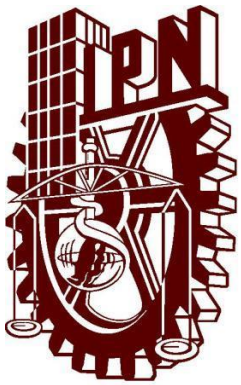


INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA EN INGENIERÍA Y TECNOLOGÍAS AVANZADAS

SISTEMAS OPERATIVOS EN TIEMPO REAL

CHONG PEREZ JORGE TEMUDSCHIN

GONZALEZ CORTES IVAN

VERGARA MUÑOZ LUIS FRANCISCO

VUDOYRA GRACIAN AARON

Proyecto

PROFESOR: MAZA CASAS LAMBERTO

Introducción

Un sistema operativo en tiempo real (Real Time Operating System, RTOS) es aquel que pretende asegurar un patrón determinado(predecible) de ejecución de las tareas. Es particularmente interesante para sistemas embebidos con restricciones temporales de tiempo real. Es decir, si tenemos tres tareas de las cuales una tiene que ejecutarse necesariamente cada 3 milisegundos, el sistema operativo provee servicios para poder llevarlo a cabo.

FreeRTOS es un tipo RTOS diseñado para ser lo suficientemente pequeño para correr en un microcontrolador, aunque no está limitado a este tipo de hardware. Las características que posee son:

Multitarea

Semáforos y buzones de correo-

Planificador cooperativo, Round Robin y basado en prioridades.

Bajo consumo de memoria.

Temporizador por software.

Las tareas son la mínima unidad de ejecución del sistema. En este tipo de sistema operativo, las tareas tienen su propio contexto de ejecución, independientes del resto y del Kernel. El contexto de ejecución es un espacio en memoria (normalmente memoria RAM) en la cuál se almacena el estado de la tarea. Dicho estado se representa con:

Siguiente instrucción a ejecutar.

Variables internas del proceso.

Valor de los registros del procesador justo antes de abandonar la CPU.

El proceso de ejecutar una tarea a otra se llama cambio de contexto y consiste en almacenar contexto del proceso en el stack (zona de memoria reservada) de la tarea. Se ejecuta el planificador, que decide qué tarea ha de ejecutarse posteriormente, y por último se restaura el contexto en el procesador.

Objetivos

Implementar el sistema operativo en tiempo real FreeRTOS en un microcontrolador, avr, pic, o tarjeta de desarrollo.

Implementar un planificador de tareas por prioridades en el sistema operativo FreeRTOS.

Las tareas a realizar son la lectura de un canal del adc y un pin digital, y mandarlos por el puerto serial.

Desarrollo

Para crear una tarea es necesario definir ciertos parámetro dentro de una función

1. Dirección de la función de la tarea.
2. Nombre de la tarea.
3. Tamaño de la tarea.
4. Parámetros de entrada de la tarea.
5. Prioridad.
6. Handler para la tarea.

Cuando termine el setup del arduino, el Kernel toma el control y comienza a ejecutar las tareas en función de su prioridad. En este caso tenemos una tarea que lee valores del adc y las manda por el puerto serie y otra, que lee el estado del pin digital y lo manda por el puerto serie.

Banderas o semáforos

Un semáforo es un mecanismo de sincronización entre tareas, que permite controlar el acceso a un único recurso por parte de diferentes tareas.

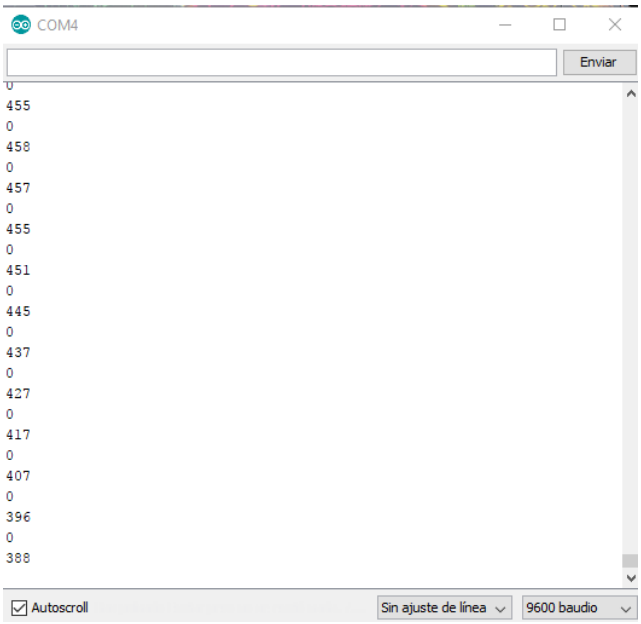
La tarea que vaya a usar un recurso, lo notifica al semáforo. Si otra tarea va a usar el mismo recurso, el semáforo le indica que no está disponible y no le dejaría usarlo. Cuando la tarea originaria termina con el recurso, le vuelve a notificar al semáforo, que dicho recurso queda libre, así la segunda tarea puede conseguir el control del recurso compartido. A este tipo de semáforos, se les denomina semáforos binarios.

Código:

```
#include <Arduino_FreeRTOS.h> // incluimos la librería de FreeRTOS al programa
#include <semphr.h>  // esta librería maneja las banderas o semáforos en arduino.

// se declara un semáforo binario para el manejo de la comunicación por el puerto serial
//para que solo una tarea a la vez tenga acceso al Puerto serial.
SemaphoreHandle_t xSerialSemaphore;

// se definen las 2 tareas a ejecutar por el planificador.
void TaskDigitalRead( void *pvParameters );
void TaskAnalogRead( void *pvParameters );

void setup() {

  // se inicializa la comunicación serial a 9600 baudios:
  Serial.begin(9600);
```

```
  while (!Serial) {
    ; // espera a que se habrá el Puerto serial
  }

  if ( xSerialSemaphore == NULL )  // se confirma que el semáforo no haya sido creado
  {
    xSerialSemaphore = xSemaphoreCreateMutex();  // se crea el semáforo que se encargara de manejar la comunicacion del puerto serial
    if ( ( xSerialSemaphore ) != NULL )
      xSemaphoreGive( ( xSerialSemaphore ) );  // se habilita el Puerto serial para ser utilizado por la función XSemaphoregive.
  }

  // se definen las tareas a realizer por el so.
  xTaskCreate(
    TaskDigitalRead
    ,  (const portCHAR *)"DigitalRead"  // se nombra la tarea
    ,  128  // tamaño de pila
    ,  NULL
    ,  2  //prioridad de la tarea .
    ,  NULL );

  xTaskCreate(
    TaskAnalogRead
    ,  (const portCHAR *) "AnalogRead" // se nombra la tarea
    ,  128  // tamaño de pila
    ,  NULL
    ,  1  // priordad de la tarea
    ,  NULL );
}

void loop()
{
  // vacio.
}
// programacion de las tareas

void TaskDigitalRead( void *pvParameters __attribute__((unused)) )
{

  uint8_t pushButton = 2;

  pinMode(pushButton, INPUT);

  for (;;)
  {
    int buttonState = digitalRead(pushButton);

    // lee el semáforo si puede usar el Puerto serie.
    // si no está disponible, espera 5 siclos del planificador y vuelve a revisar si esta libre.
```

```
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) == pdTRUE )
    {
    // Pudimos tomar el semáforo y ahora podemos acceder al recurso compartido.
    // Queremos tener el puerto serie solo para nosotros, ya que lleva algo de tiempo imprimir,
    // entonces no queremos que sea robado durante la mitad de una conversión
      Serial.println(buttonState);

      xSemaphoreGive( xSerialSemaphore ); //dejamos libre el semáforo para otras tareas.
    }

    vTaskDelay(1);  // una pequeña espera para no saturar el mcu.
  }
}

void TaskAnalogRead( void *pvParameters __attribute__((unused)) )

  for (;;)
  {
    int sensorValue = analogRead(A0);

    // lee el semáforo si puede usar el Puerto serie.
    // si no está disponible, espera 5 siclos del planificador y vuelve a revisar si esta libre.
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) == pdTRUE )
    {
      // Pudimos tomar el semáforo y ahora podemos acceder al recurso compartido.
      // Queremos tener el puerto serie solo para nosotros, ya que lleva algo de tiempo imprimir,
      // entonces no queremos que sea robado durante la mitad de una conversión
      Serial.println(sensorValue);

      xSemaphoreGive( xSerialSemaphore ); //dejamos libre el semáforo para otras tareas.
    }

    vTaskDelay(1);
  }
}
```



Conclusiones

Un RTOS permite organizar tareas de manera lógica y sencilla, asignando una prioridad a cada uno de ellos. Tiempo real significa que el Sistema Operativo garantiza una latencia reducida para los eventos e interrupciones, dado un diseño adecuado del software y las tareas.

Ventajas y desventajas

Ventajas Reducción del tiempo de desarrollo Refuerza conceptos de modularizar

Provee de métodos seguros de comunicación entre tareas

Asegura la independencia entre tareas

Facilita la extensibilidad y verificación del diseño

Desventajas

Desarrollar software para un RTOS requiere de experiencia y planeamiento cuidadoso:

Controlar acceso a recursos utilizados por más de una tarea Requieren más memoria RAM que un sistema sin RTOS

Bibliografía

https://create.arduino.cc/projecthub/feilipu/using-freertos-multi-tasking-in-arduino-ebc3cc

https://ingnotes.wordpress.com/2016/08/04/arduino-y-freertos/

https://feilipu.me/2015/11/24/arduino_freertos/

https://www.freertos.org/FreeRTOS_Features.html