Universidad
Nacional
de Córdoba

Cátedra de Sistemas Operativos II

Trabajo Práctico Nº 4

Navarro, Matias Alejandro

21 de junio de 2019

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, y el desarrollo de un programa con diversas tareas para demostrar su funcionalidad.

Propósito

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

Ámbito del Sistema

Ambas aplicaciones corrieron en un sistema embebido (LPC1769 que cuenta con un ARM Cortex M3) de arquitectura compatible con FreeRTOS, que es el sistema operativo de tiempo real utilizado. Una de las características principales de este tipo de SO, es su capacidad de poseer un kernel preemptive y un scheduler altamente configurable.
Por otro lado, la placa se encuentra conectada a una notebook, desde donde se observa el flujo de ejecución y demás datos de debugging utilizando la herramienta Tracealyzer.

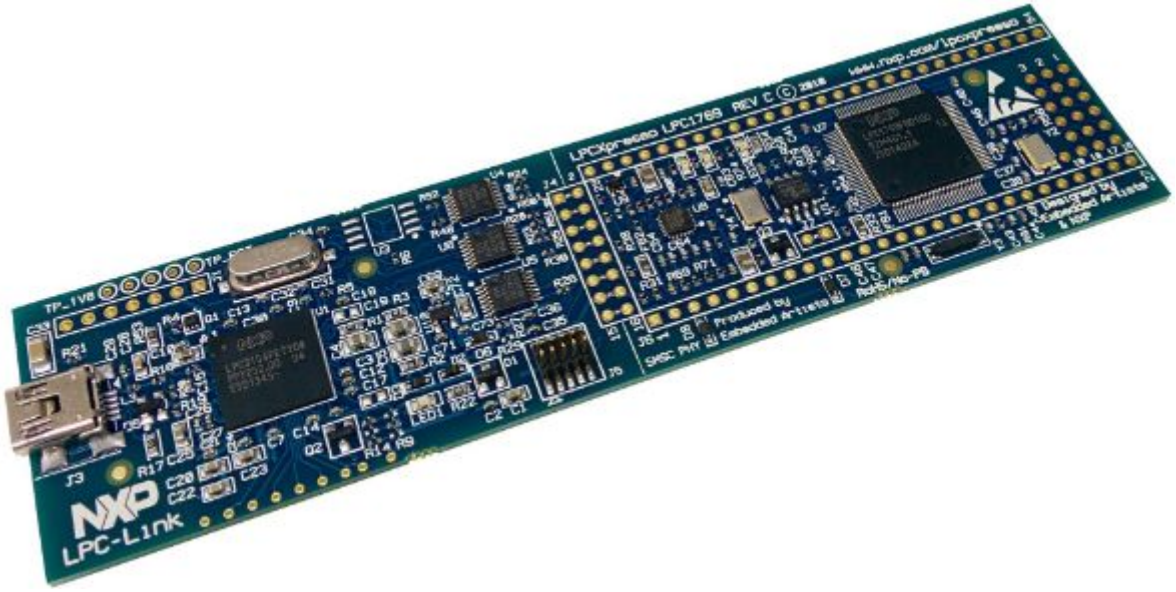

Figura 1: NXP LPC1769

En la siguiente figura, se puede observar el dispositivo utilizado para la comunicación serial entre la placa y la notebook.

Figura 2: CP2102

Definiciones, Acrónimos y Abreviaturas

- RTOS: Real Time Operating System
- Preemptive: es una manera en que los sistemas operativos pueden proveer multitarea, es decir, la posibilidad de ejecutar múltiples procesos al mismo tiempo.

Referencias

[1] Filminas Tiempo Real Sistemas Operativos II - UNC

[2] https://www.freertos.org/Creating-a-new-FreeRTOS-project.html

[3] https://www.freertos.org/Embedded-RTOS-Queues.html

[4] https://www.freertos.org/a00110.html

[5] https://github.com/lucasrangit/FreeRTOS_LPCX176x_Example

Descripción General del Documento

El presente documento muestra una visión del proceso desarrollo del trabajo práctico, su diseño, implementación, esquemas a seguir y conclusiones sacadas del mismo.

Descripción General

Perspectiva del Producto

Este trabajo práctico se realizó para implementar dos aplicaciones, la primera consta de dos tareas que simulan el caso de un productor-consumidor, la segunda consta de tres tareas que se alternan dependiendo de su prioridad y tiempo de ejecución. Se implementan en un sistema embebido con RTOS para observar el comportamiento del scheduler preemptive.

Funciones del Producto

Las funciones que ofrece la aplicación son: sensor de temperatura, generación de strings de longitud variable y transmisión de datos por puerto serie a una terminal.

Características de los Usuarios

El usuario necesita contar una placa LPC1769, un módulo UART para puerto serie (en este caso se utilizó el CP2102), y una computadora con una terminal serie (se utilizó Putty). Para el análisis de los datos se uso la aplicación Tracealyzer (el programa se obtuvo con una licencia gratuita de 30 días).

Restricciones

El sensor de temperatura no está implementado físicamente, sino que se usa un array de enteros que simulan ser temperaturas censadas aleatoriamente; al igual que los datos de usuario, los cuales se simulan con un array de strings de tamaño variable.

Suposiciones y Dependencias

- El sistema embebido debe tener una arquitectura compatible con FreeRTOS
- Las librerías de FreeRTOS se deben configurar de acuerdo al sistema embebido utilizado, en este caso ARM Cortex M3.

Requisitos Específicos

Interfaces Externas

Placa LPC1769 o similar con ARM Cortex M3-M4, o cualquiera compatible con FreeRTOS; CP2102 para comunicación serie; terminal serie Putty o similar.

Funciones

Las funciones que brinda el producto son:

- Función productor y función consumidor: ambas comparten un buffer finito (representado por un número entero), uno produce (aumenta este valor) y otro consume (decrementa este valor).
- Sensor de temperatura: toma datos generados al azar entre 0 y 100 y los envía a una cola. Función periódica.
- Generar strings: toma datos de usuario mediante un arreglo al azar y los envía a una cola. Función aperiódica.
- Transmisor: envía el valor recibido mediante una cola a la terminal de una computadora por puerto serie (UART)

Restricciones de Diseño

- ➢ El sistema operativo instalado debe ser FreeRTOS.
- ➢ El sistema embebido debe ser compatible con ese sistema operativo.
- ➢ Se debe utilizar Tracealyzer para el análisis de las aplicaciones.
- ➢ El análisis se realiza a ambas aplicaciones la que implementa el problema del consumidor-productor y la que presenta las tres tareas.

Implementación y Resultados

1. Instalación de IDE para la placa

Se descarga la IDE LPCXpresso para poder compilar y debuggear los proyectos. Se obtuvo una licencia gratuita y se corroboró su funcionamiento con un proyecto simple en el cual consistía prender y apagar un led.

Figura 3: LPCXPRESSO

Se abre el entorno de desarrollo y se crea un nuevo proyecto:

- File -> New Project -> Lpc C project

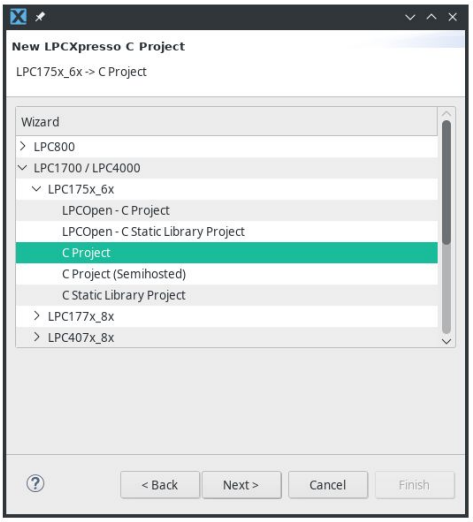

Figura 4: Crear proyecto

- Selección de placa

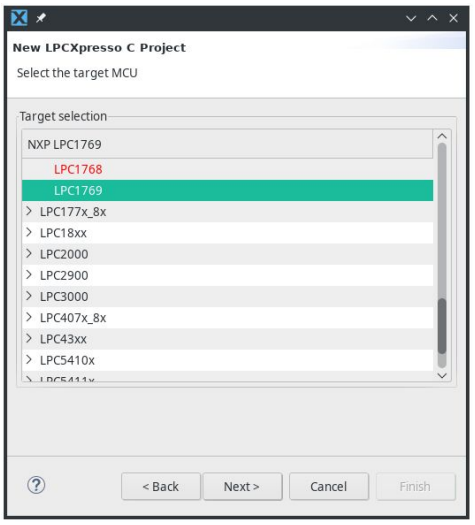

Figura 5: Seleccionar placa de desarrollo

- Se importaron las librerías **`CMSISv1p30_LPC17xx`** correspondientes a la placa y se probó que la placa y el entorno funcionaran mediante el proyecto mencionado anteriormente.

```
#ifdef __USE_CMSIS
#include "LPC17xx.h"
#endif

#define AddrFIO0DIR 0x2009C000 // Define las posiciones de memoria
#define AddrFIO0SET 0x2009C018 // donde se encuentran los registros
#define AddrFIO0CLR 0x2009C01C // que configuran al GPIO0
#define AddrFlO0PIN 0x2009C014

unsigned int volatile * const FIO0DIR = (unsigned int *) AddrFIO0DIR; // Define los punteros a las direcciones
unsigned int volatile * const FIO0SET = (unsigned int *) AddrFIO0SET; // de memoria definidas por las
unsigned int volatile * const FIO0CLR = (unsigned int *) AddrFIO0CLR; // correspondientes constantes.
unsigned int volatile * const FIO0PIN = (unsigned int *) AddrFlO0PIN;

int retardo(unsigned int); //prototipo de la función retardo

int main(void) {
    unsigned int time = 364500; //control de tiempo de retardo
    *FIO0DIR |= (1 << 22); //define al pin 22 del Puerto 0 como de salida (en "1").

    while (1) {
    *FIO0SET |= (1 << 22); //pone en alto al pin 22, encendiendo el led.
    retardo(time);
    *FIO0CLR |= (1 << 22); //pone en cero al pin 22, apagando el led.
    retardo(time);
    }
    return 0;
}
int retardo(unsigned int time) {
    unsigned int i;
for (i = 0; i < time; i++); //lazo de demora
    return 0;
    }
```

Figura 6: Código fuente del programa “Prende-Apaga”

2. Instalación del sistema FreeRTOS.

Una vez asegurado el funcionamiento de la placa con un programa, se procede a importar un ejemplo (obtenido de: https://github.com/lucasrangit/FreeRTOS_LPCX176x_Example) como lo recomienda la página oficial de **FreeRTOS.** La cual incluye los siguientes archivos fuente de FreeRTOS:

- FreeRTOS/Source/tasks.c
- FreeRTOS/Source/queue.c
- FreeRTOS/Source/list.c
- FreeRTOS/Source/portable/MemMang/heap_2.c
- FreeRTOS/Source/portable/GCC/ARM_CM3/port.c

Posterior a esto, se incluyeron algunos headers que fueron necesarios:

- FreeRTOS/Source/include
- FreeRTOS/Source/portable/GCC/ARM_CM3/portmacro.h
- FreeRTOS/Demo/CORTEX_1768_GCC_RedSuite/FreeRTOSConfig.h

Finalmente, se debe editar el archivo de configuración **FreeRTOSConfig.h** que está dentro de la carpeta **FreeRTOS/Demo/[placa_especifica]** (la LPC1769 no aparece pero se puede usar la de la LPC1768) y que define los parámetros de la placa y requisitos específicos de la aplicación. Esto significa que este archivo no es común para todos los proyectos, sino que debe ser único para cada proyecto y placa, teniendo configuraciones distintas.

3. Configuración de Tracealyzer

Para integrar el grabador Tracealyzer, se utilizó el manual de usuario de la aplicación en la sección "**Integrating the Recorder**", se realizaron los siguientes pasos:

- Usar una versión de FreeRTOS 7.3 o superior.
- Descargar la librería de Trace Recorder.
- Copiar los **3 archivos .c** a la carpeta de nuestro proyecto, estos archivos son:
  - `trcKernelPort.c`
  - `trcSnapshotRecorder.c`
  - `trcStreamingRecorder.c`

- Copiar los **3 headers** de configuración en la carpeta **inc/tracealyzer** creada en el proyecto. Estos headers son:
    - `trcConfig.h`
    - `trcSnapshotConfig.h`
    - `trcStreamingConfig.h`
- Copiar los **4 headers** de la carpeta include en la carpeta **inc/tracealyzer**. Estos headers son:
    - `trcHardwarePort.h`
    - `trcKernelPort.h`
    - `trcPortDefines.h`
    - `trcRecorder.h`
- Editar el archivo **trcConfig.h** para reflejar el sistema y las características a utilizar.
    - `#define TRC_CFG_HARDWARE_PORT  TRC_HARDWARE_PORT_ARM_Cortex_M`
    - `#include "LPC17xx.h"`
    - `#define TRC_CFG_RECORDER_MODE TRC_RECORDER_MODE_SNAPSHOT`
    - `#define TRC_CFG_FREERTOS_VERSION TRC_FREERTOS_VERSION_10_00`
    - `#define TRC_CFG_INCLUDE_MEMMANG_EVENTS 0`
    - `#define TRC_CFG_INCLUDE_READY_EVENTS 0`
    - `#define TRC_CFG_INCLUDE_OSTICK_EVENTS 0`

Para reflejar la versión usada de FreeRTOS, el hardware utilizado, desactivar tracing de eventos como manejo de memoria, tareas listas y ticks del SO.

Además, se configura el modo **SNAPSHOT**, que consiste en que los datos de trace se grabaran en una zona de memoria en la placa, para que el usuario, en algún momento,

pueda leer esa zona de memoria (tomar un snapshot) y analizar la ejecución del sistema hasta ese punto.

- Definir en `FreeRTOSConfig.h`: **`#define configUSE_TRACE_FACILITY 1`**
- Insertar en `FreeRTOSConfig.h`:

```
109 /* Integrates the Tracealyzer recorder with FreeRTOS */
110 #if ( configUSE_TRACE_FACILITY == 1 )
111 #include <inc/tracealyzer/trcRecorder.h>
112 #endif
```

- Si se usa una placa ARM Cortex-M, incluir en **`trcConfig.h`** la librería CMSIS de la placa, es decir, **`#include "LPC17xx.h"`**.

- Finalmente, llamar a la función **`vTraceEnable(TRC_START)`** en la función `main` para iniciar el tracing y la grabación de los datos en la memoria. Se debe hacer *DESPUÉS* de inicializar el hardware de la placa y *ANTES* de cualquier llamada a RTOS(tareas, etc).

Para obtener los datos que se utilizan en tracealyzer, se debe indicar la dirección de memoria donde empiezan los datos de trace agregando un watch en la variable **`RecorderDataPtr`**, obteniendo el valor de memoria en donde se encuentra. Desde esta dirección de memoria se hace el dump.

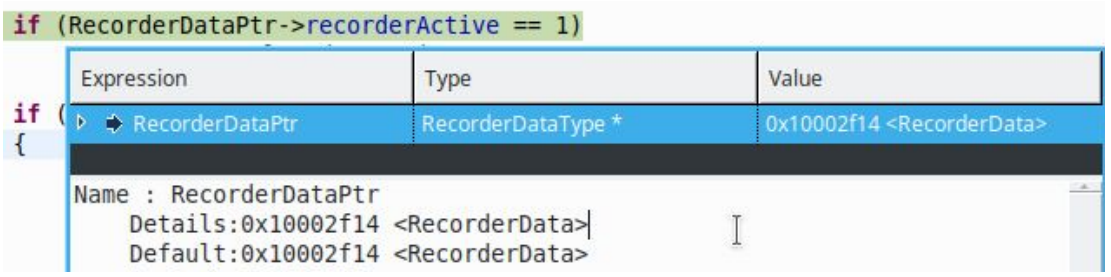

Figura 7: RecorderDataPtr - Dirección de Memória

Luego con la herramienta Memory Monitors, se agrega un monitor a esa variable y nos muestra el mapa de memoria.

Figura 8: Monitor de la variable

Con la opción export indicamos un formato binario el nombre del archivo y el tamaño.

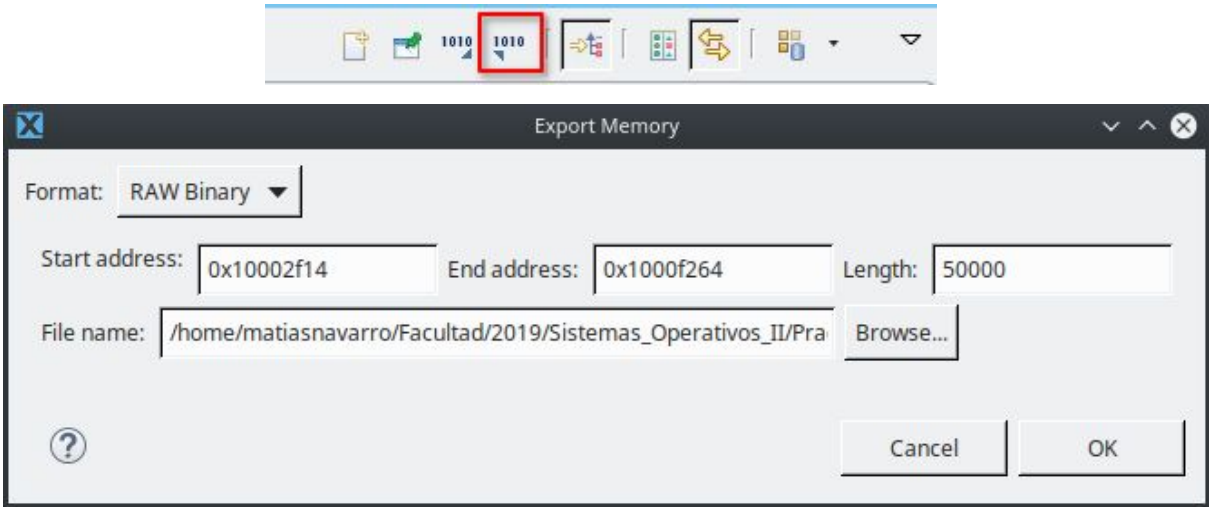

Figura 9: Export Memory

Los archivos binarios generados. Estos se abren con la herramienta Tracealyzer y se llevó a cabo los siguientes análisis.

Análisis con Tracealyzer

En ambos casos se abre el archivo binario desde Tracealyzer (File, open file) y se detalla el análisis a continuación.

Productor-Consumidor

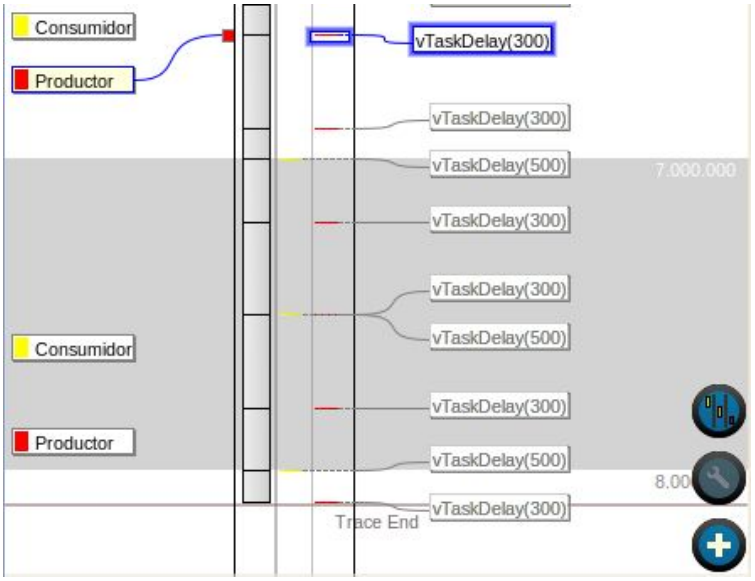

Figura 10: Tareas de Productor/Consumidor

Se puede observar en la Figura 10 las tareas que se ejecutaron alternadamente, cada una de las dos tareas mostrando el delay que hacen cuando finalizan la escritura/lectura del `buffer_size`. Ambas tareas muestran un delay correspondiente al que se declararon en el `.c`. La *tarea IDLE* es la que ocupa el mayor tiempo de ejecución dado que las tareas productor y consumidor se realizan relativamente rápido y están dormidas mayor parte de su tiempo.

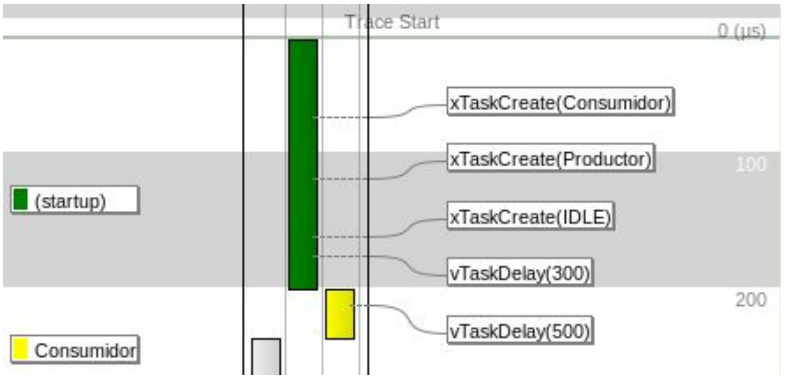

Figura 11: Startup

Al comienzo, se observa en la Figura 11 en verde al startup que se ejecuta en un lapso de tiempo mayor donde se produce la creación de las tareas *consumidor*, *productor* e *IDLE*. Posterior a la creación de estas se comienza a ejecutar la tarea consumidor.

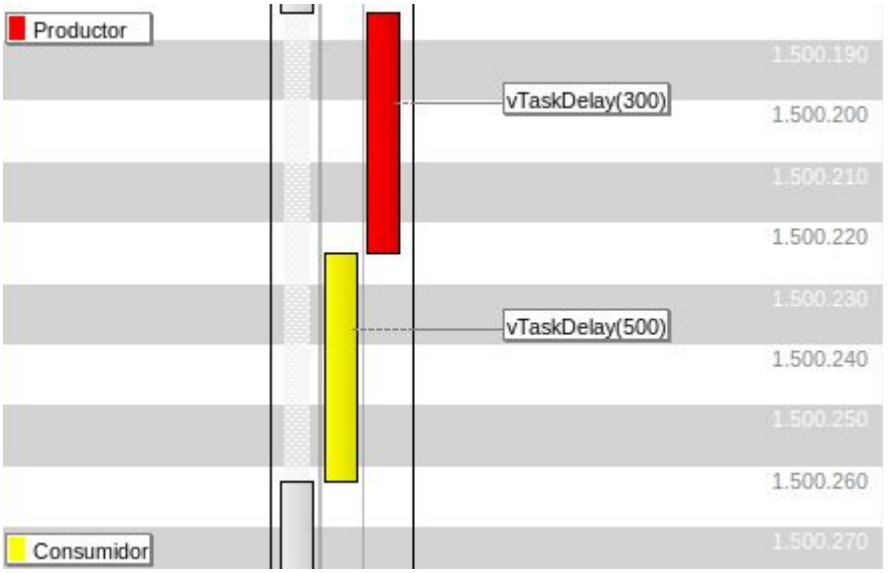

Figura 12: Tareas (Prioridades)

En la figura 12 se observa que la tarea de mayor prioridad (*Productor*) termina de ejecutarse y recién ahí se comienza a ejecutar la de menor prioridad (*Consumidor*). La prioridad del consumidor es de 1 mientras que la del productor es de 3.

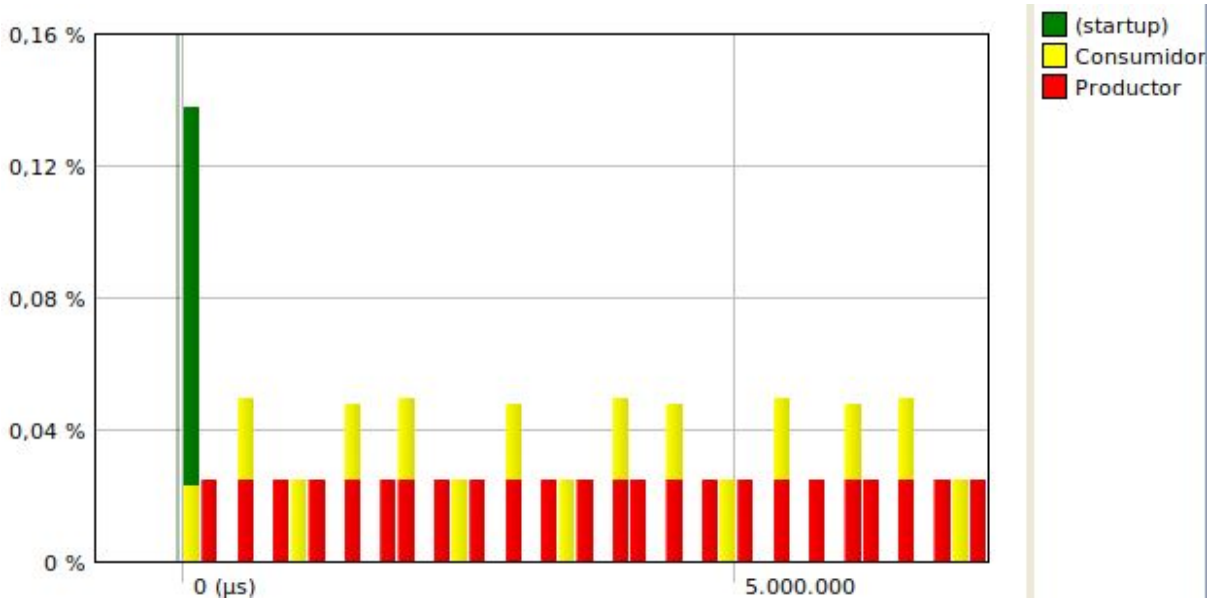

Figura 13: Uso de CPU (Flag_prog = 1)

En la figura 13 se observa el uso de la CPU por parte de las tareas, en el comienzo la tarea que mayor consume es la de startup mencionada con anterioridad, 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.

Dos productores y un consumidor:

Figura 14: Terminal de Putty

En la Figura 14 se puede observar la salida mediante la terminal Putty. El rango de temperatura varía aleatoriamente entre 00 y 99, mientras que los mensajes que pueden aparecer son los siguientes:

{"ACK", "Mate", "Termo", "Yerba", "Criollos", "Asado", "Pesca", "Folklore", "Tradicion", "Argentina"}.

Aunque la tarea de mayor prioridad es la de la escritura de mensajes se observa mayores escrituras de temperatura debido que estas tienen un tiempo de delay mucho menor.

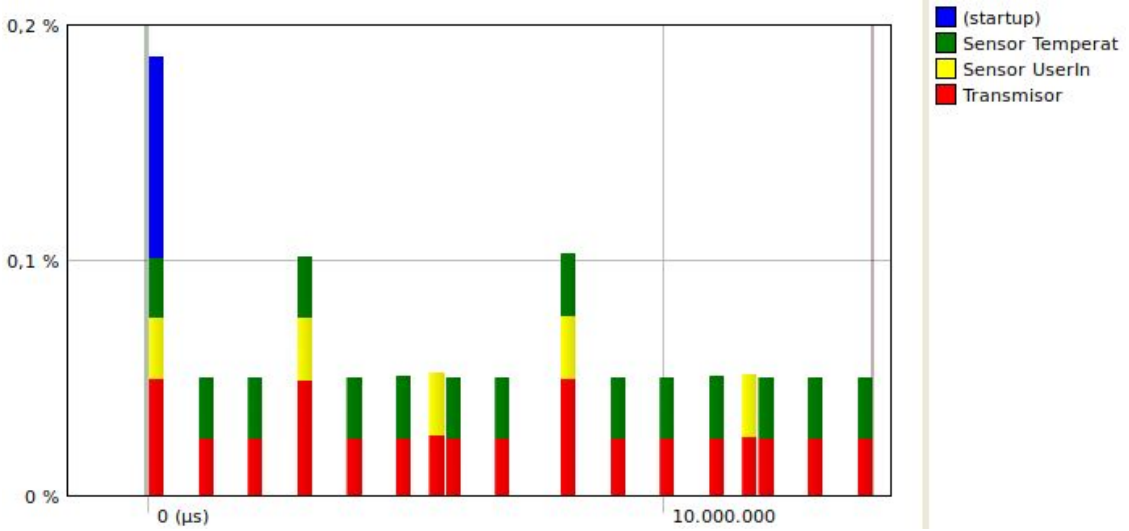

Figura 15: Uso de CPU (Flag_prog = 0)

En la figura 15 se observa el uso de la CPU por parte de las tareas, en el comienzo la tarea

que mayor consume es la de startup, en el transcurso del tiempo las tarea de transmisor es la que mayor tiempo consumo de procesador debido que esta posee mayor prioridad, en segundo lugar la de temperatura y por último la del usuario. Para destacar la irregularidad con la que se ejecuta esta última debido que es aperiódica.

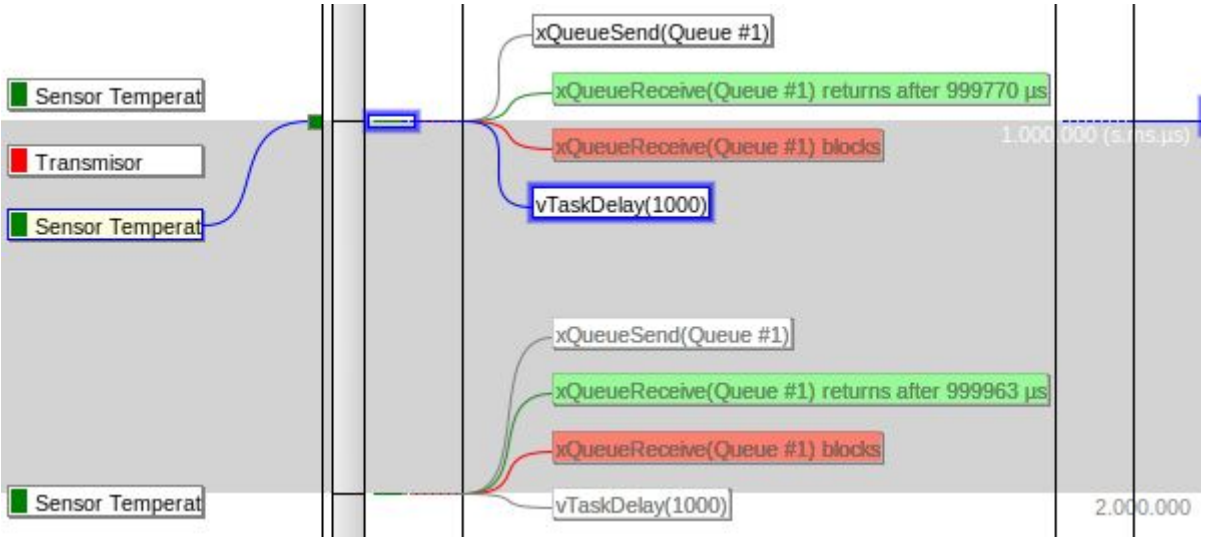

Figura 16: Tarea periódica

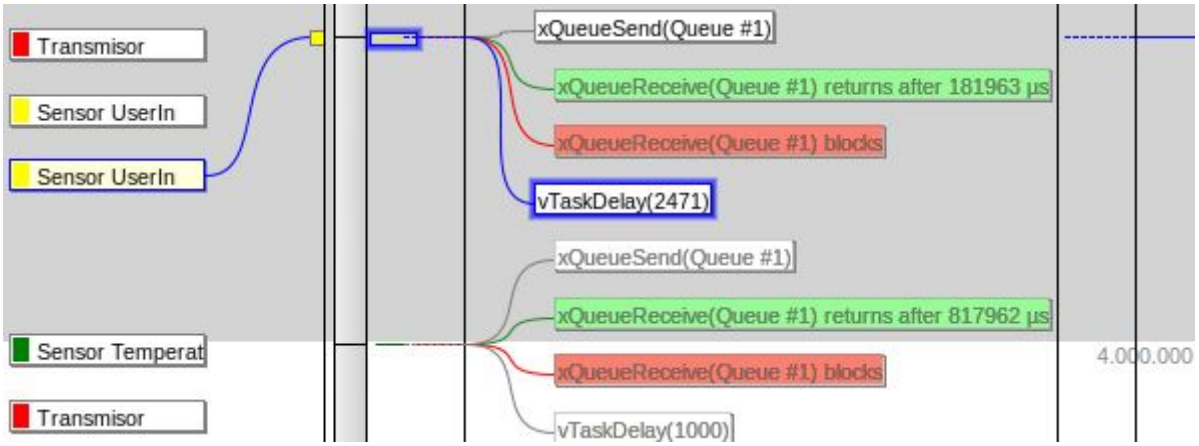

Figura 17: Tarea aperíodica

Se puede observar en la Figura 16 el comportamiento periódico de la tarea de temperatura, mientras que en la figura 17 se observa el comportamiento aperiódica de la función de usuario.

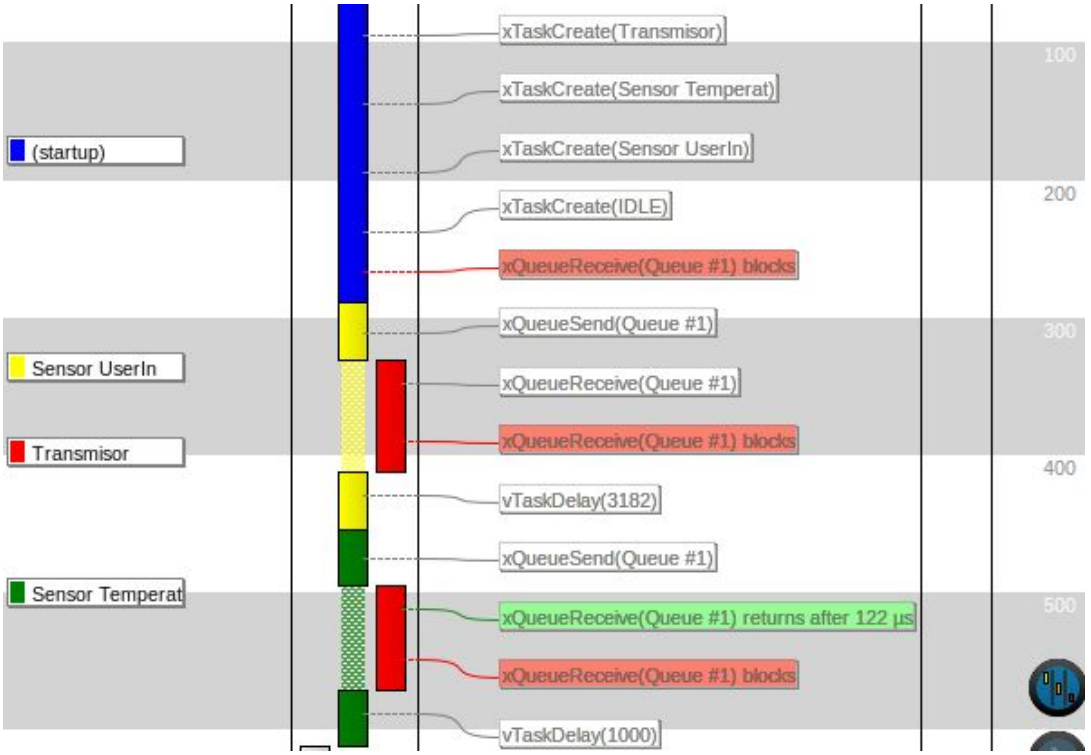

Figura 18: Uso de CPU (Task0)

En la figura 18 se observa el inicio con la creación de las 3 tareas, intenta ejecutarse la tarea de transmisión de datos, pero se bloquea debido a que no hay nada en la cola. Se ejecuta la tarea de usuario, debido a que tiene mayor prioridad que la del usuario. A medida que se ejecuta la tarea de transmisión de strings esta es interrumpida por la transmisión debido que tiene mayor prioridad. Luego se ejecuta la tarea de temperatura y sucede lo mismo.

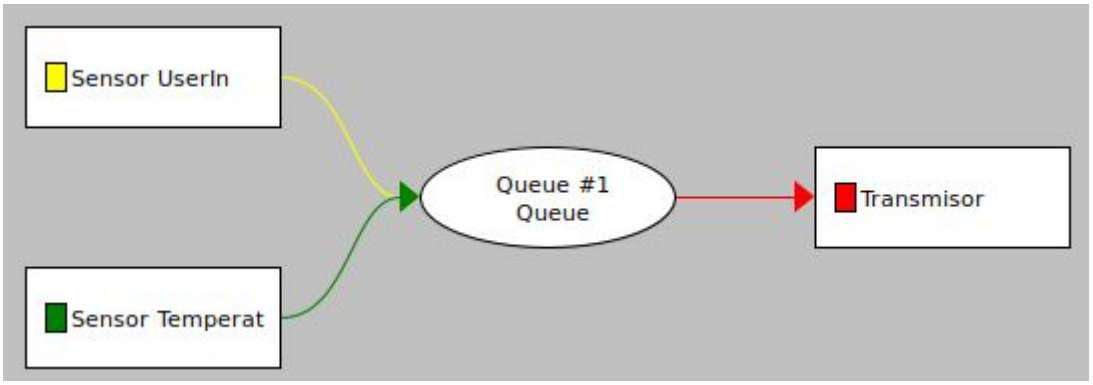

Figura 19: Flujo de datos

En la figura 19 se puede observar el flujo de datos a través de las tareas, mostrando que ambas tareas del sensor producen información y la envían a la cola, mientras que la tarea transmisor consume esta información de la cola.

Conclusión

Con la realización del trabajo se aprendió a manejar un sistema operativo en tiempo real, crear distintas clases de tareas, organizarlas y manipular sus características.

Además se trabajó con estructuras de control propias del RTOS, como las colas, que permiten la comunicación entre tareas de manera simple y segura. Por último, se trabajó con memoria dinámica, su cuidadoso manejo y se observaron los problemas que surgen de una mala gestión de la misma.