Cátedra de Sistemas Operativos II

Trabajo Práctico N° IV

D'Andrea, F. David

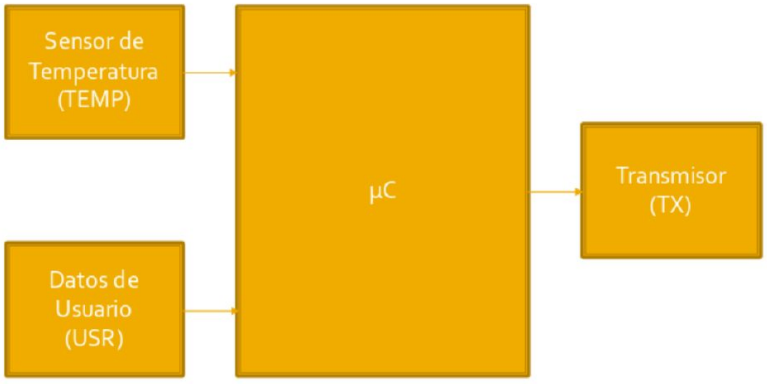
25 de Febrero de 2020

Índice

Introducción

En este trabajo práctico se aborda la configuración y utilización de un sistema operativo de tiempo real en un sistema embebido y la implementación de 3 tareas simples que son ejecutadas y planificadas por el scheduler del SO según sean las prioridades que el usuario le asigne a cada tarea.

En la figura siguiente se esboza un sencillo diagrama de la arquitectura del sistema que se implementara.



Además, una vez implementadas las tareas que se ejecutarán en el SO, se hará uso de la herramienta Percepio Tracealizer para recolectar datos de eventos del sistema operativo y a partir de los mismos analizar las gráficas que la herramienta provee.

Propósito

Lograr configurar y utilizar el sistema operativo de tiempo real FreeRTOS en un sistema embebido, implementar algunas tareas específicas y analizar el comportamiento del sistema con Percepio Tracealizer.

Definiciones, Acrónimos y Abreviaturas

- MCU: microcontrolador

- SO: Sistema Operativo
- IDE: Integrated Development Environment
- UART: Universal Synchronous Asynchronous Receiver Transmitter

Referencias

1. https://docs.aws.amazon.com/es_es/freertos-kernel/latest/dg/freertos-kernel-dg.pdf
2. https://www.freertos.org/
3. https://percepio.com/gettingstarted-freertos/
4. https://www.nxp.com/design/software/development-software/

Descripción General

El presente informe está compuesto por 6 secciones principales:

- **Introducción**
  Breve comentario sobre lo que se propone el en trabajo práctico y como se lo abordará.
- **Descripción General**
  Aquí se exhibe un vistazo de lo que es el desarrollo del trabajo.
- **Requisitos específicos**
  En esta sección se detallan los requisitos específicos que debe cumplir el sistema.
- **Implementación y resultados**
  Se muestran detalles de los componentes del sistema y del análisis realizado con la herramienta Tracealizer.
- **Conclusión**
  Comentario del autor sobre los objetivos planteados y resultados obtenidos
- **Apéndice**
  Breves instructivos sobre la descarga, configuracion e instalacion de las herramientas utilizadas.

Perspectiva del Producto

El producto en este caso será un código que se ejecuta en el RTOS que simula el ingreso de datos por teclado de manera a-periódica y aleatoria y el ingreso de datos simulando el polling a un sensor. Se simulará también la transmisión de los datos ingresados

Funciones del Producto

- Se implementa una tarea que simula la lectura de un sensor de temperatura.
- Se implementa una tarea que simula la carga de strings por teclado.
- Se implementa una tarea que simula el envío de datos por UART.

Características de los Usuarios

El producto se destina a usuarios con conocimientos básicos del manejo de sistemas operativos de tiempo real y sistemas embebidos.

Restricciones

- Se debe instalar y configurar un sistema FreeRTOS en un sistema embebido con un procesador de propósito general como un cortex M3.

Suposiciones y Dependencias

- El sistema FreeRTOS corre sobre la placa NXP LPC1769 y se implementan las tareas utilizado la IDE MCUXpresso, por lo tanto es preciso tener instalada y configurada dicha herramienta.
- Se deberá tener instalado en el sistema host la herramienta Percepio Tracelizer (consultar Apéndice).
- Se deberá descargar y configurar un complemento en la IDE MCUXpresso que vincule las trazas capturadas con la interfaz gráfica de Tracealizer (consultar Apéndice)

Requisitos Futuros

En caso de que en el futuro las simulaciones del ingreso de datos sea real y no simulada, serán necesarios:

- Un adaptador UART/USB (para el envío de datos)
- un sensor de temperatura físico, por ejemplo un integrado LM35.
- Una placa de expansión LPX Base Board (para agregar un teclado)

Requisitos Específicos

En este apartado se toma como base la consigna del trabajo:

1. Se instale y configure FreeRTOS en el sistema embebido seleccionado.

2. Crear un programa con dos tareas simples (productor/consumidor) y realizar un análisis completo del Sistema con Tracealyzer (tiempos de ejecución, memoria).

3. Diseñe e implemente una aplicación que posea dos productores y un consumidor. El primero de los productores es una tarea que genera strings de caracteres de longitud variable (ingreso de comandos por teclado). La segunda tarea, es un valor numérico de longitud fija, proveniente del sensor de temperatura del embebido. También que la primer tarea es aperiódica y la segunda periódica definida por el diseñador.

Interfaces Externas

En este caso dada la implementación (simulación del ingreso y envío de datos) la única interfaz externa que se observa es la conexión mediante USB entre la placa LPC1769 y el equipo host que correrá Tracealizer.

Requisitos de Rendimiento

En los sistemas de tiempo real, la característica principal es que las acciones que se ejecutan demoran una cierta cantidad de tiempo y deben ser ejecutadas dentro de un plazo determinado (comportamiento determinístico).

En estos sistemas se realizan tareas específicas y no se espera del sistema que este pueda hacer la mayor cantidad de tareas posibles en el menor tiempo posible como en otros sistemas de cómputo. Aquí lo que interesa es que las tareas se realicen cumpliendo los requisitos de tiempo sobre los que fueron diseñadas.

En este trabajo esos requisitos son:

1. La lectura del sensor se realizará de manera periódica cada 100[ms].
2. El ingreso de strings por teclado debe ser de máxima prioridad.

Restricciones de Diseño

sobre el sistema embebido escogido de deberá correr un SO FreeRTOS y las tareas requeridas deberán ser implementadas en lenguaje C utilizando la API del SO.

El sistema deberá ser modelado de la manera en que se presentó el diagrama de arquitectura (consultar el diagrama de arquitectura de la introducción).

Atributos del Sistema

El sistema de cómputo utilizado (NXP LPC1769) cuenta con las siguientes características:

- Procesador Arm ® Cortex-M3, funcionando a frecuencias de hasta 120 MHz
- Arm Cortex-M3 Controlador de interrupción vectorial anidado incorporado (NVIC)
- Memoria de programación flash en chip de hasta 512 kB
- Hasta 64 kB de SRAM en chip
- Programación en el sistema (ISP) y Programación en la aplicación (IAP)
- Controlador DMA de uso general de ocho canales (GPDMA)
- Ethernet MAC con interfaz RMII y controlador DMA dedicado
- Dispositivo de velocidad máxima USB 2.0 / Host / controlador OTG
- Cuatro UART con generación de velocidad de transmisión fraccional, FIFO interno y compatibilidad con DMA
- Controlador CAN 2.0B con dos canales
- Controlador SPI con comunicación síncrona, serie, dúplex completo
- Dos controladores SSP con capacidades FIFO y multiprotocolo
- Tres interfaces de bus I2C mejoradas
- Interfaz I2S (sonido Inter-IC)

Para ver el detalle completo de todas las características de la placa de desarrollo remitirse a:

https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1700-cortex-m3/512kb-flash-64kb-sram-ethernet-usb-lqfp100-package:LPC1769FBD100

Implementación y Resultados

Principales funciones de la API de freeRTOS utilizadas:

La API se puede consultar en el site oficial de FreeRTOS en el link:
https://www.freertos.org/a00106.html

Se deja una breve síntesis de las funciones utilizadas:
Queues
xQueueHandle xQueueCreate ()

```
QueueHandle_t xQueueCreate (UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize);
```

Crea una nueva cola y devuelve un identificador (handle) mediante el cual se puede hacer referencia a la cola.
Cada cola requiere RAM que se utiliza para mantener el estado de la cola y para contener los elementos contenidos en la cola (el área de almacenamiento de la cola). Si se crea una cola usando xQueueCreate(), la RAM requerida se asigna automáticamente desde el heap de FreeRTOS.

Parámetros:

uxQueueLength El número máximo de elementos que la cola puede contener en cualquier momento.
uxItemSize El tamaño, en bytes, requerido para mantener cada elemento en la cola.

Los elementos se ponen en cola por copia, no por referencia, por lo que este es el número de bytes que se copiarán para cada elemento en cola. Cada artículo en la cola debe ser del mismo tamaño.

return:

Si la cola se crea correctamente, se devuelve un identificador a la cola creada. Si no se pudo asignar la memoria requerida para crear la cola, se devuelve NULL.

[tipo] **xTaskCreate()**
Las tareas se crean utilizando la función **xTaskCreate()**.
Cada tarea se crea y coloca en estado “Listo”. La memoria RAM necesaria para la tarea es asignada automáticamente por la función ocupando parte de la memoria que queda libre en el heap del RTOS.

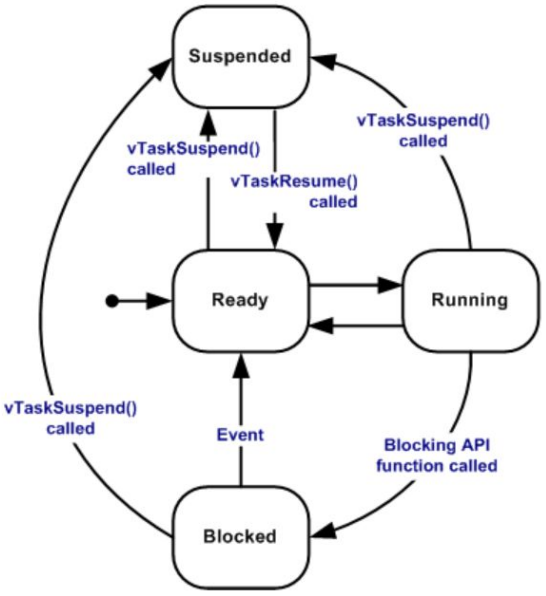
los parámetros de la función xCreateTask() son:

- **pvTaskCode:** es un puntero a una tarea que corre infinitamente. Estas tareas pueden ser eliminadas cuando terminan su ejecución.
- **pcName:** nombre arbitrario que se le asigna a la tarea. No afecta para nada la ejecución de la misma. Es para que el usuario la identifique facilmente nada más.
- **usStackDepth:** es la cantidad de palabras que puede tener el stack de la tarea. Si una palabra equivale a 4 bytes y el parámetro se setea en 100, entonces el stack tendra un tamano de 400 bytes.
- **pvParameters:** las tareas pueden recibir un parámetro que es un puntero a void (void*)
- **uxPriority:** representa la prioridad de la tarea, donde 0 (cero) es la menor prioridad y (configMAX_PRIORITIES – 1) es la mayor. La constante configMAX_PRIORITIES es definida por el usuario, pero si bien la cota superior es “ilimitada” es recomendable usar el menor valor posible para evitar desperdiciar RAM.
- **pxCreatedTask():** se trata de un identificador de la tarea que se utiliza para referenciar la tarea al momento de eliminar la tarea o cambiarle la prioridad, este campo acepta el valor NULL.

Las tareas pueden crearse al principio de programa en la función main.c o bien pueden crearse durante la ejecución de una tarea en sí misma, sin embargo, si el programa está iniciado, la/s tareas creadas comenzarán a ejecutarse recién cuando se invoque la función que inicia el scheduler: “**vTaskStartScheduler()**”.

Estados en los que puede estar una tarea en FreeRTOS

puede consultarse información detallada en el sitio oficial de FreeRTOS:
https://www.freertos.org/RTOS-task-states.html



Transiciones de estado de tarea válidas

Implementación de un productor (sensor) y un consumidor (transmisor)

Dentro del código principal de sistema (main.c) se crea una tarea denominada “lectura_sensor” que es el productor, y otra tarea llamada “transmisor” que es el consumidor

se muestran a continuación algunas capturas de 2 gráficas de tracealizer que muestran de manera muy clara la ocupación de CPU y la planificación de ejecución según las prioridades de las tareas.

Figura 1 - CPU Load

En la **Figura 1**, se observa como el procesador apenas ocupa poco menos del 20% de su capacidad como máximo. Este hecho está en realidad manipulado para que sea computacionalmente costoso agregando un loop for de varias miles de iteraciones. Si no se colocara este bucle, la simplicidad de la tarea de escritura en el buffer ocuparía alrededor del 1% del procesador.

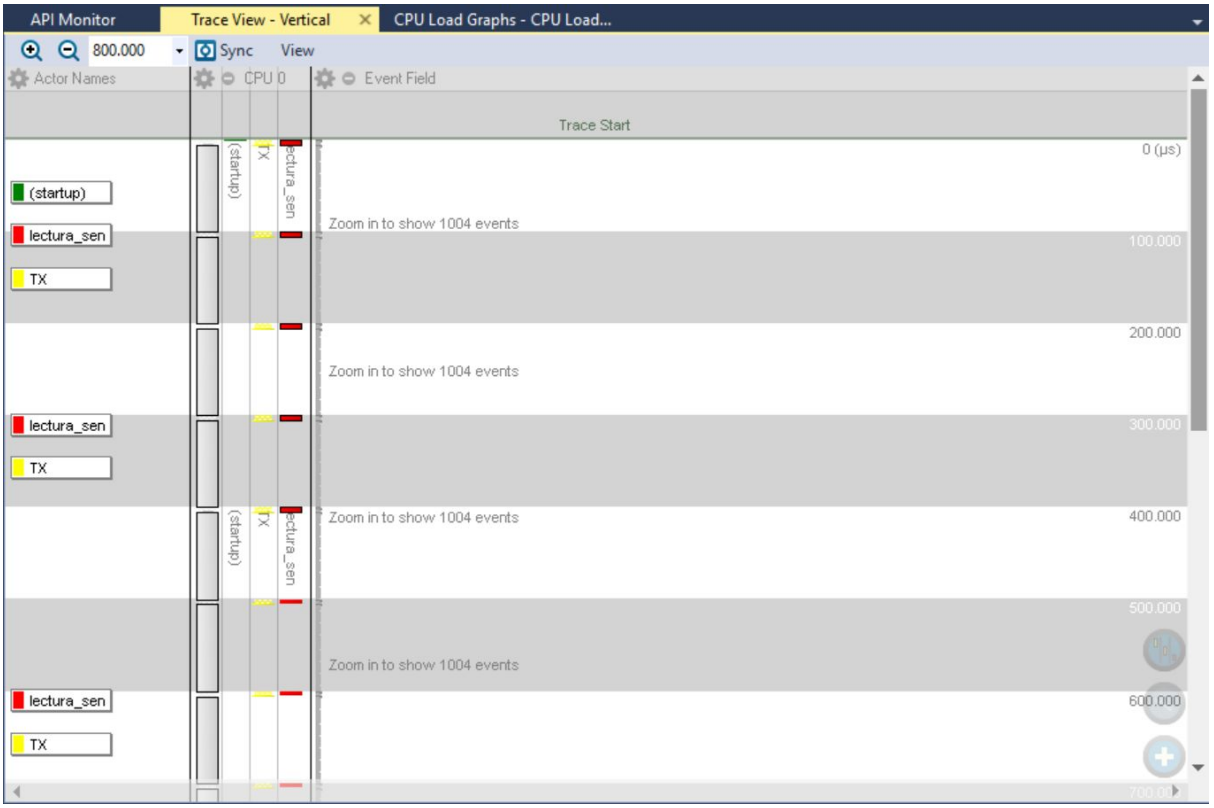


Figura 2 - Trace View Vertical (periodicidad de lectura del sensor)

En la **Figura 2** se observa como en este caso, se utilizó la función vTaskDelayUntil() para lograr un efecto de periodicidad en lo que es la simulación de la lectura del sensor. Cabe aclarar que en la API de FreeRTOS se recomienda hacer este tipo de implementación para las tareas periódicas.

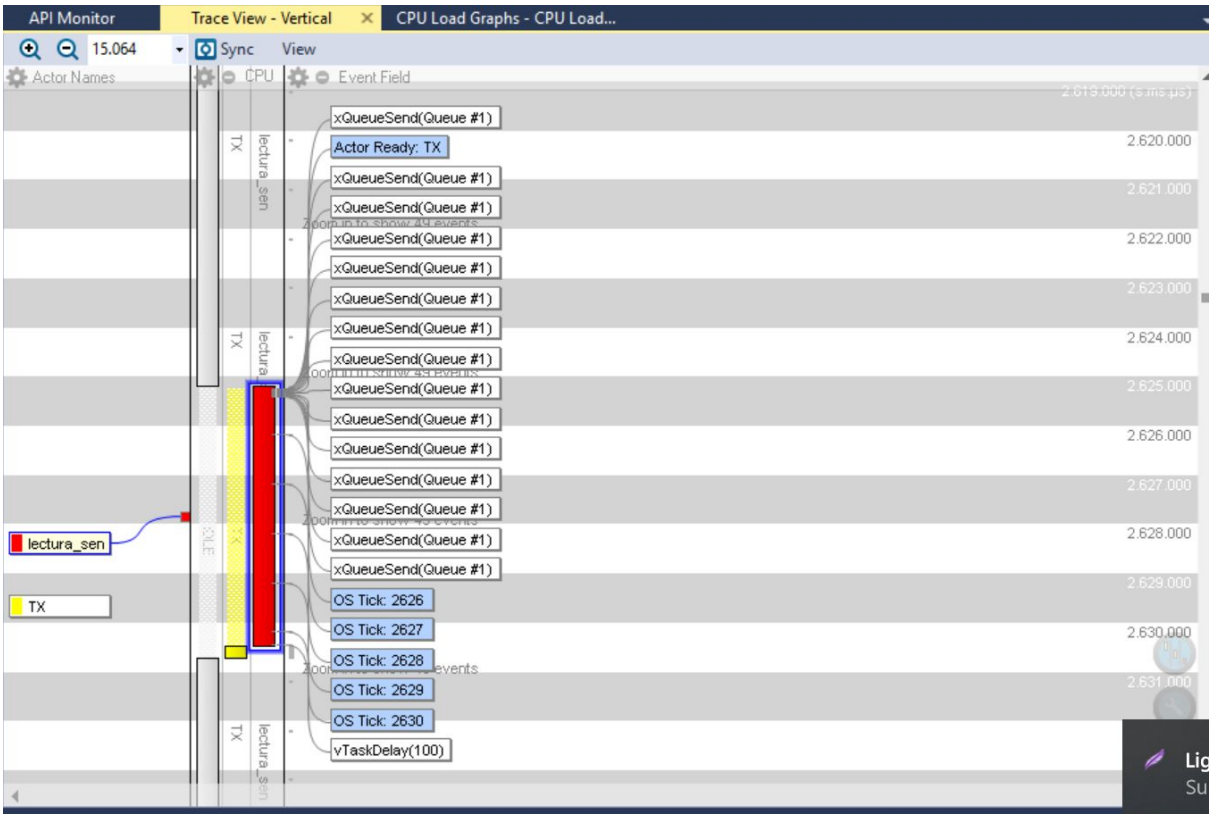


Figura 3 - Trace View Vertical (ejecución de tareas)

En la **Figura 3** se observa claramente como el procesador está ocioso (la barra vertical gris es la “tarea IDLE”) hasta que se produce la primera escritura del sensor sobre la cola de datos (xQueueHandle cola_de_datos). Dada la naturaleza de la implementación, la tarea coloca uno por uno los caracteres del string que se imprime por consola y una vez ejecutada esta tarea de mayor prioridad, el scheduler pasa a ejecutar la tarea “TX” que irá consumiendo uno por uno los caracteres almacenados. Puede notarse en la imagen como la tarea TX pasa al estado “listo” cuando ya hay un elemento en el buffer, pero no se ejecuta hasta que termine la tarea “lectura_sensor” que tiene mayor prioridad.

Implementación de dos productores (sensor y teclado) y un consumidor (transmisor)

Con el propósito de observar la expulsión de una tarea de menor prioridad cuando llega a estado “listo” otra tarea de mayor prioridad que la primera, en esta implementación se

volvieron a manipular los tiempos de ejecución de las tareas “agrandandolas” de manera de aumentar la probabilidad de que se solapen.
Las tareas que se implementaron en este caso, son 3: dos productores (sensor y teclado) y un consumidor (el transmisor). Tracealizer ofrece una vista que deja esto claro de manera gráfica.

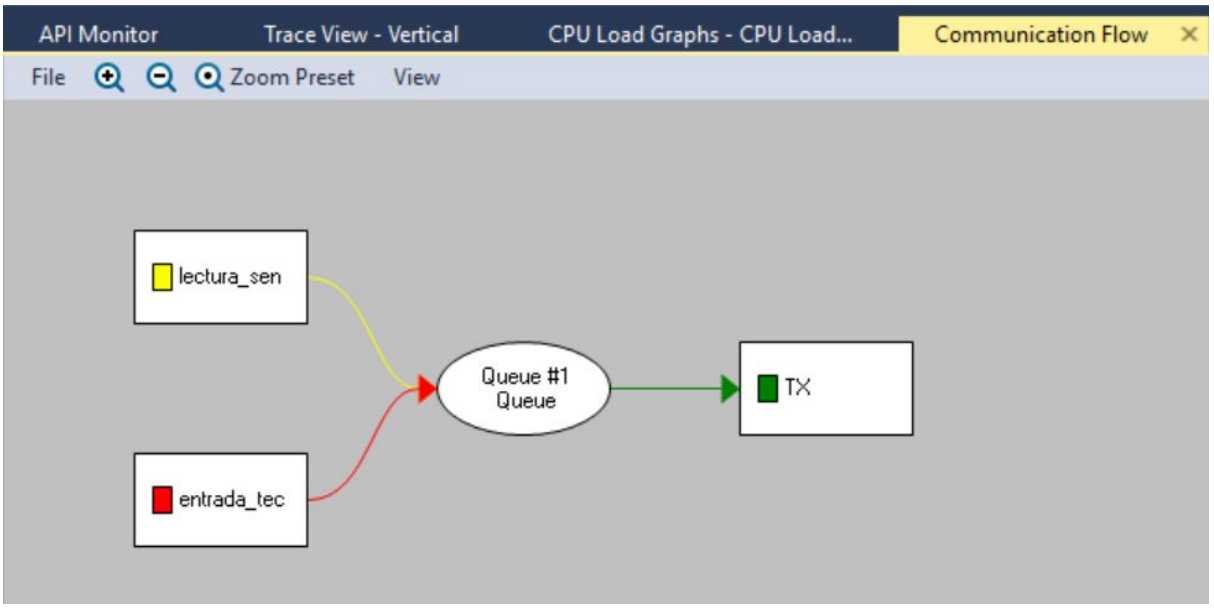


Figura 4 - Communication flow

En la figura 4, se observa que en la cola de elementos, los mismos son introducidos por el sensor y el teclado y son consumidos por la tarea transmisor.

En este simple caso que se propone se presentan muchas situaciones que merecen análisis. Las prioridades que se les asignaron a las tareas son:

- entrada teclado: prioridad=3
- lectura_sensor: prioridad=2
- transmisor: prioridad=1

Estas prioridades son arbitrarias, manipuladas para observar el comportamiento del planificador y como las tareas son cambiadas de estado según su prioridad y la disponibilidad de los recursos del sistema.

En las figuras que se muestran a continuación, se describen algunos de los escenarios que pueden presentarse:

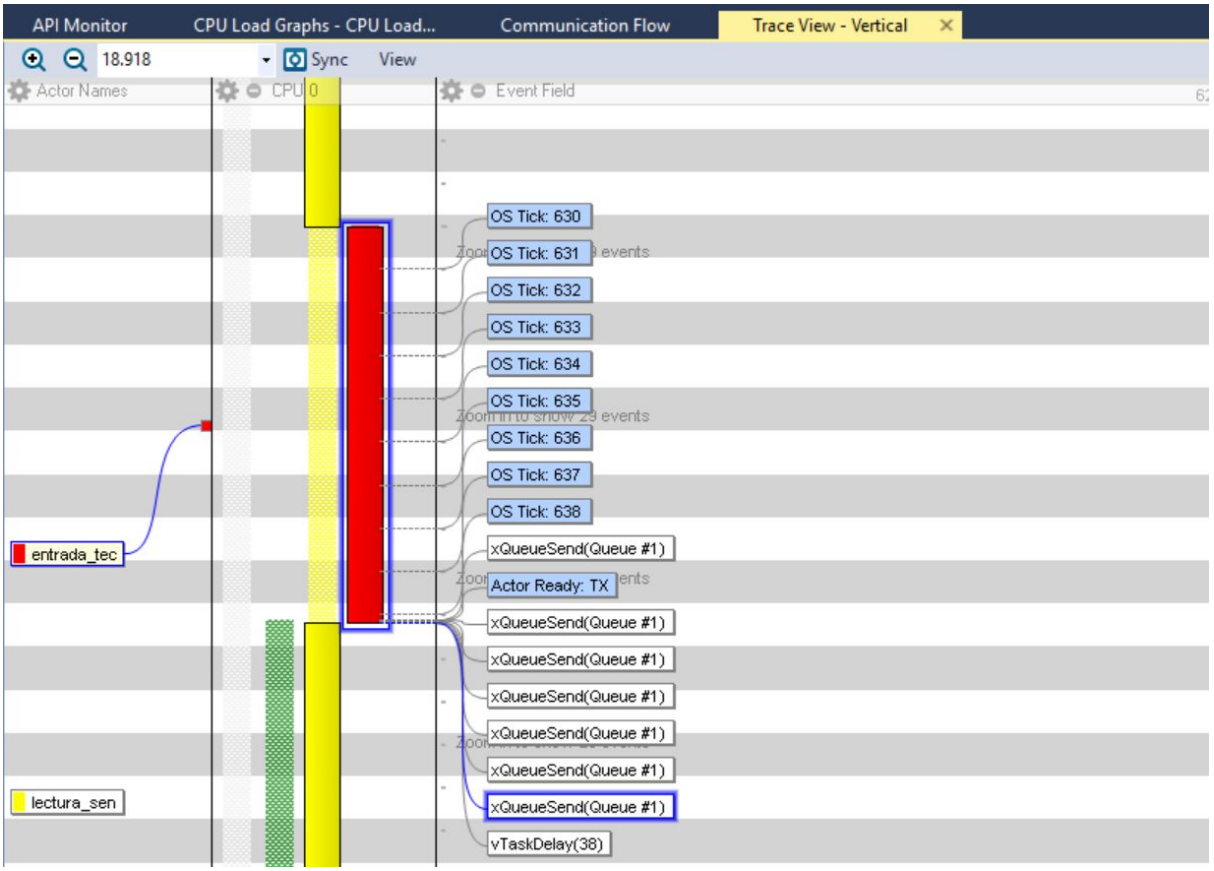


Figura 5 - Trace View Vertical (expulsión por prioridad)

En la Figura 5, se observa que la tarea “lectura_sensor” está en estado ejecutando la tarea de máxima prioridad “entrda_teclado” pasa a estado “listo”. Cuando se ejecuta el scheduler, se determina que la tarea de máxima prioridad debe pasar al estado “ejecutando” y expulsa a la tarea “lectura_sensor”. Dada la implementación de la tarea “entrada_teclado”, que simula un usuario ingresando un string sabemos que la tarea es a-periodica y de longitud no determinista. Una vez que la tarea terminó de ejecutarse y se bloquea, entonces la tarea “lectura_sensor” vuelve a ser la tarea en estado “listo” con prioridad más alta y pasa al estado ejecutando.
Puede verse en esta Figura también, que ni bien la tarea “entrada_teclado” coloca un dato en la cola, la tarea “transmisor” pasa al estado “listo” ya que existe un dato para enviar, sin embargo la tarea de mayor prioridad en este instante es “lectura_sensor” que sigue ejecutando.

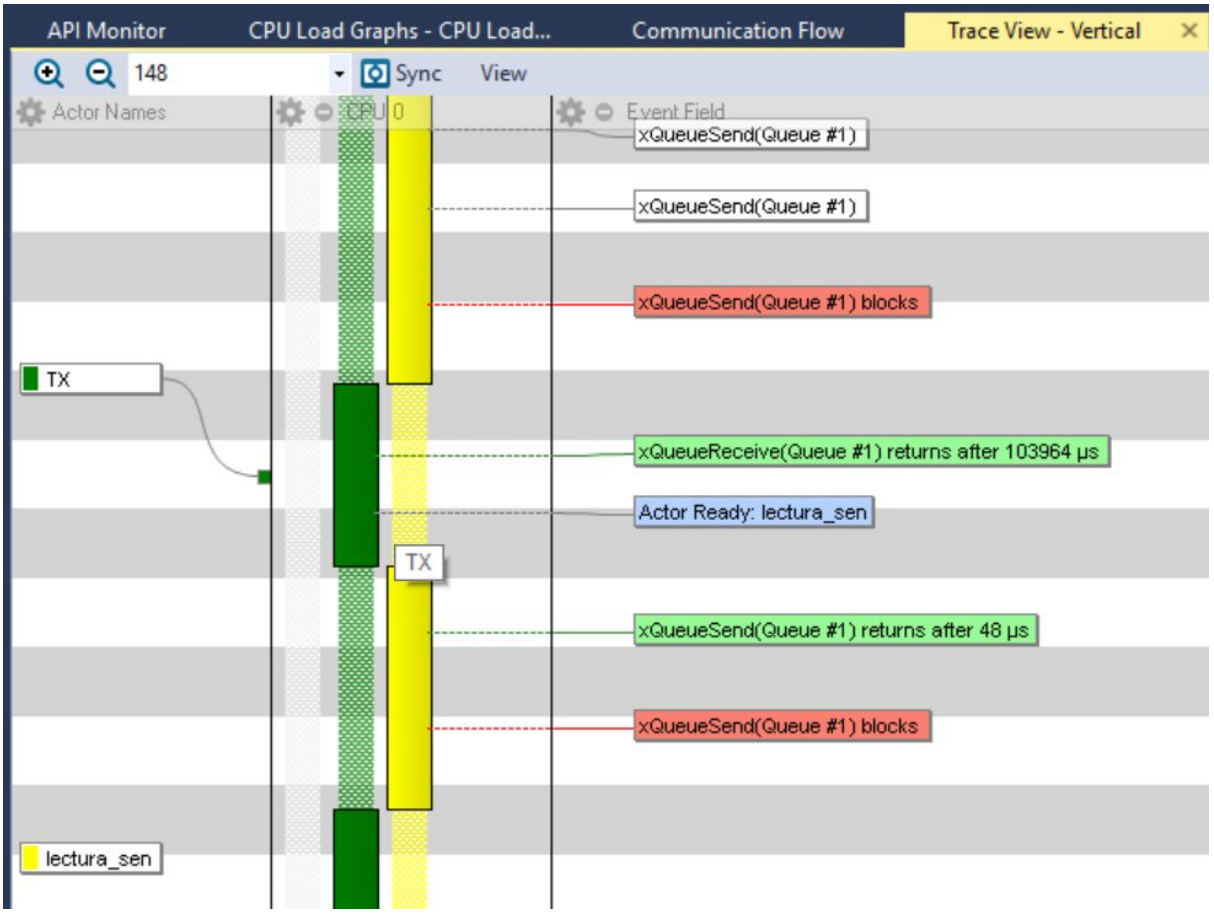


Figura 6 - Trace View Vertical (bloqueo y expulsión por prioridad)

En la Figura 6, se observa que la tarea “lectura_sensor” está ejecutando e inserta 3 caracteres en la cola de datos, los cuales llegan a llenar el buffer. El hecho de que el buffer esté lleno, impide que la tarea pueda continuar escribiendo y se bloquea, ya que necesita un recurso que no esta disponible momentaneamente. Al pasar la tarea “lectura_sensor” al estado bloqueado, y estar bloqueada también la tarea “entrada_teclado”, la única tarea que queda en estado “listo” es la tarea “transmisor” y pasa a ser ejecutada liberando (extrae un elemento) un lugar en el buffer. Inmediatamente “lectura_sensor” pasa de “bloqueado” a “listo” y ni bien se ejecuta el planificador pasa a ejecutarse porque tiene mayor prioridad. Este hecho sugiere hacer alguna mejora en el código.

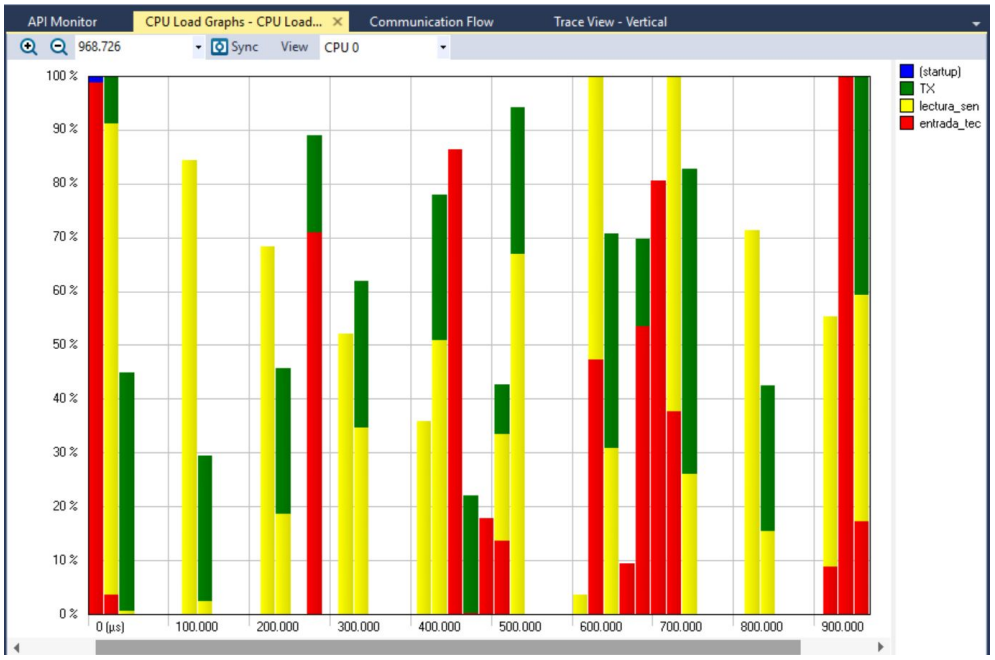


Figura 6 - CPU Load (ejecución de las 3 tareas)

En la Figura 6, se ve que si bien por momentos se alcanzan picos de utilización de CPU, en general las tareas son ejecutadas holgadamente (CPU ocioso la mayor parte del tiempo) y cumpliendo con los requisitos. Estos son: Leer el sensor con la periodicidad establecida (100 ms) y darle prioridad a la entrada por teclado, siendo de menor importancia (en este caso) el envío de los datos por terminal (print de consola).

Conclusión

En lo visto durante la realización de este trabajo se logro hacer correr sobre un sistema embebido (NXP LPC1769) un sistema operativo de tiempo real de código abierto de amplio uso en la ingeniería. El sistema embebido que se utilizó fue capaz de correr el sistema operativo de manera correcta y permitió implementar las 3 tareas que que se plantearon como objetivo. Se logró observar que la asignación de prioridades funcionó de la manera esperada y se pudo observar además cómo la sencillez de las tareas no llegaban a estresar el CPU, pudiendo éste realizar muchas más tareas si fuera necesario o más complejas.

Hay que señalar que la implementación de la tarea que simula la lectura de datos de un sensor se pudo haber implementado perfectamente con un sensor conectado a los puertos GPIO de la placa usada como se hizo en otra asignatura de la carrera, en concreto, Electrónica Digital 3. Se decidió simular la obtención de datos para simplificar ya que este no era el objetivo principal del trabajo.

En cuanto a la tarea que simula el ingreso de strings por teclado de manera aleatoria y a-periodica, la simulación de ingreso de strings fue la única posibilidad ya que la placa con

la que se trabajó no cuenta con puertos USB para la conexión de un teclado. Implementar el ingreso de datos a través de un teclado matricial es algo plausible de implementar pero es realmente complejo de ser realizado desde cero por GPIO y no agregaba valor significativo al análisis que se hizo en este trabajo.

En cuanto a la tarea del TX que tomaba datos desde la 'Queue' también hubiera sido posible utilizar el módulo UART del microcontrolador y hacer un envío de los datos por puerto serie conectando por GPIO un adaptador UART/USB a una pc y recibirlos para ser tratados o almacenados. Por razones de tiempo y de valor al análisis hecho, la implementación que se hizo en su lugar fue otra simulación: el consumidor retiraba elementos de la 'queue' pero los imprimía por la consola de la IDE en lugar de enviarlos por UART.

Si bien todos los componentes de la implementación de las tareas fueron simulados, se logró observar correctamente el funcionamiento del scheduler del FreeRTOS. Anteriormente el autor no había trabajado con un SO de tiempo real y el presente trabajo permitió aprender a configurarlo y utilizarlo. La experiencia es muy positiva ya que se incorporó como conocimiento nuevo una herramienta sumamente útil para realizar tareas específicas con requisitos de tiempo estrictos y que hagan uso de los recursos del hardware puntual. En este caso la placa LPC 1769 cuenta con muchos periféricos que permitan controlar el accionamiento de motores, automatismos, sistemas de riego, recoger información de sensores, etc. y a la vez procesar estos datos y enviarlos. El sistema FreeRTOS es compatible con un extenso número de microcontroladores y sistemas embebidos, diversos tanto en marcas de fabricantes, como en procesadores de propósito general que portan, memoria y periféricos incluidos.

Sumado a FreeRTOS se utilizó la herramienta Tracealizer, que si bien es software propietario y requiere de licencias para su uso, ha mostrado ser una herramienta muy cómoda y completa para el análisis de las tareas implementadas. Los datos que recoge Tracealizer permiten "debuguear" gráficamente (si se permite el término) las tareas que se ejecutan y cómo estas interactúan y son manejadas por el scheduler del sistema. Permitió observar consumo de CPU y líneas de tiempo de ejecución y estados que alcanzaban las tareas durante la ejecución. La herramienta es muy potente y permite observar muchos más aspectos que no se tuvieron en cuenta en este trabajo.

Apéndice

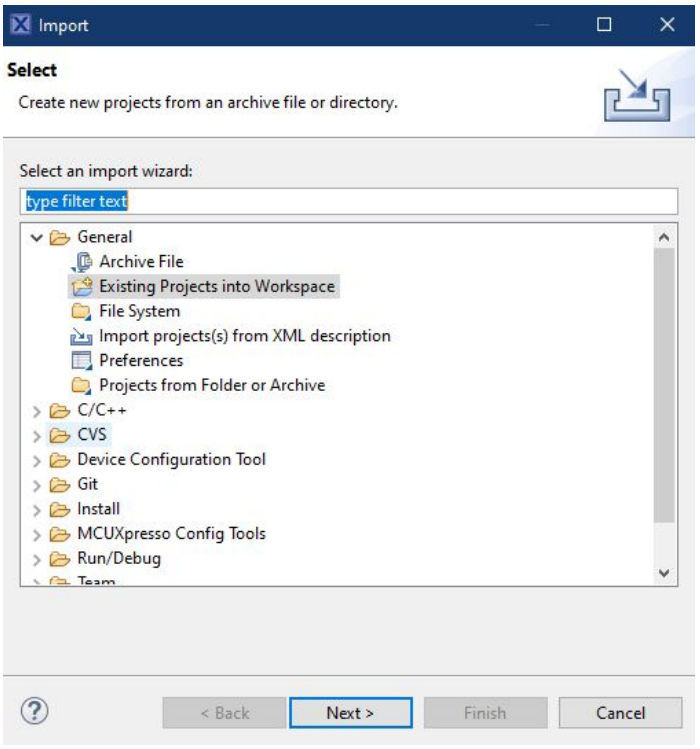
Instalación / configuración de freeRTOS

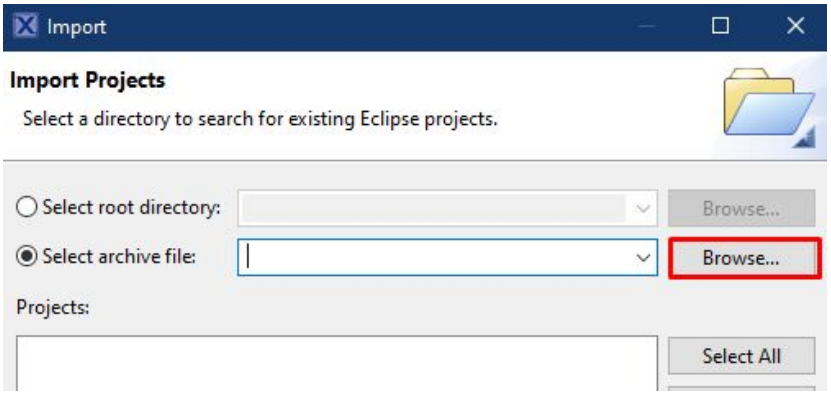
Se descarga el código fuente de FreeRTOS desde la página oficial:
https://www.freertos.org/Documentation/code/

se debe importar el proyecto que contiene el código del RTOS [1] y configurar archivos y librerías específicas para el hardware sobre el que va a correr, en este caso la LPC1769 de NXP (CORTEX-M3) [2].

LPCXpresso1769-CD

[1]





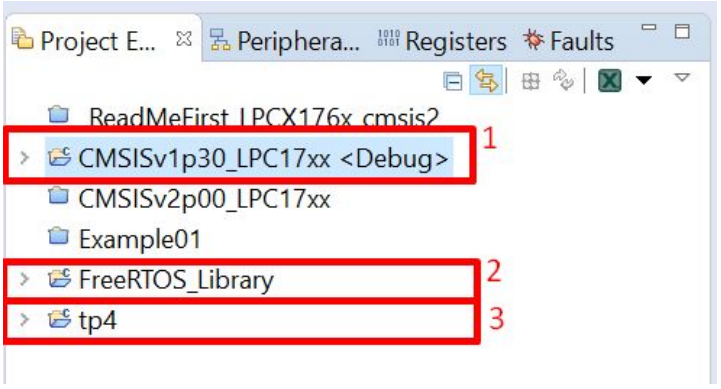
se abre el .zip que se descargó del sitio oficial y se seleccionan los siguientes proyectos:

- CMSISv1p30
- FreeRTOS_Library

En el project manager deben estar los siguientes proyectos:

1. CMSISv1p30:
   Al compilar este proyecto surge un error relacionado a las instrucciones strxb y strxh ya que las mismas no están definidas en el cortex M3 cuando se se ha de guardar el resultado de la operación en un registro que también es argumento de entrada de la operación.

   la solución en ambos casos es modificar la línea de assembler donde se invoca a



   estas instrucciones de la siguiente manera:

   reemplazar

```
__ASM volatile ("strexb %0, %2, [%1]" : "=r" (result) : "r" (addr), "r" (value) );
```

   por

```
__ASM volatile ("strexb %0, %2, [%1]" : "=&r" (result) : "r" (addr), "r" (value) );
```

   y reemplazar

```
__ASM volatile ("strexh %0, %2, [%1]" : "=r" (result) : "r" (addr), "r" (value) );
```
por
```
__ASM volatile ("strexh %0, %2, [%1]" : "=&r" (result) : "r" (addr), "r" (value) );
```

para información más precisa sobre este tema, referirse a:

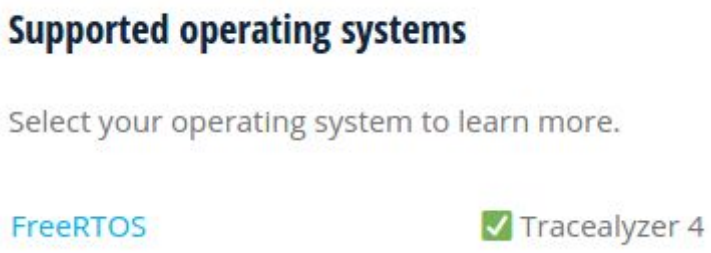
https://bug-binutils.gnu.narkive.com/493kemg7/bug-gas-13215-new-arm-cortex-m3-strexh-strexb-instructions-with-same-registers-generates-error

2. FreeRTOS_Library:
   Este proyecto se descarga desde la página oficial:
   https://www.freertos.org/Documentation/code/

3. Proyecto particular con el código que correrá sobre el RTOS

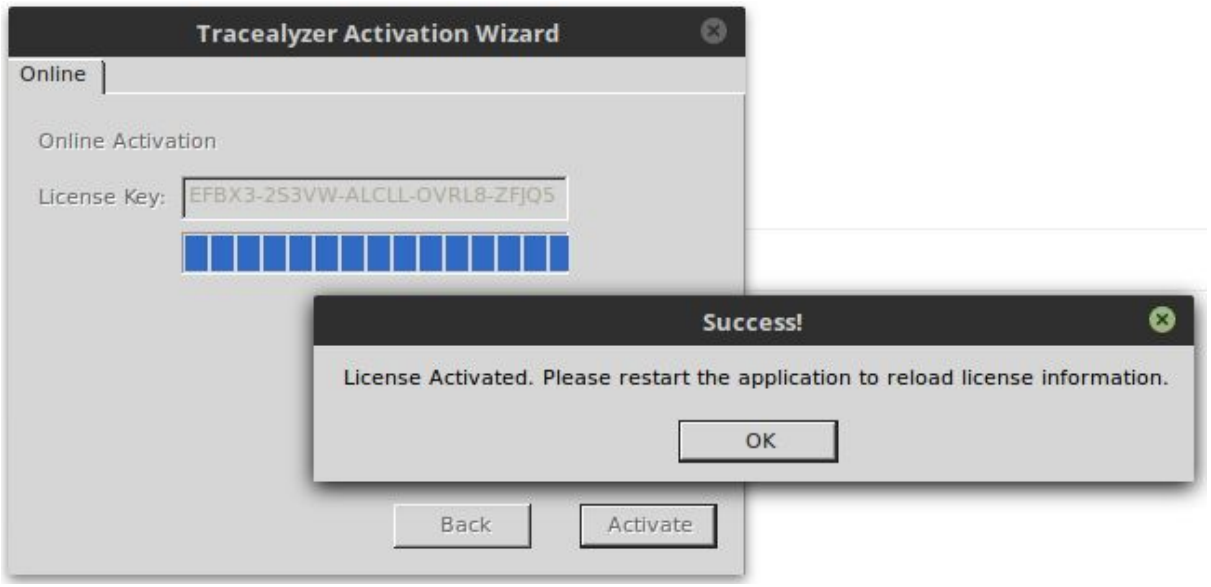
Instalación y configuración de Tracealyzer

1- En el sitio de programa: https://percepio.com/tracealyzer/ se descarga la versión que se utilizara en el sistema destino (target) en este caso es FreeRTOS



2- Para instalar Tracealyzer y sus dependencias en sistemas operativos tipo Linux se siguen las instrucciones en el siguiente link:
https://percepio.com/2018/09/11/running-tracealyzer-4-on-linux-hosts/

Si se logran instalar la aplicación y sus dependencias, una vez iniciado el programa habrá que activarlo. En este caso se introdujo una licencia de prueba. Se adjunta una captura que muestra lo que ha de ocurrir si se siguieron correctamente los pasos.



Configuración de la interfaz entre la IDE y Tracelyzer

en la IDE MCU Xpresso IDE se debe instalar el plugin que permite “exportar” los datos que serán analizados y mostrados por la aplicación Tracealyzer. Para este fin se siguen las instrucciones del siguiente tutorial:
https://mcuoneclipse.com/2017/03/08/percepio-freertos-tracealyzer-plugin-for-eclipse/

Configuración de los parámetros de TraceAlizer

se deben definir los parámetros de las snapshots o “grabaciones” que hace Tracealizer. Para ello han de modificarse los siguientes archivos:

FreeRTOSConfig.h

```
//se debe verificar que se encuentran estas líneas
#define configUSE_TRACE_FACILITY 1
#if (configUSE_TRACE_FACILITY == 1)
   #include "trcRecorder.h"
#endif
```

trcSnapshotConfig.h

En el apartado
TRC_CFG_SNAPSHOT_MODE
habrá que seleccionar el más apropiado para el análisis que se quiera efectuar, existiendo un total de 3 opciones:

- snapshot
    - TRC_SNAPSHOT_MODE_STOP_WHEN_FULL
    - TRC_SNAPSHOT_MODE_RING_BUFFER
- streaming

En este caso, se seleccionó el modo “snapshot” con la modalidad de buffer circular.
para más detalles sobre cómo setear las distintas opciones disponibles se puede consultar la guía rápida de Percepio Tracealizer:
https://percepio.com/gettingstarted-freertos/

trcConfig.h

se debe abrir el archivo trcConfig.h y verificar que esta seteado lo siguiente:

- Reemplazar la línea **#error** con un **#include** del archivo de encabezado del procesador (por ejemplo, #include 'stm32f4xx.h'). En este caso, el header correspondiente es “LPC17xx.h”
- Configurar TRC_CFG_HARDWARE_PORT para que coincida con la familia de procesadores de la placa en uso.
  En este caso, La configuración TRC_HARDWARE_PORT_ARM_Cortex_M funciona para la LPC1769 y para todos los procesadores con un núcleo Arm Cortex-M, por ejemplo:
  Cypress PSoC 4, 5, 6
  Infineon XMC4000
  Microchip (Atmel) SAM
  Renesas Sinergia, RA
  Silicon Labs EFM32

- Asegurarse de que TRC_CFG_FREERTOS_VERSION coincida con la versión de FreeRTOS en uso.
  En este caso se está usando FreeRTOS 10.3.0.

- Existe una funcionalidad de Tracealizer activada por defecto que sirve para informar periódicamente el estado del espacio libre en el stack de las tareas. Esta opción fue desactivada para este análisis ya que no interesa esa métrica y sobre todo porque produce un pequeño e innecesario overhead en el sistema.
  Para desactivarlo, se modificó la línea
  #define TRC_CFG_ENABLE_STACK_MONITOR 1
  por la línea
  #define TRC_CFG_ENABLE_STACK_MONITOR 0
  anulando así la existencia de la tarea “TzCtrl”

- Dentro del código main del programa, el guardado de los eventos registrados por Tracealizer iniciaran llamando a la función **vTraceEnable(TRC_START).**
  Esta llamada debe ubicarse en el código después de la configuración inicial del hardware, pero antes de que se hayan realizado llamadas FreeRTOS (como xTaskCreate).

Esta información también fue extraída de la la guía rápida de Percepio Tracealizer:
https://percepio.com/gettingstarted-freertos/