y (12)?

Esta zona se llama sección crítica, que es la región donde se accede al recurso compartido y es necesario protegerla para evitar condiciones de carrera. En este caso, esta sección crítica está protegida por un objeto de kernel, mutex.