Actividad Experimental:
Sistemas Operativos de Tiempo Real

Competencias a Desarrollar:

- Integrar tareas en FreeRTOS, asignando prioridades y gestionando su ciclo de vida.
- Utilizar semáforos y mutex para la coordinación y protección de recursos compartidos.
- Intercambiar datos de forma segura y eficiente entre tareas mediante colas y grupos de eventos.
- Conectar eventos de hardware a tareas FreeRTOS de forma segura y reactiva.
- Gestionar memoria dinámicamente, monitoreando el uso y previniendo fugas.
- Utilizar temporizadores de software para ejecutar rutinas periódicas o diferidas.
- Analizar el rendimiento (uso de pila, CPU, heap) para diagnosticar y mejorar la estabilidad y eficiencia del sistema.

Objetivo general

Comprender los fundamentos y la aplicación práctica de los sistemas operativos de tiempo real (RTOS) mediante la implementación de lógicas en FreeRTOS con microcontroladores ESP32, abarcando el diseño, implementación, depuración y optimización de aplicaciones embebidas robustas y eficientes que gestionen la concurrencia y los recursos de hardware de forma efectiva.

E1. Sistema de Gestión Climática para un Invernadero de Cultivo Hidropónico.

En un invernadero de alta tecnología, se requiere un monitoreo constante de la temperatura, humedad e intensidad de luz, y la activación automática de sistemas (ventilación e iluminación) para optimizar el crecimiento de los cultivos y asegurar condiciones estables. El sistema debe responder de manera oportuna a los cambios ambientales.

Se requiere definir prototipos de dos funciones que actuarán como tareas de FreeRTOS:

- void taskLecturaSensores(void *parameter)
- void taskControlInvernadero(void *parameter)

Función taskLecturaSensores:

1. Implementar un bucle infinito (for (;;)) para que la tarea se ejecute continuamente.
2. Simular la lectura de sensores generando valores aleatorios para temperatura, humedad e intensidad de luz. Ejemplo: función random() de Arduino.
3. Imprimir los valores simulados de los tres sensores por el puerto serie (usando Serial.print y Serial.println).
4. Utilizar vTaskDelay(pdMS_TO_TICKS(TIEMPO_EN_MS)); para simular que la tarea toma un tiempo en leer los sensores, por ejemplo, cada 2 segundos.

Función taskControlInvernadero:

1. Implementar un bucle infinito (for (;;)) para que la tarea se ejecute continuamente.
2. Utilizar los valores simulados de temperatura e intensidad de luz (que estarán en variables globales, compartidas con la tarea de lectura) para decidir acciones de control.

Control de Iluminación: Si la intensidad de luz simulada está por debajo de un umbral predefinido (ej. 400lux), imprimir por Serial "Luces ON". De lo contrario, "Luces OFF".

Control de Ventilación: Si la temperatura simulada supera un umbral alto (ej. 28°C), imprimir "Ventilación ON". Si la temperatura baja de un umbral bajo (ej. 22°C), imprimir "Ventilación OFF". Para temperaturas intermedias, indicar que está "Auto".

Utilizar vTaskDelay(pdMS_TO_TICKS(TIEMPO_EN_MS)) para simular que estas tareas se ejecutan periódicamente, por ejemplo, cada 1.5 segundos.

Creación y Configuración de Tareas en setup():

1. Inicializar la comunicación serial Serial.begin(115200).
2. Utilizar xTaskCreatePinnedToCore() para crear ambas tareas:
3. Asignar a taskLecturaSensores una prioridad más alta (ej. 2) y, a taskControlInvernadero, una prioridad más baja (ej. 1).
4. Asignar cada tarea a un núcleo diferente del ESP32 (0 y 1 respectivamente), con un tamaño de pila adecuado (ej. 2048 bytes para ambas).

5. Dejar la función loop() vacía, ya que el scheduler de FreeRTOS tomará el control una vez que las tareas sean creadas.

Actividad Experimental:

1. Verificar que los mensajes de la tarea con mayor prioridad aparecen con mayor frecuencia, especialmente si los vTaskDelay son diferentes o si hay alguna demora simulada en la ejecución.
2. Cambiar las prioridades de las tareas (invertir o definir ambas a la misma prioridad), para analizar cómo el comportamiento en el Monitor Serial cambia.

E2. Sistema de Gestión de Recetas en una Planta de Dosificación Química.

En una planta de procesos químicos, es vital controlar la dosificación de ingredientes con precisión. Una unidad de control debe recibir datos de diversos sensores (pH, nivel de líquido, temperatura de reacción) y, basándose en estos, enviar instrucciones a una unidad de dosificación que active bombas para añadir los componentes químicos necesarios. La comunicación entre estas unidades debe ser fiable y asíncrona para no bloquear el sistema.

Se requiere:

1. Definir una Estructura de Datos que contenga variables para simular las lecturas de sensores: pH, nivel y temperatura.
2. Crear una Cola de FreeRTOS:
   - En la sección global del código o dentro de setup(), se debe declarar un manejador de cola: QueueHandle_t sensor_data_queue;
   - En setup(), inicialicen la cola usando xQueueCreate(). La cola debe ser capaz de almacenar varias estructuras SensorData_t (ej. 5 elementos) y el tamaño de cada elemento debe ser sizeof(SensorData_t).

3. Implementar task_adquisicion_datos (Tarea Emisora). Esta tarea debe simular la lectura de los sensores cada 2 segundos. Dentro de su bucle infinito:
   - Llenar una instancia de SensorData_t con valores aleatorios los sensores.
   - Imprimir por Serial los datos que se están a punto de enviar.
   - Intentar enviar esta estructura a la cola utilizando xQueueSend(), utilizando un tiempo de espera (portTICK_PERIOD_MS * 100) para el caso en que la cola esté llena. Si el envío falla, impriman un mensaje de error.

- Añadir un vTaskDelay() adecuado (ej. 2 segundos) para simular el intervalo de lectura.

4. Implementar task_procesamiento_y_dosificacion (Tarea Receptora): Esta tarea debe esperar por datos en la cola. Dentro de su bucle infinito:
   - Declarar una instancia de SensorData_t para almacenar los datos recibidos.
   - Intentar recibir datos de la cola utilizando xQueueReceive(), utilizando el tiempo de espera (ej. portMAX_DELAY o un pdMS_TO_TICKS(500) para un timeout). Imprimir por Serial los datos recibidos.
   - Implementar una lógica de control simple: Si pH está fuera de un rango (ej. < 6.5 o > 7.5), imprimir "Bomba de Acido ON" o "Bomba de Base ON". De lo contrario, "Bombas OFF".

En el setup(), después de inicializar la cola, se deben de crear ambas tareas usando xTaskCreatePinnedToCore() y, asignarles las prioridades apropiadas. Por ejemplo: task_adquisicion_datos con prioridad 1, task_procesamiento_y_dosificacion con prioridad 2 para que reaccione más rápido a los datos.

Actividad Experimental:

1. Modificar los vTaskDelay() para que la tarea emisora sea más rápida que la receptora. ¿Qué sucede con la cola? ¿Se llena? ¿El emisor se bloquea?
2. Modificar el tamaño de la cola (ej. a 1 elemento). ¿Cómo afecta esto al comportamiento?

E3. Control de Unidades Producidas en una Línea de Ensamblaje.

Dada una línea de producción de componentes electrónicos con dos estaciones de ensamble robotizadas (Robot_A, Robot_B) y una estación de inspección de calidad (Inspector_C), las tres estaciones, necesitan incrementar un contador global de "unidades completadas" que se almacena en el sistema central. Si varias estaciones intentan actualizar el contador al mismo tiempo sin un mecanismo de protección, el valor final del contador podría ser incorrecto, llevando a errores en el inventario o en las métricas de calidad.

Tareas a Realizar:

- Declarar una variable global entera, por ejemplo, int unidades_producidas_total. Esta será el recurso compartido que las tareas intentarán modificar.

- Crear tres funciones de tareas: void taskRobotA(void *parameter), void taskRobotB(void *parameter), y void taskInspectorC(void *parameter). Cada tarea debe contener un bucle infinito dentro del cual se incremente la variable unidades_producidas_total hasta un máximo de 3000, 1000 por cada estación.
- Usar un vTaskDelay() muy pequeño (ej., 1 ms) para permitir que el scheduler cambie entre tareas y aumente la probabilidad de una condición de carrera.
- Al final de la ejecución de cada tarea, imprimir el valor actual del contador, indicando qué tarea lo hizo.

Actividad Experimental:

Parte I:

- En setup(), crear las tres tareas con prioridades iguales (ej. todas con prioridad 1) y asígnarlas a diferentes núcleos si es posible, o a tskNO_AFFINITY para balanceo de carga automático.
- Ejecute y verifique si el valor final de unidades_producidas_total es igual a 3000.

Parte II:

- Declarar una variable de tipo SemaphoreHandle_t globalmente, por ejemplo, SemaphoreHandle_t xMutex_registro.
- En setup(), inicializar el Mutex usando xSemaphoreCreateMutex().
- Dentro de cada tarea, adquirir el Mutex usando xSemaphoreTake(xMutex_registro, portMAX_DELAY). Una vez adquirido el Mutex, realizar la operación de incremento del contador de unidades y, luego liberarlo usando xSemaphoreGive(xMutex_registro). Esto permitirá que otra tarea lo adquiera.
- Ejecute y verifique si el valor final de unidades_producidas_total es igual a 3000.

Tanto para la parte I como la II analizar:

1. Exclusión Mutua: ¿Cómo un Mutex garantiza que solo una tarea a la vez pueda acceder a una sección crítica de código?
2. Sincronización: ¿Cómo el Mutex sincroniza el acceso entre tareas, aunque se ejecuten concurrentemente?

Información adicional:

- Deadlock (Interbloqueo): Si una tarea adquiere un Mutex y nunca lo libera (por ejemplo, debido a un error o a un bucle infinito después de xSemaphoreTake), otras tareas que intenten adquirirlo se bloquearán indefinidamente, lo que se conoce como deadlock.
- Prioridad de Tareas y Mutex: Si una tarea de baja prioridad adquiere un Mutex y luego una tarea de alta prioridad necesita el mismo Mutex, la tarea de alta prioridad quedará bloqueada hasta que la de baja prioridad lo libere. FreeRTOS implementa un mecanismo llamado herencia de prioridades para mitigar este problema, elevando temporalmente la prioridad de la tarea que posee el Mutex a la de la tarea de mayor prioridad que lo espera. Esto se aplica automáticamente con xSemaphoreCreateMutex().
- Contexto de Interrupción: Nunca se debe adquirir (xSemaphoreTake) o liberar (xSemaphoreGive) un Mutex directamente desde una ISR, ya que las ISRs no pueden bloquearse. Para comunicar con ISRs, se usan semáforos binarios o contadores de la forma xSemaphoreGiveFromISR() y xSemaphoreTakeFromISR(), lo cual se verá en el Ejercicio 5.

E4. Sistema de Alerta y Seguridad en un Almacén Automatizado

En un almacén moderno, se utilizan varios sensores para detectar condiciones anómalas. Un sistema de seguridad centralizado monitorea diferentes condiciones: si una puerta de acceso está abierta, si hay movimiento dentro de una zona restringida y si se detecta humo. En función de la combinación de estos eventos deben activarse diferentes respuestas, según se detalla a continuación:

- Puerta abierta: se activa una "luz de advertencia" y se notifica al personal de seguridad.
- Movimiento en zona restringida: se activa un "grabador de video" y se emite una alerta silenciosa.
- Detección de humo: activa la "sirena de evacuación" y se envía una notificación crítica a los bomberos.
- Cualquier combinación de las anteriores: el sistema debe activar la "sirena de evacuación máxima" y enviar una alerta de emergencia de alto nivel.

Procedimiento:

- Definir constantes para los bits que representarán cada evento dentro de un EventGroup.

- Crear un manejador de grupo de eventos global: EventGroupHandle_t xSecurityEventGroup.
- En setup(), inicializar el grupo de eventos usando xEventGroupCreate(). Luego crear las cuatro tareas (taskSensorPuerta, taskSensorMovimiento, taskSensorHumo, y taskSistemaSeguridad) usando xTaskCreatePinnedToCore() y asignar prioridades.
- Implementar tareas de simulación de sensores. Cada tarea debe simular la detección de su respectivo evento periódicamente (cada 3-5 segundos, o usando un botón físico conectado al ESP32 para cada sensor). Cuando el "sensor" detecta su evento, debe establecer el bit correspondiente en el EventGroup utilizando xEventGroupSetBits() e imprimir por Serial que el evento ha sido generado. Incluir un vTaskDelay() adecuado para simular el intervalo entre detecciones.
- Implementar una función de tarea receptora taskSistemaSeguridad. Se debe utilizar una variable EventBits_t uxBits para almacenar los bits de eventos recibidos y xEventGroupWaitBits() para esperar por las combinaciones de eventos definidas en el contexto.
- Imprimir las acciones correspondientes (activar luces, sirena, notificaciones) según las combinaciones de bits recibidas. Consideren que un bit puede estar activado por sí solo o en combinación con otros.

Actividad Experimental:

- Activar los "sensores" individualmente y en combinación (presionando los botones o esperando los temporizadores).
- Observar cómo taskSistemaSeguridad reacciona a los eventos individuales y a las combinaciones.
- Cambiar el parámetro xClearOnExit de xEventGroupWaitBits() de pdTRUE a pdFALSE y, el parámetro xWaitForAllBits, de pdFALSE a pdTRUE. Analizar el comportamiento en cada caso.

Información adicional:

- xClearOnExit vs. xWaitForAllBits: Estos dos parámetros son cruciales para controlar el comportamiento de xEventGroupWaitBits. Experimentar con ellos es la mejor manera de entenderlos.
- pdTRUE en xClearOnExit es común para eventos tipo "señal" que solo necesitan ser procesados una vez.

- pdTRUE en xWaitForAllBits se usa para escenarios donde múltiples condiciones deben ser verdaderas al mismo tiempo (ej. "activar motor si puerta cerrada y sensor de nivel OK y operador presente").
- Bits vs. Datos: Event Groups son excelentes para señalizar eventos y sincronizar el flujo de control. Resulta una opción muy eficiente, ya que la tarea se bloquea y no consume CPU hasta que se activa uno de los bits de interés. Sin embargo, no permiten pasar datos (como el valor de un sensor) junto con el evento, las colas (Queues) son la opción más adecuada para esto.

E5. Botón de Parada de Emergencia para un Brazo Robótico Industrial.

En una celda de manufactura, un brazo robótico realiza operaciones de soldadura de precisión. Es vital que, ante cualquier imprevisto (ej. un operador entra en la zona de seguridad, un fallo inesperado), se pueda activar un botón de "parada de emergencia" que detenga el brazo de forma inmediata y segura. Esta señal debe ser procesada con la máxima prioridad para evitar daños o accidentes. La ISR (Rutina de Servicio de Interrupción) debe ser lo más corta posible para no bloquear otras interrupciones, y una tarea de FreeRTOS de alta prioridad debe encargarse de la lógica de "parada segura" del robot.

Procedimiento:

- Definir un Semáforo Binario mediante el manejador SemaphoreHandle_t xEmergencyStopSemaphore, para señalizar el evento del botón desde la ISR a la tarea.
- Configurar el Pin GPIO para Interrupción, puede utilizarse la resistencia de pull-up interna del ESP32 mediante INPUT_PULLUP.
- Implementar la Rutina de Servicio de Interrupción (ISR) con el prototipo void IRAM_ATTR isrEmergencyButton(). La única acción que debe realizar la ISR es activar el semáforo binario para señalizar a la tarea: xSemaphoreGiveFromISR(xEmergencyStopSemaphore, &xHigherPriorityTaskWoken).
- Declarar la variable BaseType_t xHigherPriorityTaskWoken = pdFALSE; localmente en la ISR y pásenla por referencia. Después de xSemaphoreGiveFromISR(), llamar a portYIELD_FROM_ISR(xHigherPriorityTaskWoken) para forzar un cambio de contexto si la tarea de mayor prioridad que espera el semáforo se ha desbloqueado.
- Implementar la Tarea Receptora de Semáforo mediante la función: void taskManejoEmergencia(void *parameter). Imprimir un mensaje indicando que la tarea está esperando el botón de emergencia.

- Esperar indefinidamente el semáforo binario usando xSemaphoreTake(xEmergencyStopSemaphore, portMAX_DELAY).
- Una vez que la tarea adquiere el semáforo (lo que significa que el botón fue presionado), imprimir el mensaje "¡PARADA DE EMERGENCIA ACTIVADA!".

*IRAM_ATTR es importante para asegurar que la ISR se ejecute desde la RAM interna, lo que mejora la velocidad y confiabilidad. Este patrón de ISR -> Semáforo -> Tarea de Alta Prioridad es fundamental para sistemas de tiempo real duro, donde las respuestas deben ser predecibles y rápidas.

Actividad Experimental:

- Implementar Serial.print() o delay() dentro de la ISR. Explicar.
- Cambiar la prioridad de taskManejoEmergencia a una más baja que otra tarea activa y observar si la respuesta se vuelve menos "instantánea".

E6. Dispositivo IoT de Monitoreo de Vibraciones para Mantenimiento Predictivo.

Suponer un dispositivo IoT instalado en proceso industrial para monitorear vibraciones en diferentes puntos de la tubería. Los datos recolectados pueden variar en tamaño, y es necesario asignarle memoria de forma eficiente. Además, el sistema necesita generar un "reporte de estado" cada cierto tiempo y activar una "alarma de calibración" solo una vez, después de un período inicial.

Procedimiento para manejo de memoria:

- Crear una función de tarea void taskGeneradorDatosVibracion(void *parameter), para generar un tamaño aleatorio para un "paquete de datos de vibración" (ej. entre 50 y 200 bytes).
- Asignar memoria dinámicamente para este paquete utilizando pvPortMalloc(size_of_packet).
- Verificar si la asignación fue exitosa (si pvPortMalloc no retorna NULL). Si no, imprimir un error de memoria.
- Simular el llenado de este paquete e informar memoria libre actual después de la asignación usando xPortGetFreeHeapSize().

- Importante: Después de simular el procesamiento, liberar la memoria asignada utilizando vPortFree(puntero_al_paquete). Imprimir la cantidad de memoria libre después de la liberación.

Procedimiento para manejo temporizadores:

- Crear dos funciones de callbacks para los temporizadores, con el prototipo void vPeriodicReportCallback(TimerHandle_t xTimer); y void vOneShotCalibrationCallback(TimerHandle_t xTimer).
  - vPeriodicReportCallback: Simulará la generación de un reporte periódico. Dentro de ella, imprimiendo "REPORTE PERIODICO".
  - vOneShotCalibrationCallback: Simulará una alarma de calibración de una sola vez. Dentro de ella, imprimir "ALARMA CALIBRACION".
- Declarar dos manejadores de temporizador globales: TimerHandle_t xPeriodicReportTimer y xOneShotCalibrationTimer.
- En setup(), crear los temporizadores usando xTimerCreate() e iniciarlos con xTimerStart().
  - xPeriodicReportTimer: Debe ser un temporizador periódico.
  - xOneShotCalibrationTimer: Debe ser un temporizador de un solo disparo.

Actividad Experimental:

- Observar cómo la cantidad de memoria disponible cambia después de cada asignación y liberación.
- Comentar la línea vPortFree() en taskGeneradorDatosVibracion y observar la memoria disponible.
- Verificar los tiempos de disparo de los temporizadores.

Información adicional:

- Si no se libera la memoria (vPortFree), el heap disponible se reducirá progresivamente. Además, si se asigna y liberas bloques de diferentes tamaños continuamente, la memoria puede fragmentarse (tener muchos pequeños "agujeros" libres que son demasiado pequeños para satisfacer una nueva solicitud de asignación grande), lo que eventualmente puede llevar a fallos en pvPortMalloc incluso si hay suficiente memoria libre total. La observación en el Monitor Serial del xPortGetFreeHeapSize() es clave para entender esto.

- Tarea de Servicio de Temporizador: Todos los temporizadores de software de FreeRTOS son gestionados por una única tarea interna de FreeRTOS (la Timer Service Task). Esto significa que las funciones de callback de los temporizadores se ejecutan en secuencia por esta tarea. Si un callback es demasiado largo, retrasará la ejecución de otros callbacks y la reconfiguración de otros temporizadores.
- Temporizadores vs. vTaskDelay(): Los temporizadores de software son ideales para ejecutar lógica periódica o programada sin bloquear la tarea principal que los crea. Una tarea puede crear varios temporizadores y luego dedicarse a otras cosas, mientras los temporizadores se disparan de forma asíncrona. Usar vTaskDelay() dentro de una tarea solo bloquea esa tarea específica.