
Universidad Nacional
de Córdoba

# Cátedra de Sistemas Operativos II

# Trabajo Práctico Nº IV

**ZIMMEL CECCÓN, Ezequiel José**

https://gitlab.com/sistiop2/tp4_rtos

2 de febrero de 2020

# Índice

# Introducción

En el presente informe se mostrará el proceso de instalación de un sistema operativo de tiempo real (FreeRTOS) en el sistema embebido NXP-LPC1769, a la vez que se realizará el desarrollo de un programa que ejecute diversas tareas para demostrar su funcionalidad.

## Propósito

Todo desarrollo de software que posea requerimientos rigurosos de tiempo y que esté controlado por un sistema de computación, utiliza un Sistema Operativo de Tiempo Real (RTOS). Una de sus principales características es su capacidad de poseer un *kernel preemptive* y un *scheduler* altamente configurables.

El propósito del presente Trabajo Práctico es aprender a trabajar con un sistema operativo en tiempo real, comprender el *scheduling* de tareas, manejo de tareas periódicas y aperiódicas, interrupciones y las estructuras de datos que provee para el manejo de la información inter-tarea. Todo esto, haciendo uso de herramientas de control y *profiling* como Tracealyzer, haciendo posible observar el *scheduling*, el uso de memoria y de CPU a lo largo de la ejecución del programa.

## Ámbito del Sistema

El RTOS correrá sobre una placa de desarrollo NXP-LPC1769, siendo programada mediante la IDE MCUXpresso. Como sistema operativo se instalará FreeRTOS sobre la placa de desarrollo, y el análisis se realizará mediante la herramienta Trace Lyzer, instalado tanto en la placa de desarrollo como en una notebook con sistema operativo Ubuntu.

## Definiciones, Acrónimos y Abreviaturas

- **RTOS**: *Real Time Operating System*. Sistema Operativo de Tiempo Real. Estos sistemas operativos son diseñados para servir aplicaciones de tiempo real.
- **FreeRTOS**: Sistema operativo de tiempo real para sistemas embebidos con soporte para múltiples plataformas de microcontroladores.
- ***Heap***: Porción de memoria donde se realizan las asignaciones dinámicas, como las que utilizan al invocar malloc. Para liberar dicha porción de memoria, se debe utilizar alguna función, como *free* en C. La reserva de memoria en esta porción se realiza de manera aleatoria, sin seguir un orden específico.
- ***Stack***: Porción de memoria donde las reservas se hacen de forma secuencial. La manipulación de la memoria se hace automáticamente por el compilador al crear las variables.
- ***Scheduler***: Parte del *kernel* responsable de la administración de las tareas en ejecución. Es decir, determina cuándo y cómo se realizan los cambios de estado de la ejecución de las tareas.

- ***Task***: Tarea. En el contexto de un RTOS, hace referencia a las funciones independientes en las que se divide el programa a ejecutar. Las mismas se ejecutan indefinidamente una vez creadas.

## Referencias


- *https://www.freertos.org/FreeRTOS-quick-start-guide.html*
- *https://www.freertos.org/wp-content/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf*
- *https://www.nxp.com/docs/en/user-guide/UM10360.pdf*
- *http://www.cs.umd.edu/~rohit/pinout.pdf*
- *https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_Trace/RTOS_Trace_Instructions.shtml*
- *https://codereview.stackexchange.com/questions/29198/random-string-generator-in-c*
- *https://www.exploreembedded.com/wiki/LPC1768:_External_Interrupts*
- *https://exploreembedded.com/wiki/RTOS_Basics_:_TASK*
- *https://www.freertos.org/RTOS-task-states.html*


## Descripción General del Documento

El presente documento muestra todos los pasos realizados para poder cumplir con la consigna, desde la instalación de los componentes necesarios, hasta la formulación de conclusiones.

# Descripción General

## Perspectiva del Producto

El proyecto consta de dos partes: una implementación inicial, que involucra dos tareas haciendo las veces de consumidor/productor, y otra de similares características con algunos agregados, comunicación UART y sensado de variable física como ser la temperatura.

## Funciones del Producto

Al momento de cargar el programa sobre la placa, la interacción con el usuario es poca o nula. Una vez que se desea comenzar el análisis de las tareas, se debe pausar la ejecución del mismo y realizar un vuelco de memoria con lo ejecutado hasta el momento.

## Características de los Usuarios

El único usuario es el desarrollador del software, alumno autor del presente trabajo.

## Restricciones

La plataforma sobre la cual se instala el RTOS (LPC1769) presenta un *bug* que impide el análisis de uso de memoria desde Tracelyzer. A su vez, dicha plataforma no permite un análisis en tiempo real, debiéndose primero realizar un vuelco de RAM sobre un archivo, y luego leer dicho archivo al momento de realizar el análisis.

## Suposiciones y Dependencias

Se debe haber creado y configurado un proyecto desde MCUXpresso con todos los archivos requeridos por FreeRTOS y Tracelyzer. Se debe haber configurado, dentro de la IDEm, los directorios donde el compilador buscará los archivos de cabecera a incluir.

## Requisitos Futuros

No hay requerimientos futuros, ya que todos deben ser satisfechos en el presente trabajo.

# Requisitos Específicos

## Interfaces Externas

La interfaz entre el usuario y el programa en ejecución será mediante una terminal, por lo que es necesario una conexión UART entre la placa de desarrollo y la computadora del usuario. Para ésto, se utiliza un adaptador USB-UART cp2102, con Putty como emulador de terminal. Para controlar el inicio y pausa del proyecto, se utiliza la herramienta de *debugging* provista por MCUXpresso. En algunos casos se hace uso del LED integrado en la placa.

## Funciones

La función del sistema consta de ejecutar un programa del consumidor/producto, e ir mostrando los datos de ejecución por pantalla. En un momento, el usuario podrá actuar como productor, al presionar sobre una tecla.

## Requisitos de Rendimiento

No hay requisitos de rendimiento.

## Restricciones de Diseño

Se debe instalar el RTOS sobre una arquitectura compatible con FreeRTOS.

## Atributos del Sistema

La plataforma del sistema embebido es una placa de desarrollo de NXP basada en un LPC1769. El mismo es un microcontrolador ARM Cortex-M3 con un alto nivel de integración y bajo consumo energético, corriendo a frecuencias de 120 MHz. Entre algunas características notables, se pueden mencionar:

- 512 Kb de memoria flash
- 64 Kb de memoria de datos
- Ethernet
- USB tanto como dispositivo como host,, y soporte OTG
- Controlador DMA de 8 canales.
- 4 UARTs
- 3 SPI
- 3 I2C
- ADC de 12 bit.
- 70 pines de E/S de propósito general.

# FreeRTOS

FreeRTOS es un *kernel* de tiempo real, sobre el cual se pueden construir aplicaciones destinadas a sistemas embebidos para cumplir con sus requisitos en tiempo real, permitiendo que las aplicaciones se organicen como una colección de hilos de ejecución independientes.

Para el caso del microcontrolador Cortex-M3, que tiene un solo núcleo, sólo un hilo puede ejecutarse a la vez, siendo el *kernel* quien decide qué hilo se debe ejecutar al examinar la prioridad asignada a cada uno de ellos.

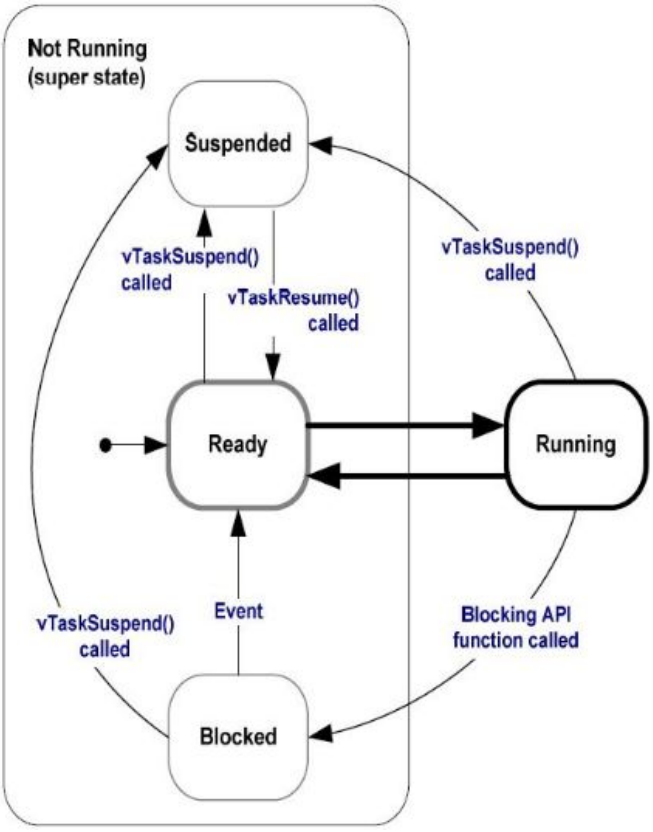
Cada tarea es un programa pequeño en sí, que tiene un punto de entrada y normalmente corre en un *loop* infinito, y no termina. Las tareas en FreeRTOS no deben tener permitido retornar de las funciones que las implementan, sino que deben de ser eliminadas explícitamente si no se necesitan más.

Cada tarea puede permanecer en uno de los siguientes estados:

- **Ejecución**. Actualmente está utilizando el procesador. Si el procesador en el que se está ejecutando el RTOS sólo tiene un único núcleo, entonces sólo puede haber una tarea en estado de ejecución en un momento dado.
- **Listo**. Las tareas listas son aquellas que son capaces de ejecutarse (no están en el estado Bloqueado o Suspendido) pero que no se están ejecutando actualmente porque una tarea diferente de igual o mayor prioridad ya está en el estado Ejecución.
- **Bloqueado**. Se dice que una tarea está en estado de bloqueo si está actualmente esperando un evento temporal o externo. Por ejemplo, si una tarea llama a

vTaskDelay() se bloqueará (se colocará en el estado Bloqueado) hasta que el período de retraso haya expirado - un evento temporal. Las tareas también pueden bloquearse para esperar una cola, un semáforo, un grupo de eventos, una notificación, etc. Las tareas en el estado Bloqueado normalmente tienen un período de "tiempo de espera", tras el cual la tarea se desbloqueara, incluso si el evento que la tarea estaba esperando no ha ocurrido. Las tareas en el estado Bloqueado no utilizan ningún tiempo de procesamiento y no pueden ser seleccionadas para entrar en el estado de Ejecución.

- **Suspendido**. Al igual que las tareas que se encuentran en estado Bloqueado, las tareas en estado Suspendido no pueden ser seleccionadas para entrar en estado de Ejecución. Sin embargo, las tareas en estado Suspendido no tienen un tiempo de espera. En su lugar, las tareas sólo entran o salen del estado Suspendido cuando se les ordena explícitamente que lo hagan, a través de las llamadas a la API vTaskSuspend() y xTaskResume(), respectivamente.




# Diseño de solución

El proyecto consta, a grandes rasgos, de los siguientes pasos:

1. Instalación y configuración de FreeRTOS en el sistema embebido seleccionado.
2. Creación un programa con dos tareas simples (productor/consumidor).
3. Adaptar el programa para realizar el *profiling*.
4. Realizar el análisis completo de memoria y procesos en  Tracelyzer.

5. Modificar el programa para cumplir con la segunda consigna, en donde se tienen ahora dos productores, uno periódico y otro aperiódico.
6. Realizar el *profiling*.

# Implementación y Resultados

## *HOW TO*

**Instalación de IDE MCUXpresso**

1. Descargar la IDE desde el enlace *https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE?tab=Design_Tools_Tab*
   Requiere cuenta de usuario.
2. Una vez descargado el binario, proceder a ejecutarlo *sudo ./<nombre del binario>*

**Uso de FreeRTOS**

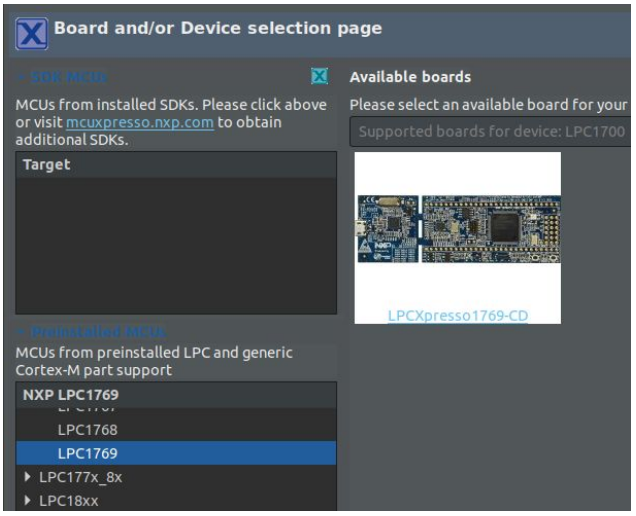
1. Descargar los recursos desde el enlace *https://www.freertos.org/*
2. Crear un proyecto, siguiendo la estructura sugerida en su página. No obstante, se sugiere hacer uso de los proyectos base que se nos ponen a disposición en el siguiente enlace *https://www.freertos.org/FreeRTOS-quick-start-guide.html#page_top.*

**Instalación de Tracelyzer**

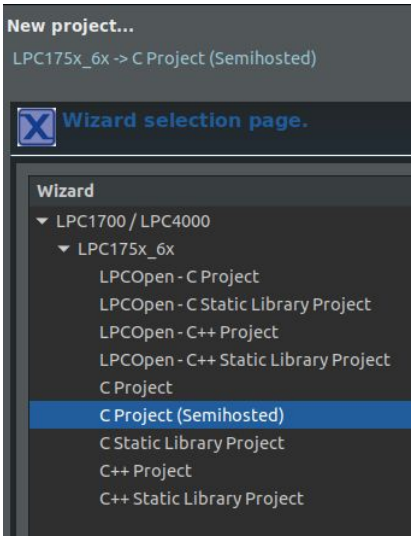
1. Para implementar las funciones de Tracelyzer, debe hacerse uso de un extra "*FreeRTOS+Trace*". El mismo forma parte de un paquete de funciones extendidas de FreeRTOS llamado "*FreeRTOS plus*". *https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_Trace/FreeRTOS_Plus_Trace.html*
   Al proyecto, con los archivos de FreeRTOS, es necesario agregarle la funcionalidad "+trace". Eso se logra copiando y pegando archivos en directorios específicos, y modificando ciertos archivos según la plataforma objetivo.

Una vez realizada la descarga e instalación de los recursos antes descritos, crearemos el proyecto y llevaremos a cabo las configuraciones necesarias.

1. Dentro del entorno de desarrollo MCUXpresso, crearemos un nuevo proyecto desde las opciones **File -> New -> New C/C++ Project**
2. Seleccionamos la placa de desarrollo

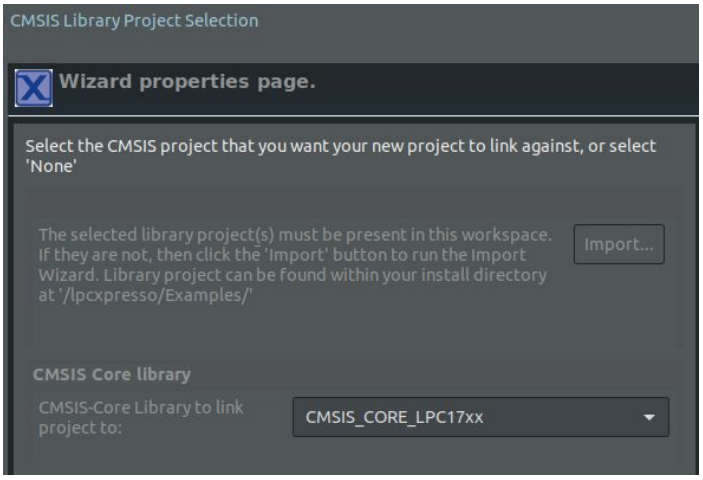



3. Seleccionamos el tipo de proyecto. Si se desea hacer uso de la consola del IDE, a los fines del *debug*, debe de seleccionarse '*Semihosted*'.

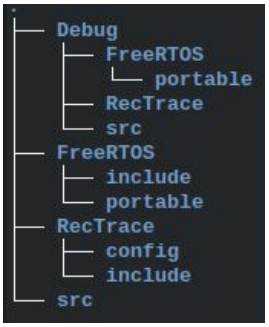



Una vez seleccionado el tipo de proyecto, se solicitará el nombre del proyecto y el directorio en dónde almacenarlo.

4. Importamos las librerías CMSIS y luego las seleccionamos. La librería CMSIS DSP puede ignorarse.



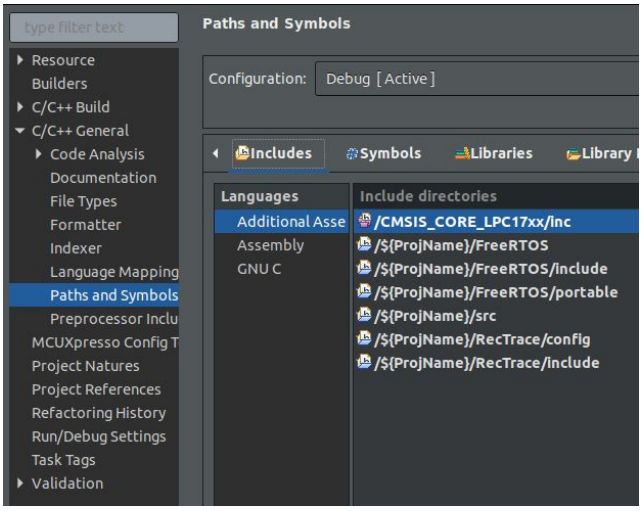
5. Por último, solicitará elegir el compilador y la estructura del proyecto, pudiendo dejarse por defecto.
6. Debemos agregar las librerías de FreeRtos y Tracealyzer y asociarlas al proyecto. La estructura de directorios debería quedar según la siguiente figura:

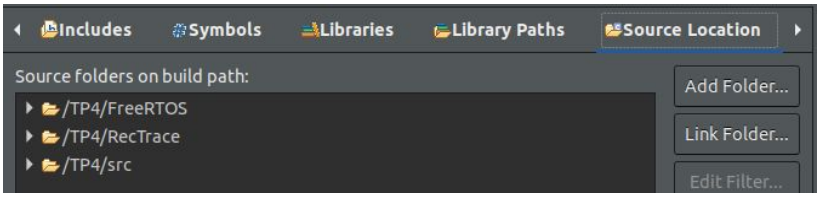



donde:

- *Debug*: carpeta autogenerada luego de la compilación del proyecto.
- FreeRTOS: librería de FreeRTOS. El nombre puede reemplazarse.
- RecTrace: librería de tracealyzer. El nombre puede reemplazarse.
- src: carpeta donde se aloja nuestro código. Dicha carpeta debe contener el archivo *FreeRTOSConfig.h*

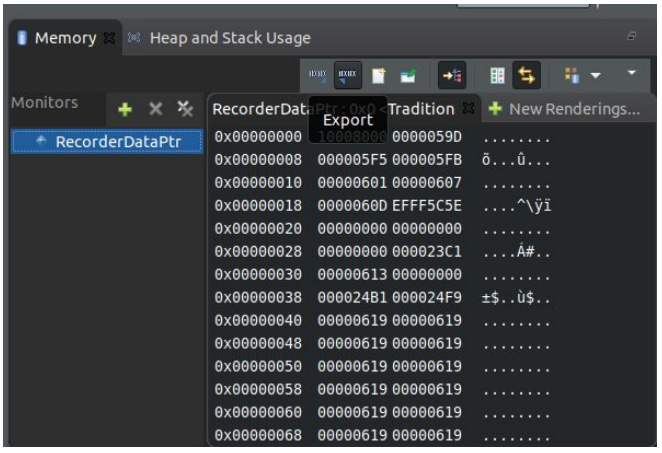
Luego, dentro de la IDE y accediendo a las propiedades del proyecto, agregamos los directorios de las cabeceras de FreeRTOS y Trace, y los directorios de los códigos fuentes.

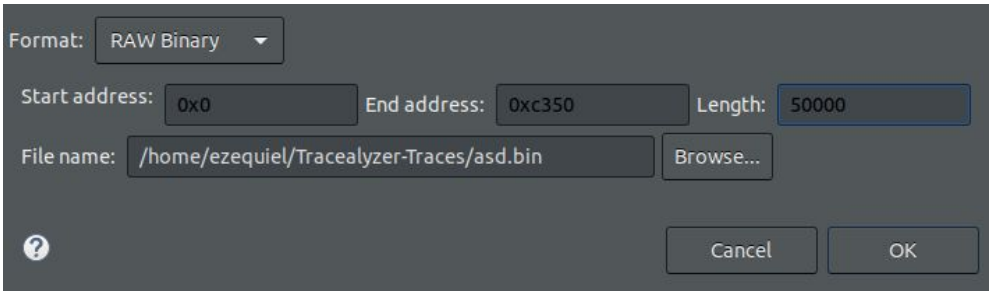



7. A la hora de realizar el *profiling* del código, mediante Tracealyzer, el volcado de memoria puede realizarse de dos maneras:

a. Manual. Para obtener los datos que se utilizan en tracealyzer, se debe indicar la dirección de memoria donde empiezan los datos de *trace*, agregando RecorderDataPt, obteniendo así el valor de memoria en donde se encuentra. Desde esta dirección de memoria se hace el *dump*.



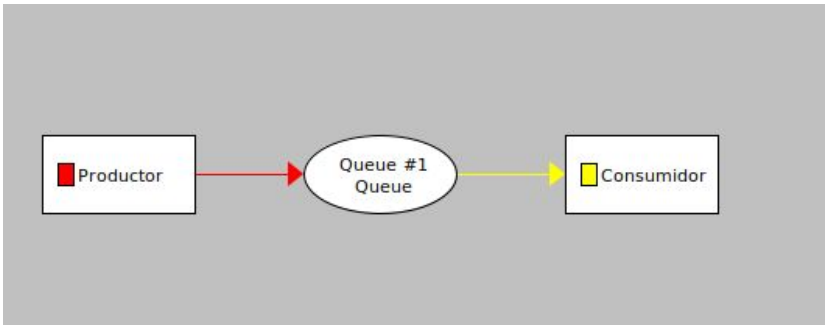

b. Automatizado. Instalar el plugin de Tracealyzer en la IDE según el tutorial del siguiente enlace:
*https://mcuoneclipse.com/2017/03/08/percepio-freertos-tracealyzer-plugin-for-eclipse/*
Desde la misma IDE se permite hacer un *snapshot* durante la ejecución del programa. Si el programa local Tracealyzer se encuentra en ejecución, la IDE MCU enviará automáticamente el vuelco de memoria.

# Ejecución

## 2 Tareas: Productos / Consumidor simple

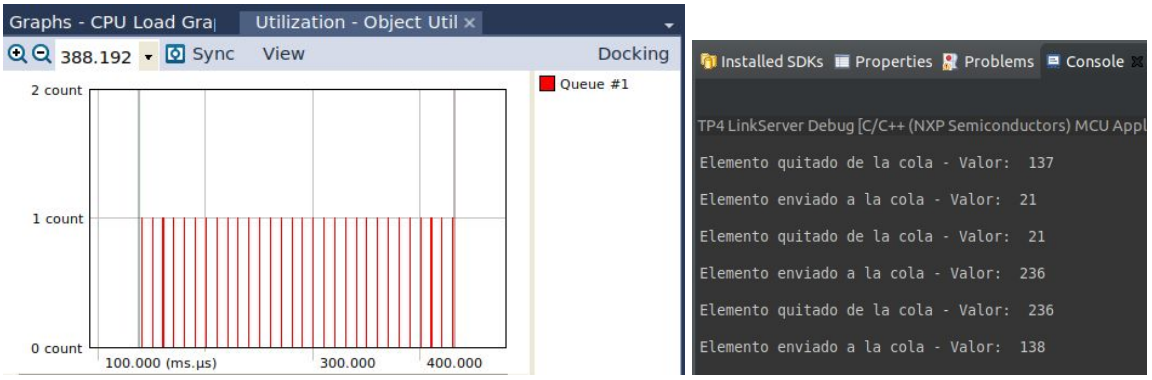
En esta etapa, a los fines familiarizarse con el funcionamiento del sistema y de la herramienta tracealyzer, se hizo uso de un ejemplo de la bibliografía, a partir del cual se implementó un programa que consiste en dos tareas simples y una cola. Una de las tareas emite mensajes (productor) y la otra los va recibiendo (consumidor). A continuación, se muestran los datos obtenidos con la herramienta tracealyzer.

Los actores intervinientes y cómo se relacionan se observa en la siguiente figura:

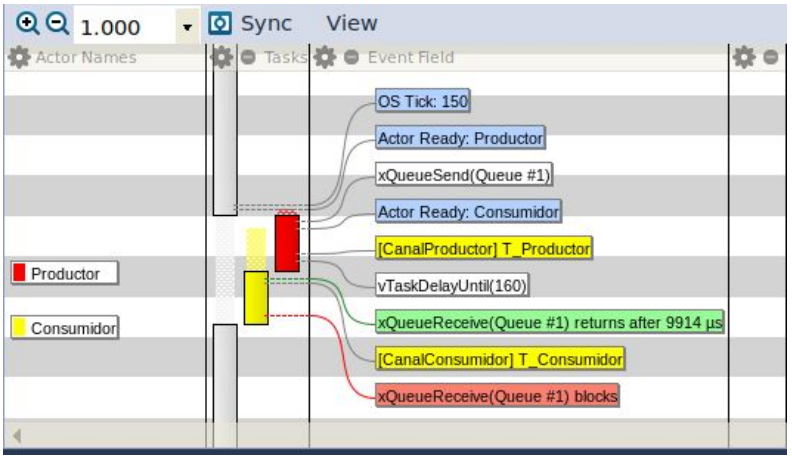



Durante la ejecución, los elementos colocados en la cola por el productor son sacados por el consumidor.

El uso de la cola puede observarse en la siguiente figura, la cual coincide con la ejecución del programa.

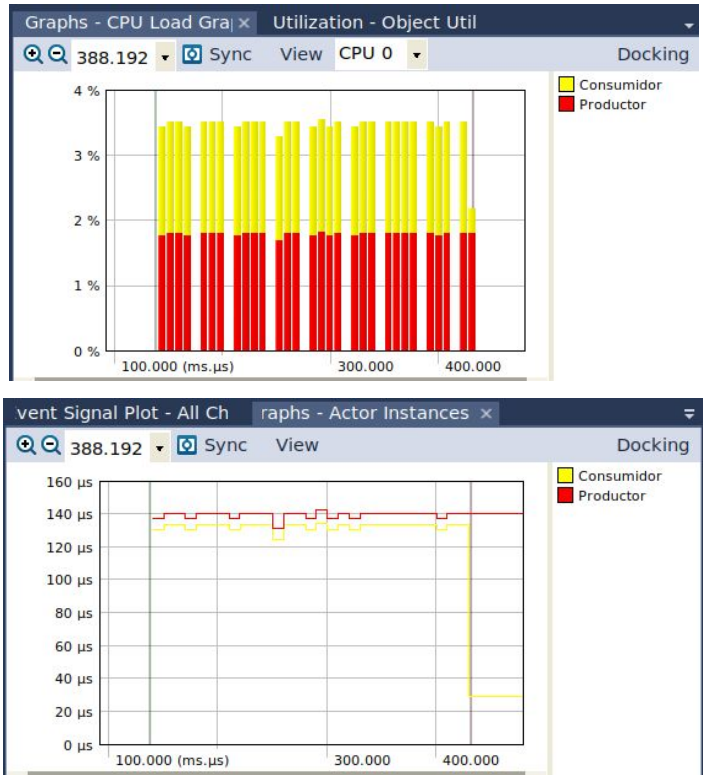



Una vez creadas y planificadas las tareas, éstas comenzarán a competir por el procesador. En la siguiente figura se observa que la tarea de mayor prioridad (Productor) termina de ejecutarse y, recién en ese momento, se comienza a ejecutar la de menor prioridad (Consumidor), aun cuando esta última estuviera disponible para ejecutar antes de que finalice el productor.




Luego, podemos observar el uso de la CPU por parte de las tareas; en el transcurso del tiempo las tareas consumidor y productor se reparten el consumo de CPU. Este

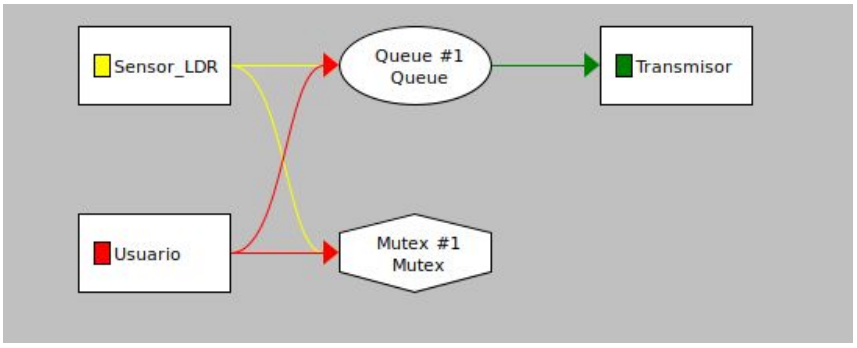
consumo es muy pequeño debido que ninguna tarea demanda carga computacional y son de poca duración.




Simulación con 5 tareas en *5 Tareas: 2 Productores / 3 Consumidores*.

## 3 Tareas simuladas: Sensor / Usuario / Consumidor

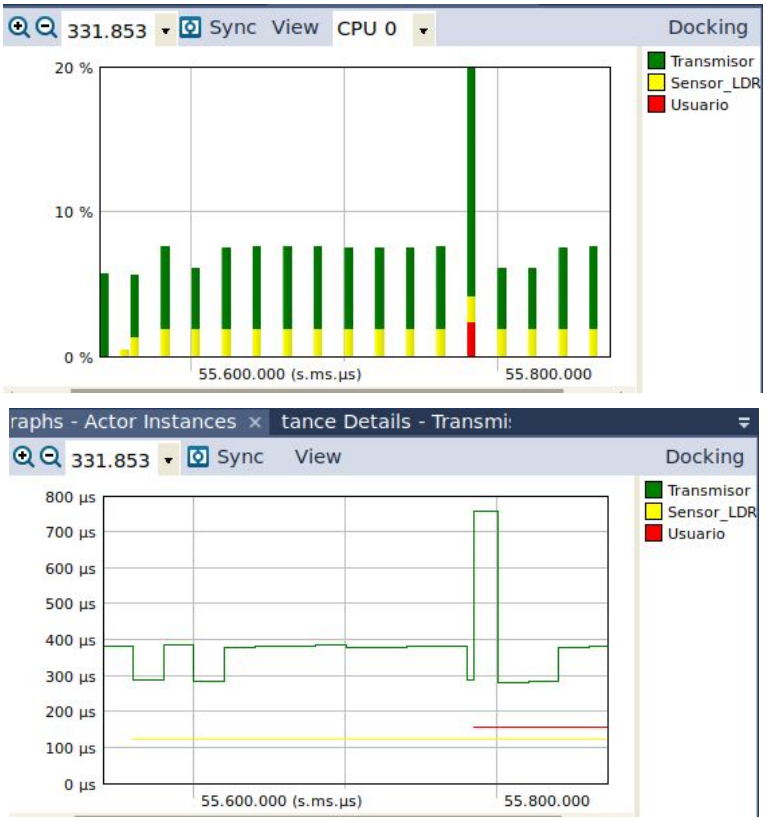
Se pueden observar las tareas involucradas y los recursos de los que hace uso. Para garantizar la exclusión mutua entre los productores en el uso de la cola compartida se emplea un mutex.



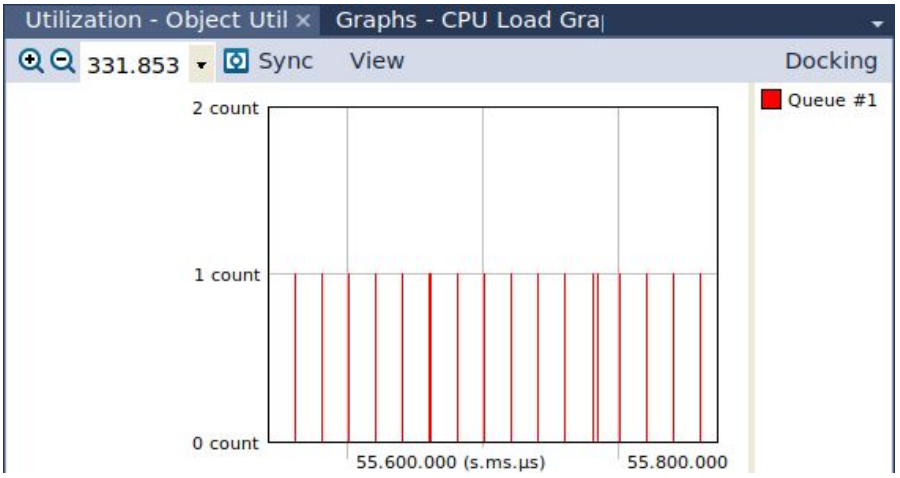

La tarea que hace mayor uso del CPU es aquella a la cual no se le impusieron restricciones de tiempo y de exclusión mutua, por lo que nunca se bloquea estando siempre a disposición para ejecutar.

La tarea que le sigue en el uso del CPU es aquella de características periódica, dejando para el último la tarea aperiódica. Ésta última está disponible para ejecutar cada

[100 - 1000]ms, sumado a la restricción de exclusión mutua en el acceso a la cola compartida.

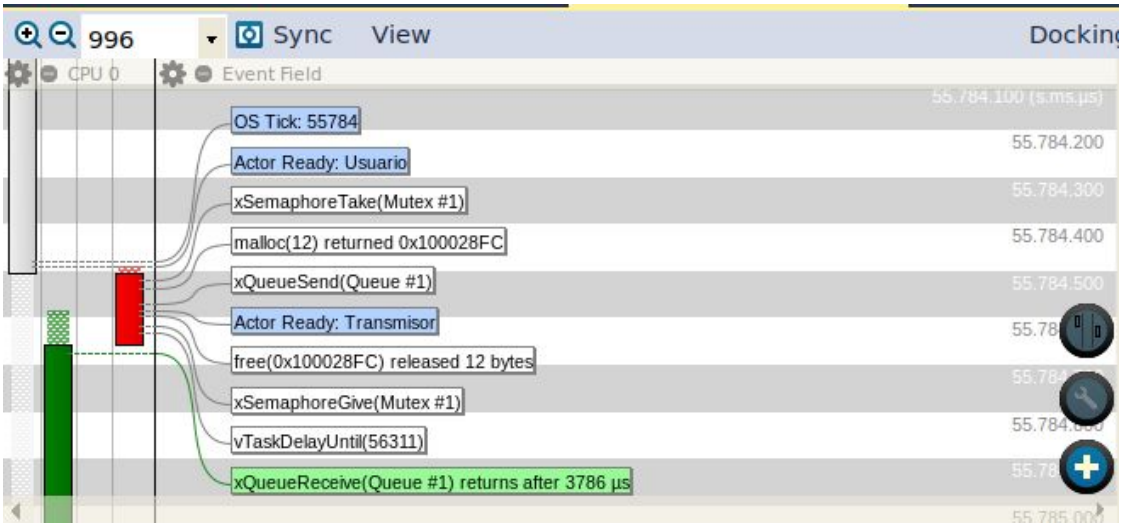



El comportamiento en ejecución se corresponde con lo que se observa en la cola compartida. La tarea consumidor, como se mencionó anteriormente, no posee ningún tipo de restricción por lo que cada elemento insertado es consumido.

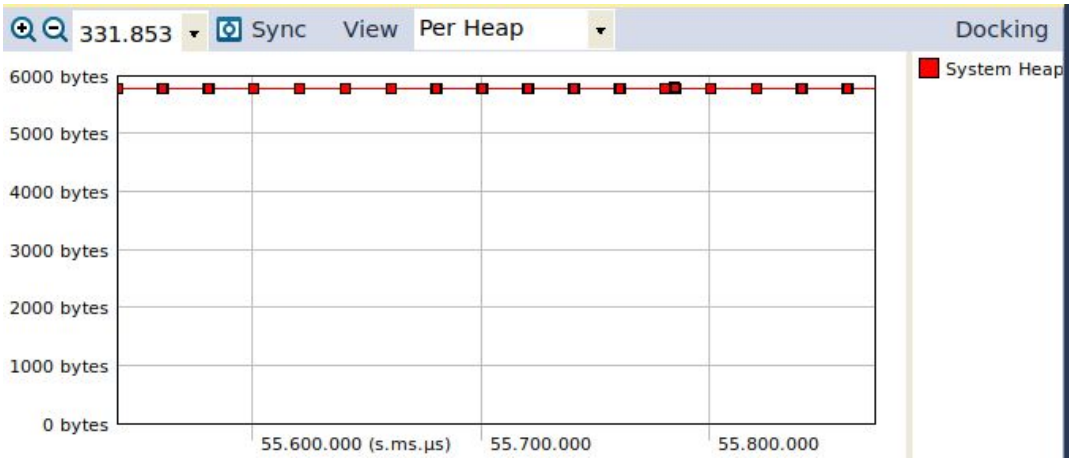



La siguiente imagen muestra cómo se realiza la planificación. Como la tarea aperiódica tiene más prioridad que la tarea del consumidor, ésta última permanecerá

bloqueada hasta que la tarea más prioritaria finalice. También se observa cómo hacen uso de la exclusión mutua para poder tener acceso a la cola compartida.




Por último, se muestra el total de memoria que emplea el código en ejecución.




Para realizar la implementación con componentes físicos ver *3 Tareas desplegadas: Sensor / Usuario / Consumidor.*

# Conclusiones

Con la realización de esta actividad se aprendieron a utilizar herramientas para trabajar con tareas de tiempo real, como realizar su creación y manipulación de manera dinámica con sus *handlers*. Por otro lado, se trabajó con estructuras propias de RTOS, como las colas y los semáforos, que permiten la comunicación entre la tareas.

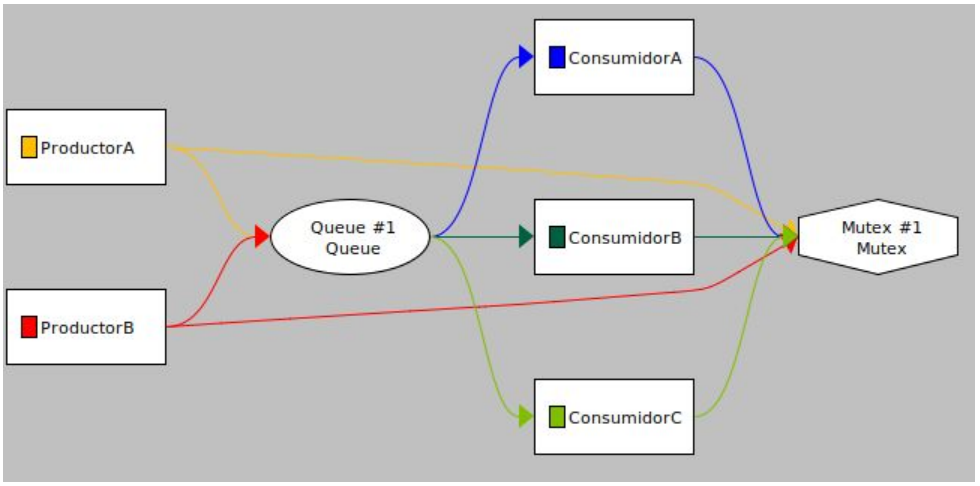
Se realizó la implementación con interrupciones, aprendiendo cómo realizar la suspensión y la reactivación de la tarea asociada a dicha interrupción.

Por último, se tomó conciencia de lo que implica la gestión de la memoria de manera dinámica, a los fines de evitar problemas vinculados a ellos.

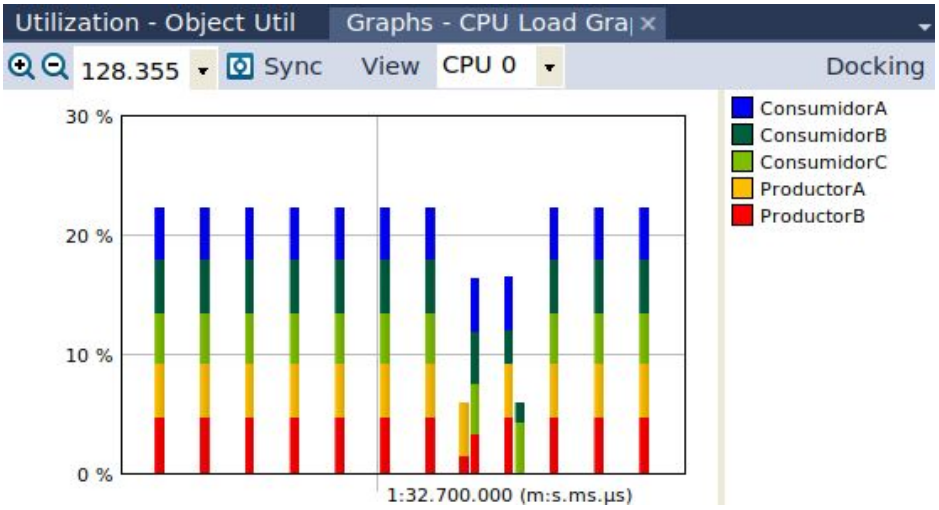
# Anexos

## 5 Tareas: 2 Productores / 3 Consumidores

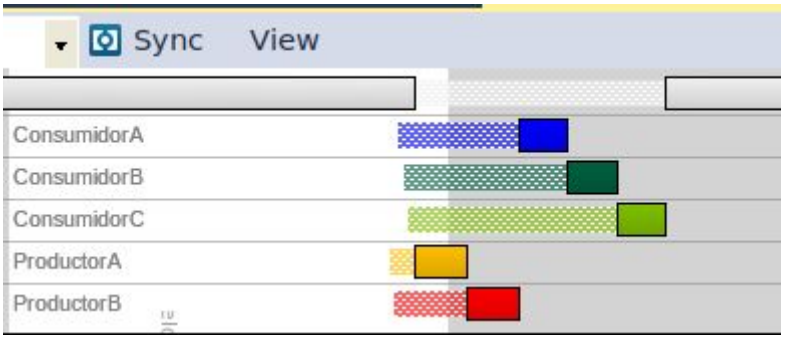
Se crean y planifican 5 tareas, variando sus prioridades de manera aleatoria cada 5 segundos. Se observa el uso compartido de la cola por los 2 productores y los 3 consumidores. A su vez, todas las tareas requieren del acceso a la exclusión mutua para poder interactuar con la cola compartida.

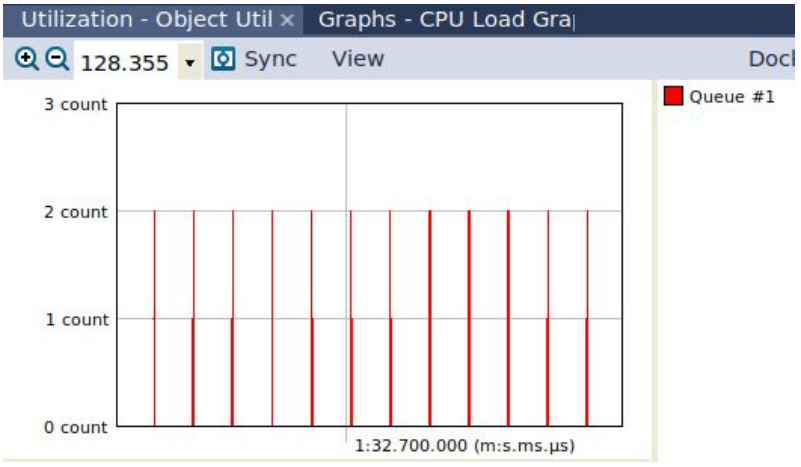



La variabilidad en el uso del CPU es debido a las prioridades dinámicas de las tareas.



Como cada tarea requiere del acceso a mutex para poder ejecutar su porción de código, y éste solo es devuelto una vez finalizada la ejecución de la porción, no existe cambio de contexto entre tareas. La asignación del mutex a 2 o más tareas bloqueadas en su espera se realiza por orden de prioridad.

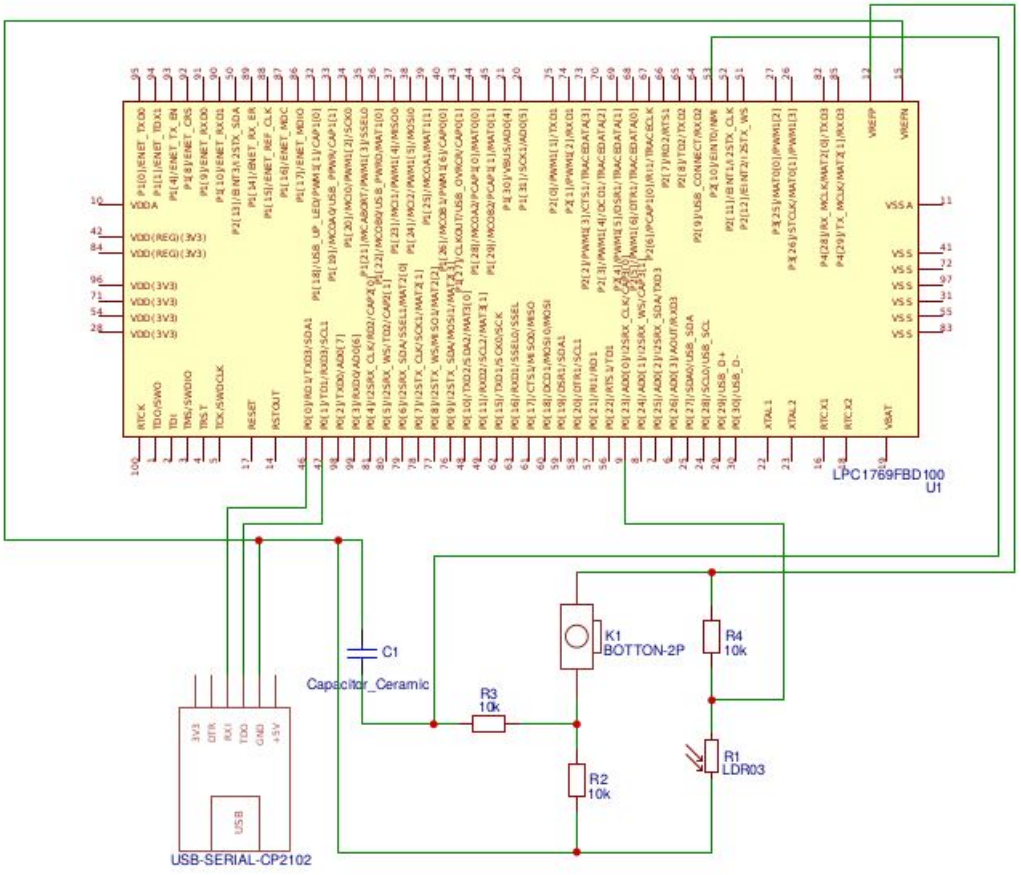



# 3 Tareas desplegadas: Sensor / Usuario / Consumidor

Repitiendo las mismas condiciones anteriores, la planificación de 3 tareas donde 2 de ellas son productores y una un consumidor, se procede a desplegar el desarrollo sobre una protoboard realizando el siguiente conexionado de componentes, donde:

- **CP2102**. Consumidor. Empleado para la comunicación serial entre la LPC1769 y el ordenador. Se hace uso del periférico UART3, pines P0.0 (TXD3) y P0.1 (RXD3).
- **Fotoresistor**. Productor. Empleado por la tarea periódica. El conexionado se realiza sobre el pin P0.23 (ADC0) configurado en modo burst.
- **Botón**. Productor. Empleado por la tarea aperiódica. El conexionado se realiza sobre el pin P2.10 (EINT0) configurado para detectar interrupciones externas. El capacitor y la resistencia R3 se agregan como antirrebote.


LPC1769FBD100
U1
C1
Capacitor_Ceramic
K1
BOTTON-2P
R4
10k
R3
10k
R2
10k
R1
LDR03
USB-SERIAL-CP2102