Trabajo práctico Nº IV - FreeRTOS

CÁTEDRA DE SISTEMAS OPERATIVOS II

Casabella Martin, 39694763
martin.casabella@alumnos.unc.edu.ar

18 de junio de 2019

Índice

Introducción

Toda aplicación de ingeniería que posea requerimientos rigurosos de tiempo, y que este controlado por un sistema de computación, utiliza un Sistema Operativo de Tiempo Real (RTOS, por sus siglas en ingles).

Una de las características principales de este tipo de SO, es su capacidad de poseer un kernel preemptive y un scheduler altamente con
gurable. Numerosas aplicaciones utilizan este tipo de sistemas tales como radares, satélites, etc. lo que genera un gran interés del mercado por ingenieros especializados en esta área.

Objetivo

El objetivo del presente trabajo practico es que el estudiante sea capaz de disenar, crear, comprobar y validar una aplicación de tiempo real sobre un RTOS.

RTOS: conceptos principales

Un RTOS (Real Time Operating System) es un programa que se encarga de:

- Ordenar con precisión el tiempo de ejecución de las tareas
- Administrar los recursos del sistema como tiempo de uso de
- procesador, memoria, etc.
- Proveer una base consistente para el desarrollo del código

El RTOS crea la ilusión de múltiples tareas ejecutándose en simultáneo.

El RTOS se situa entre la capa BSP (Board Support Package, o "port") y la capa de aplicación. Puede incluir varios módulos (protocolos de red, sistema de archivos, etc.):

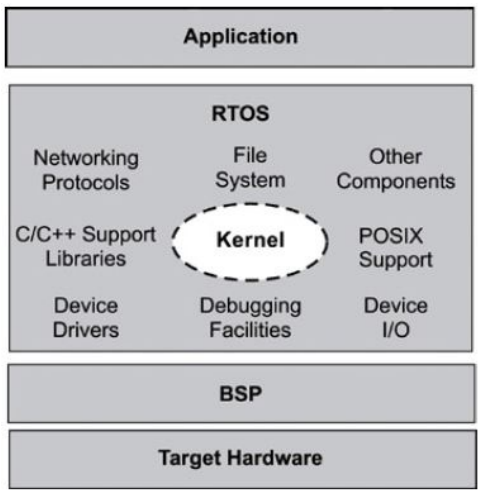

Figura 1: **RTOS en el sistema**

Los componentes de un RTOS pueden clasificarse ampliamente en 3 grupos:

- ★ **Scheduler:** maneja los hilos de ejecución de las tareas.
- ★ **Objetos:** tareas, colas, semáforos, etc.
- ★ **Servicios:** operaciones realizadas sobre los objetos (manejo de interrupciones, de memoria, etc.)

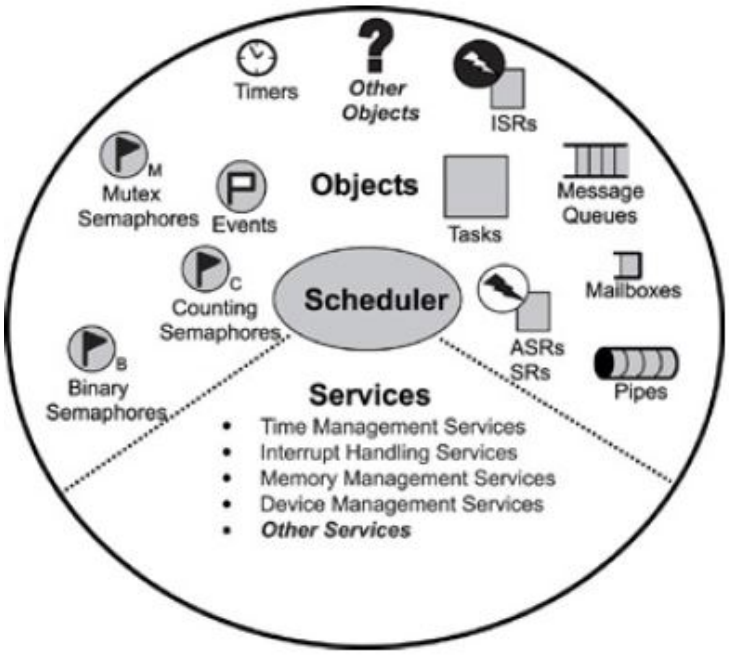

Figura 2: Componentes de un RTOS

Scheduler

El Scheduler determina cuando se ejecutar cada tarea. Existen diferentes esquemas de scheduling:

- **Cooperativo:** La tarea en ejecución cede el uso de CPU a otra voluntariamente.
- **Preemptive:** La tarea en ejecución cede el uso de CPU a otra por orden del scheduler.
  - Priority-Based: Se asignan prioridades a las tareas para acceder al uso de CPU.

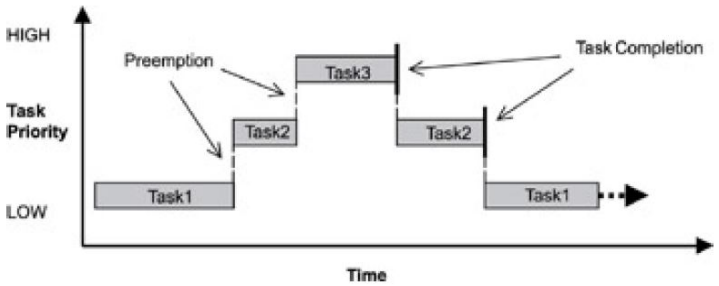

Figura 3: Ejemplo de esquema Priority Based

- Round-Robin: Se asigna un tiempo
  jo de uso de CPU a cada tarea en orden circular. Puede combinarse con el uso de prioridades.

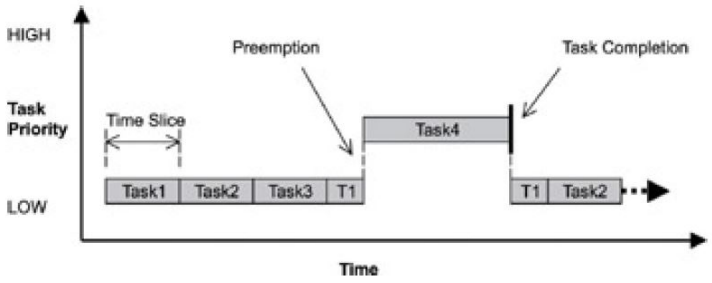

Figura 4: Ejemplo de esquema Round Robin con prioridades

Tareas

Una tarea es un hilo de ejecución independiente que puede competir con otras tareas por tiempo de ejecución. Pueden ser creadas y eliminadas en tiempo de ejecución.

Se componen por:

- Nombre/ID
- Prioridad (esquema preemptive)
- Stack
- Rutina (código)
- Bloque de control

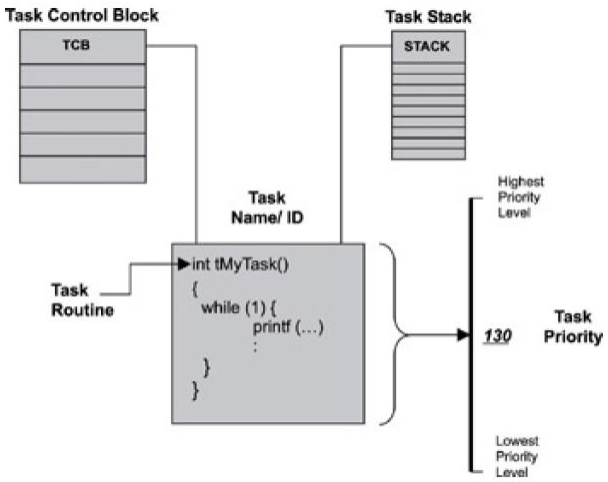

Figura 5: Componentes de una tarea o task

Los estados posibles de una tarea son:

1. Ready: compite por tiempo de ejecución
2. Running: tarea activa
3. Blocked: esperando pasar a Ready (podra activarse ante un evento o cuando pase cierto tiempo)

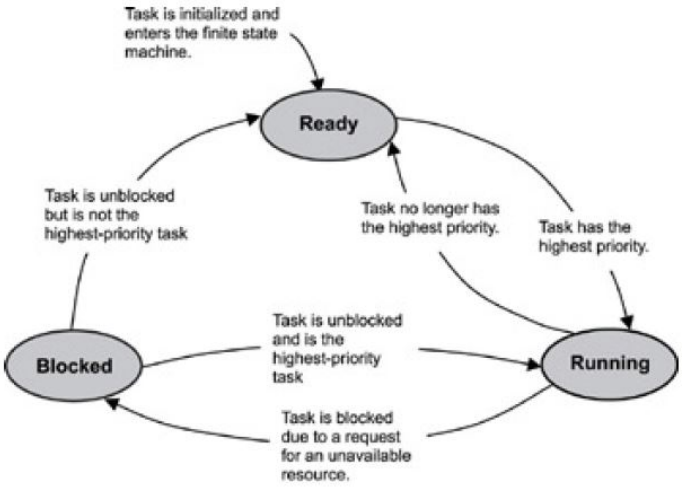

Figura 6: Estados posibles de una tarea

Colas

Su función principal es proveer un mecanismo de intercambio de datos entre tareas. Son FIFO. Se componen por: Nombre/ID, tamaño y tipo de datos a almacenar, bloque de control.

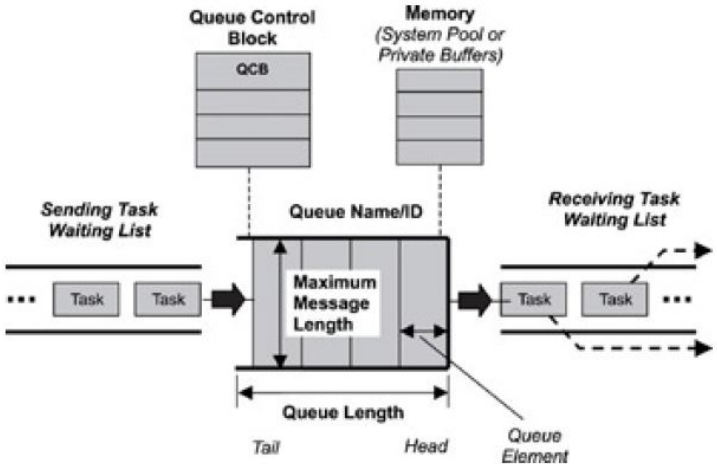

Figura 7: Colas y sus componentes

- Varias tareas pueden acceder a una misma cola.
- Una tarea puede elegir bloquearse si su cola esta vaca. Al llegar un elemento, automáticamente pasa a Ready.
- Cargar datos en una cola causa una copia de los datos en memoria.

FreeRTOS

Porque el uso de FreeRTOS:

- Es de código abierto
- Código ampliamente comentado
- Sencillo de portar (existen mas de 23 ports)
- Ocupa poco espacio en ash (5KB) necesita poca RAM (5KB + Heap) y el overhead que introduce es mínimo (entre 1 % y 4 % del tiempo de CPU) a cambio de una gran utilidad).
- Ampliamente documentado
- Existe una comunidad de usuarios importante
- Libre de regalas. Puede ser usado en aplicaciones comerciales bajo licencia GNU versión 2.

FreeRTOS y sistemas embebidos

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.

Permite que las aplicaciones se organicen como una colección de hilos de ejecución independientes. Como la mayoría de los microcontroladores (como por ejemplo, Cortex-M3) tienen un solo núcleo, en realidad solo un hilo puede ejecutarse a la vez.

El kernel decide qué hilo se debe ejecutar al examinar la prioridad asignada a cada hilo por el diseñador de la aplicación. En el caso más simple, el diseñador de la aplicación podría asignar mayores prioridades a hilos que implementan requisitos *hard real-time*l, y menores prioridades a hilos que implementan requisitos *soft real-time*.

Esto aseguraría que los hilos *hard real-time* siempre se ejecuten por sobre los hilos *soft real-time*, pero las decisiones de asignación de prioridad no siempre son tan simplistas.

Se detallan conceptos introducidos previamente, que serán utilizados en el trabajo.

Tasks y freeRTOS

Las tasks se implementan como funciones en C. La única característica especial recae en su prototitpo, que debe devolver y tomar como argumento un *void pointer*.

Cada tarea es un programa pequeño en si, 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 sus funciones que las implementan (i.e, no debe haber un *return*. Si no se necesita mas cierta tarea, se debe eliminar explícitamente.

Una única definición de la función que implementa una tarea puede crear cualquier numero de tareas desde la misma, y cada tarea sera una instancia separada de ejecución con su propia pila y su propia copia (automática) al stack de las variables definidas para la tarea que las creo.

Si el microcontrolador corriendo la aplicación contiene un solo core, entonces una tarea puede estar en ejecución en cierto instante de tiempo. Esto implica que los estados de la misma son dos: Running y Not Running. Los demas estados, son subestados de cada uno de estos dos estados principales.

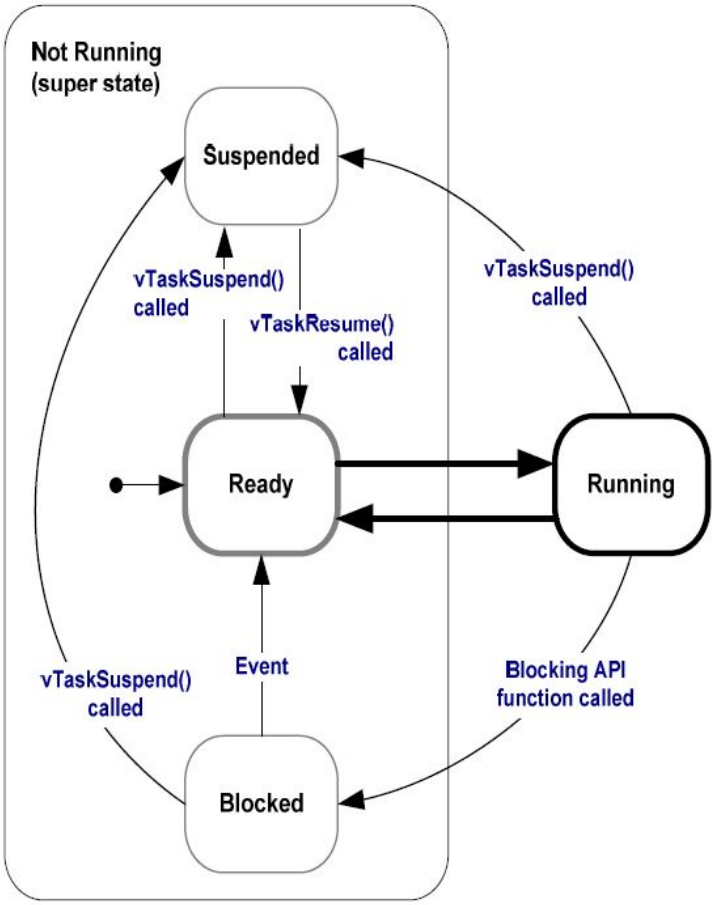

Figura 8: **Maquina de estados completa de las tareas**

- Cuando una task esta en *Running* el procesador esta ejecutando su codigo
- Cuando la task esta en *Not Running*, permanece inactiva, y su estado fue guardado para que pueda retomar la ejecución la próxima vez que el scheduler lo decida

Una tarea que paso de no ejecutando a en ejecución, se le dice cambiada o *switched* (hay herramientas de Profiling miden estos cambios).

IDLE task

El procesador siempre necesita algo para ejecutar, y por ende, debe haber al menos una tarea en estado *Running*. Para asegurar esto, se crea automáticamente por el scheduler una *IDLE task*.

Esta tarea no hace mucho mas que estar en un loop, y se le asigna la prioridad mas baja, para evitar que no permita a una tarea de mayor prioridad, ejecutarse, y para que también ni bien una tarea de mayor prioridad pase al estado *Ready*, se realice inmediatamente la transición de estados.

Queues y freeRTOS

Las aplicaciones que utilizan FreeRTOS están estructuradas como un conjunto de tareas independientes: cada tarea es efectivamente un mini programa por derecho propio.

Es probable que estas tareas autónomas tengan que comunicarse entre sí para que, de forma colectiva, puedan proporcionar una funcionalidad útil del sistema.

La cola es la primitiva subyacente utilizada por todos mecanismos de comunicación y sincronización freeRTOS.

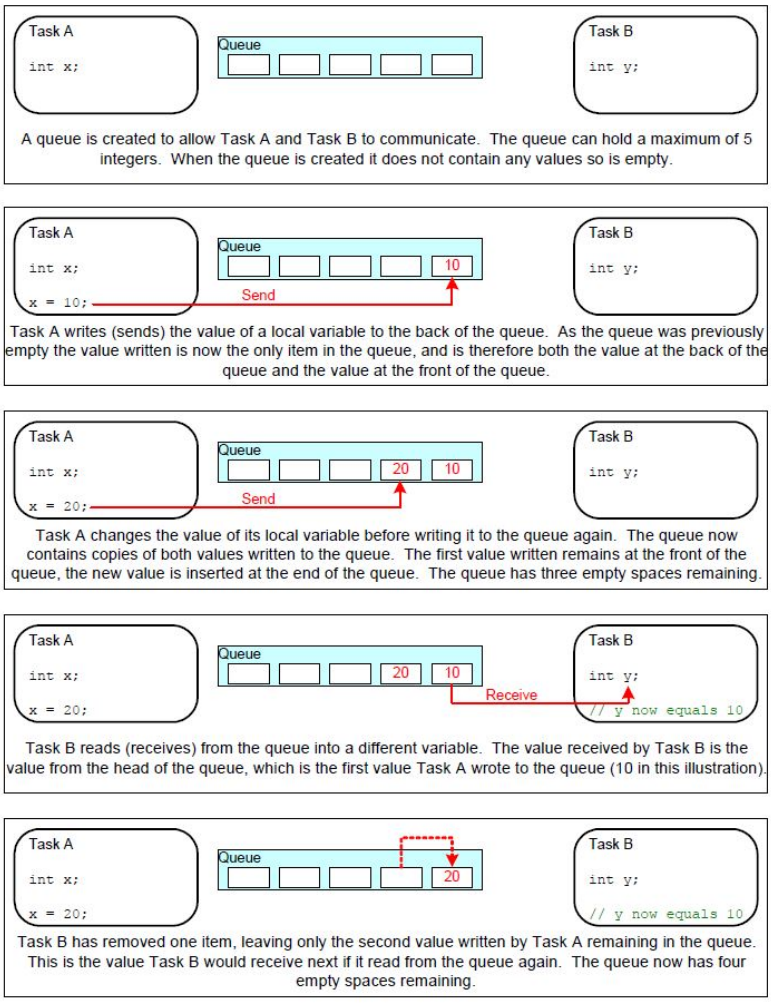

Figura 9: Ejemplo de secuencia de lectura y escrituras

- Una cola puede tener un numero finito de elementos de tamano fijo
- El tamano y la longitud de la cola se establecen cuando la misma es creada
- Las colas se utilizan como buffers FIFO, en donde los datos escritos van al final de la cola (tail), y se remueven desde el frente (head). No obstante, se puede modificar
- Escribir datos en la cola causa una copia byte a byte. Leer de la cola, implica otra copia byte a byte.

LECTURA Y BLOQUEO

Cuando una tarea intenta leer de una cola, puede adicionalmente setear su tiempo de bloqueo. Esto se define como el tiempo que la tarea debe permanecer bloqueada esperando que haya datos disponibles en la cola para leer (i.e, trata de leer y la cola se haya vacía).

Una tarea bloqueada por este motivo, automáticamente pasa a estar lista (*Ready*) cuando otra tarea o alguna interrupción, coloca datos en la cola involucrada. Lo mismo ocurre si expira el tiempo seteado, automáticamente cambia su estado a *Ready*.Siempre se desbloqueara primero aquella de mayor prioridad, y de tener todas la misma prioridad, se despertara a aquella que mas tiempo haya estado esperando.

ESCRITURA Y BLOQUEO

También se puede setear el tiempo (máximo en este caso) que una tarea se bloquea cuando no puede escribir en la cola involucrada. Esto ocurriria si la cola se encuentra llena en el momento que la tarea apunta a escribir un dato.

Al tener muchos escritores, puede ocurrir que mas de una tarea se bloquee, lo que recae en que solo una se desbloqueara cuando haya espacio. Tal como ocurre con la lectura, el criterio de desbloqueo es el mismo.

Acceso a colas por múltiples tareas

Las colas son objetos que existen por si mismo, y no tienen tarea propietaria ni están asignadas a ninguna en particular.

Cualquier numero de tareas puede escribir y leer en la misma cola. De hecho, una cola con múltiples escritores es una necesidad común.

Setup del sistema

El sistema consta por un lado, en el embebido LPC1769 , donde se encuentra instalado FreeRTOS, que ejecuta la aplicacion de tiempo real. Por otro lado, la placa se encuentra conectada a la PC, desde donde se observan diferentes métricas Tracealyzer.

El proyecto se ejecutara en una computadora con Windows como sistema operativo residennte. Existe conectividad entre el embebido y la PC mediante UART. Se utilizo el integrado CP2102 como controlador USB para conectar la placa a la PC.

La PC tiene instalado el software LPCXpresso v8.2.2, y el software Tracealyzer 4 en su versión de prueba. La versión de FreeRTOS instalada, es la v10.0.1.

Se instalo ademas, un plugin para Eclipse (LPCXpresso esta basado en eclipse) para Tracealyzer 4, que permite que el software de profiling lea los datos rastreados utilizando la interfaz de *debug* normal, permitiendo que cualquier dispositivo y *Debug probe* que el IDE soporte emplee este plugin.

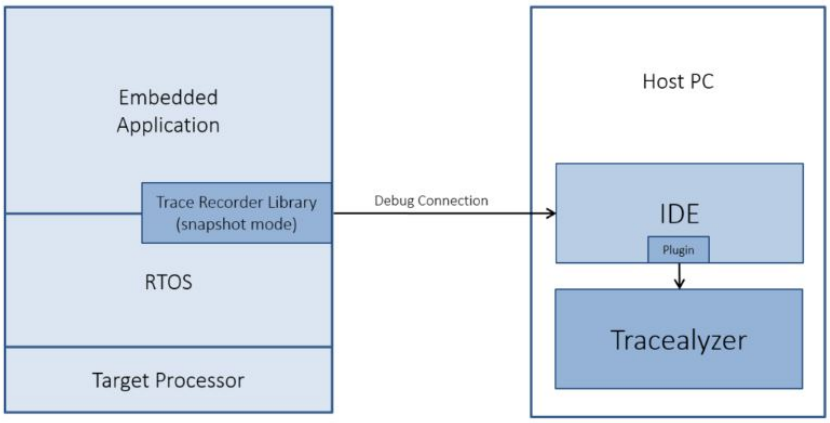

Figura 10: **Debug y tracealyzer con el plugin instalado**

Programa con dos tareas simples

Se utilizo un ejemplo de los anexados en la bibliografia utilizada, que consiste de dos tareas simples, y una cola. Una de las tareas es la emisora de mensajes, mientras que otra es la receptora.

Este programa se utilizo para familiarizarse con Tracealyzer y sus funcionalidades.

Vemos en primera instancia, la interacción en el tiempo entre las tareas y demás eventos intermedios:

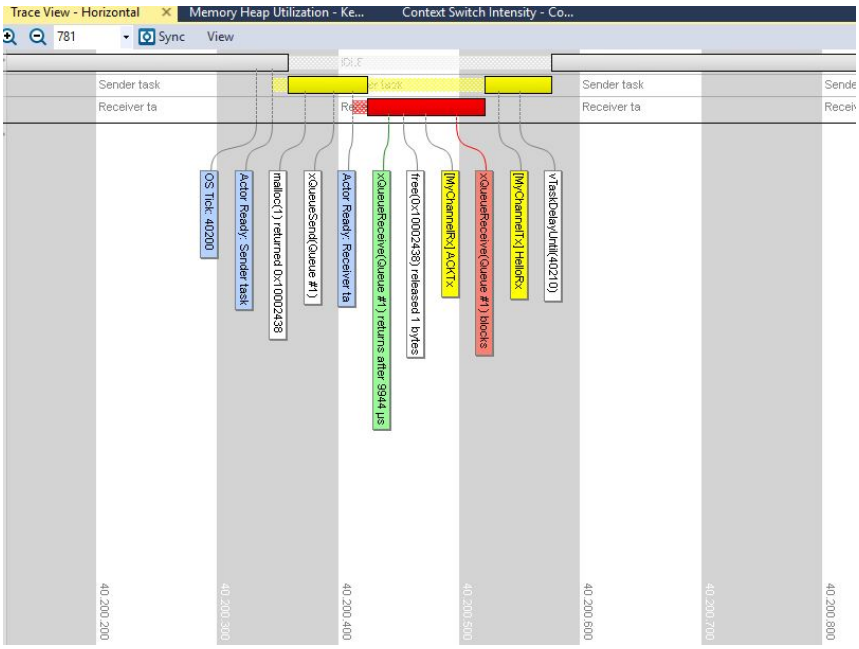

Figura 11: **Seccion de la traza capturada**

En cuanto al flujo de comunicación, vemos los actores involucrados junto con los objetos que utilizan:

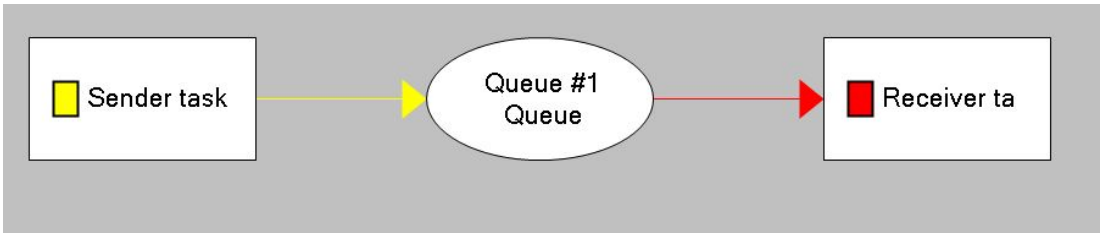

Figura 12: **Flujo de comunicación**

Podemos también analizar el cambio de contexto en la aplicación corriendo:

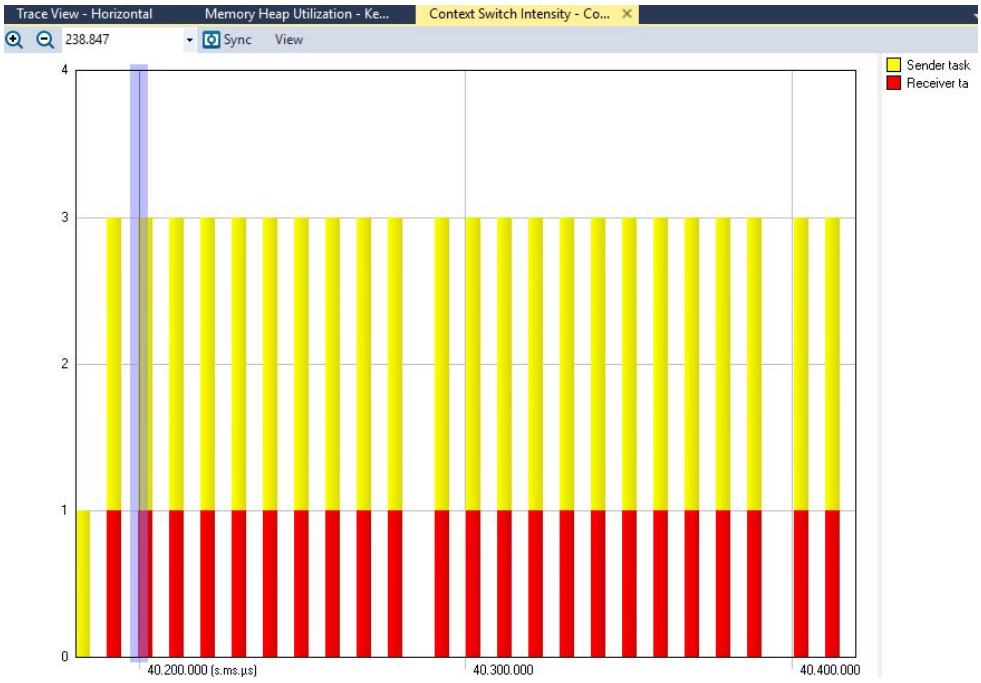

Figura 13: **Context switching**

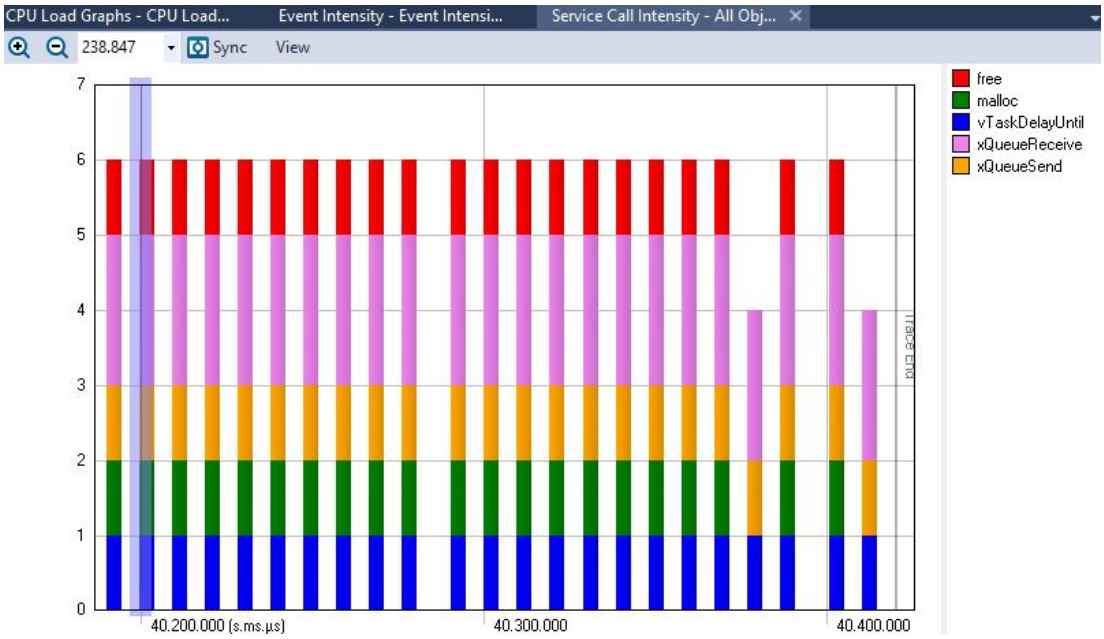

Figura 14: Carga CPU

Figura 15: Intensidad de llamados a servicios

Vemos que en este caso tenemos unicamente una cola (objeto), y podemos ver la utilización de la misma por parte de las tareas corriendo:

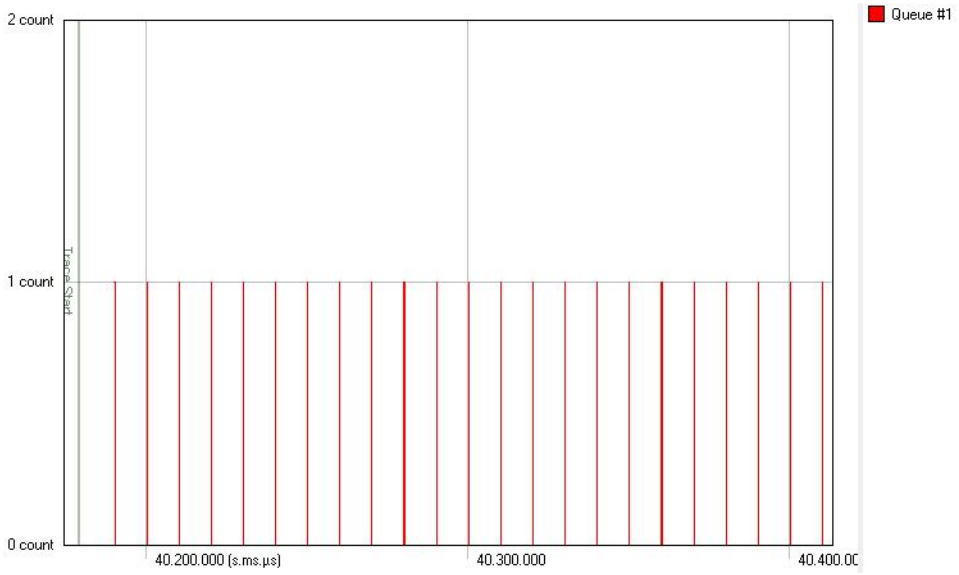

Figura 16: **Utilización de objetos**

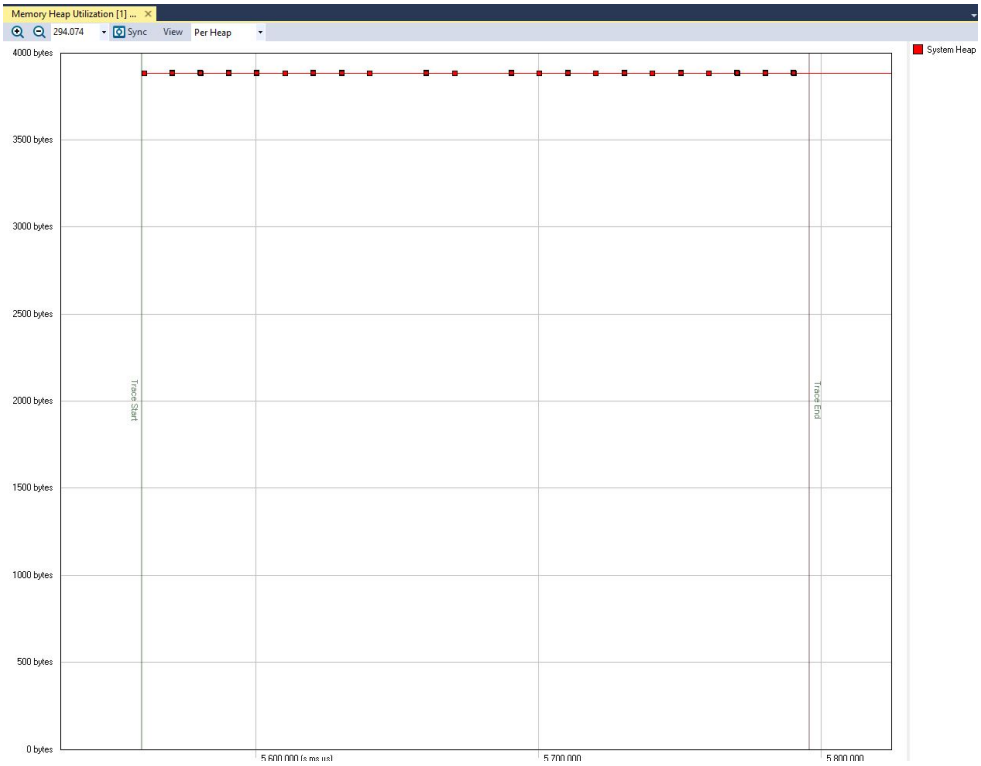

Figura 17: **Memory Heap**

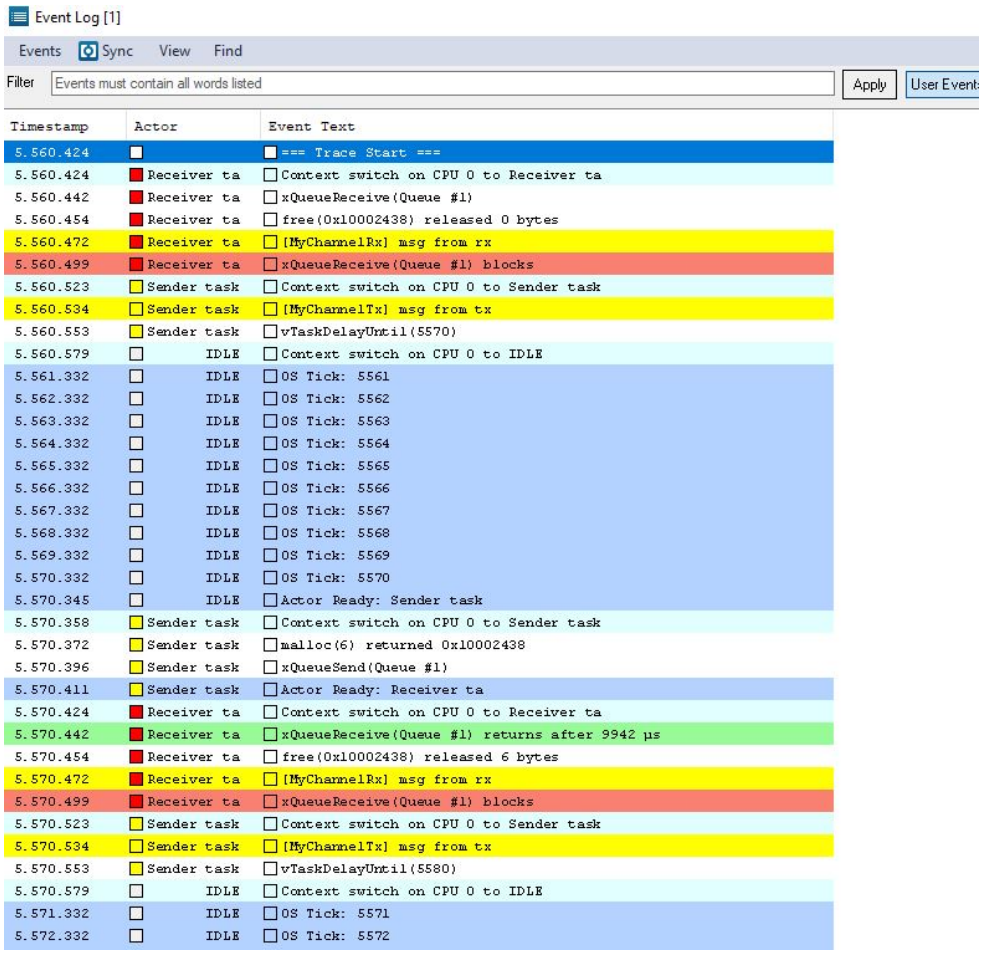

Figura 18: **Event log**

Código

```

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

#include <cr_section_macros.h>
#include <stdlib.h>
/* Kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* Priorities at which the tasks are created. */
#define mainQUEUE_RECEIVE_TASK_PRIORITY    ( tskIDLE_PRIORITY + 2 )

#define mainQUEUE_SEND_TASK_PRIORITY     ( tskIDLE_PRIORITY + 1 )

/* The bit of port 0 that the LPCXpresso LPC13xx LED is connected. */
#define mainLED_BIT              ( 22 )

/* The rate at which data is sent to the queue, specified in milliseconds. */
#define mainQUEUE_SEND_FREQUENCY_MS      ( 10 / portTICK_RATE_MS )

```

```
/* The number of items the queue can hold.  This is 1 as the receive task
will remove items as they are added, meaning the send task should always find
the queue empty. */


#define mainQUEUE_LENGTH          ( 1 )

/*
 * The tasks as described in the accompanying PDF application note.
 */

static void prvQueueReceiveTask( void *pvParameters );
static void prvQueueSendTask( void *pvParameters );
void vConfigureTimerForRunTimeStats( void );

/*
 * Simple function to toggle the LED on the LPCXpresso LPC17xx board.
 */

static void prvToggleLED( void );

/* The queue used by both tasks. */
static xQueueHandle xQueue = NULL;



int main(void)
{
  /* Initialise P0_22 for the LED. */

  LPC_PINCON->PINSEL1 &= ( ~( 3 << 12 ) );
  LPC_GPIO0->FIODIR |= ( 1 << mainLED_BIT );

  /* Init and start tracing */
  vTraceEnable(TRC_START);

  /* Create the queue. */
  xQueue = xQueueCreate( mainQUEUE_LENGTH, sizeof(char)*4 );

  if( xQueue != NULL )
  {
    /* Start the two tasks as described in the accompanying application
    note. */

    xTaskCreate( prvQueueReceiveTask, ( signed char * ) "Rx", configMINIMAL_STACK_SIZE, NULL,
    mainQUEUE_RECEIVE_TASK_PRIORITY, NULL );
    xTaskCreate( prvQueueSendTask, ( signed char * ) "TX", configMINIMAL_STACK_SIZE, NULL,
    mainQUEUE_SEND_TASK_PRIORITY, NULL );

    /* Start the tasks running. */
    vTaskStartScheduler();
  }

  /* If all is well we will never reach here as the scheduler will now be
  running.  If we do reach here then it is likely that there was insufficient
  heap available for the idle task to be created. */

  for( ;; );
}



static void prvQueueSendTask( void *pvParameters )
{
  portTickType xNextWakeTime;

  char* log;

```

```
90    /* Initialise xNextWakeTime - this only needs to be done once. */
91
92    xNextWakeTime = xTaskGetTickCount();
93
94    /*The first step to visualizing custom information that is specific to your application is to
         create a user event
95     *  channel. This is basically a string output channel that allows a developer to add their
        own custom events, called
96     *   User events in Tracealyzer.
97     *
98     *   For example, if I wanted to transmit sensor event data, I would first create the channel
        using the following
99     *    code:  traceString MyChannel = xTraceRegisterString("DataChannel);
100    *
101    *    In case your compiler does not recognize this function, you need to #include
        "trcRecorder."h
102    *
103    *    This function registers a user event channel named DataChannel in the trace. This makes
        Tracealyzer
104    *    show a checkbox for this channel in the filter panel, so you can easily enable/disable
        the display of
105    *     these events. */
106
107   traceString ch = xTraceRegisterString("MyChannelTx");
108
109   for( ;; )
110   {
111     /* Place this task in the blocked state until it is time to run again.
112     The block state is specified in ticks, the constant used converts ticks
113     to ms.  While in the blocked state this task will not consume any CPU
114     time. */
115
116     vTracePrint(ch, "msg from tx");
117     vTaskDelayUntil( &xNextWakeTime, mainQUEUE_SEND_FREQUENCY_MS );
118
119     /* Send to the queue - causing the queue receive task to flash its LED.
120     0 is used as the block time so the sending operation will not block -
121     it shouldn't need to block as the queue should always be empty at this
122     point in the code. */
123     log = pvPortMalloc(sizeof(char)*(rand()%8));
124     xQueueSend( xQueue, &log, 0 );
125   }
126 }
127
128
129 static void prvQueueReceiveTask( void *pvParameters )
130 {
131   char * log;
132   traceString ch = xTraceRegisterString("MyChannelRx");
133
134   for( ;; )
135   {
136
137     vTracePrint(ch, "msg from rx");
138
139     /* Wait until something arrives in the queue - this task will block
140     indefinitely provided INCLUDE_vTaskSuspend is set to 1 in
141     FreeRTOSConfig.h. */
142
143     xQueueReceive( xQueue, &log, portMAX_DELAY );
144
145     /*  To get here something must have been received from the queue, but
146     is it the expected value?  If it is, toggle the LED. */
147
148     vPortFree(log);
149     prvToggleLED();
150   }
151 }
```

```
152
153
154
155 static void prvToggleLED( void )
156 {
157 unsigned long ulLEDState;
158
159   /* Obtain the current P0 state. */
160   ulLEDState = LPC_GPIO0->FIOPIN;
161
162   /* Turn the LED off if it was on, and on if it was off. */
163   LPC_GPIO0->FIOCLR = ulLEDState & ( 1 << mainLED_BIT );
164   LPC_GPIO0->FIOSET = ( ( ~ulLEDState ) & ( 1 << mainLED_BIT ) );
165
166
167 void vConfigureTimerForRunTimeStats( void )
168 {
169 const unsigned long TCR_COUNT_RESET = 2, CTCR_CTM_TIMER = 0x00, TCR_COUNT_ENABLE = 0x01;
170
171   /* This function configures a timer that is used as the time base when
172   collecting run time statistical information - basically the percentage
173   of CPU time that each task is utilising.  It is called automatically when
174   the scheduler is started (assuming configGENERATE_RUN_TIME_STATS is set
175   to 1). */
176
177   /* Power up and feed the timer. */
178   LPC_SC->PCONP |= 0x02UL;
179   LPC_SC->PCLKSEL0 = (LPC_SC->PCLKSEL0 & (~(0x3<<2))) | (0x01 << 2);
180
181   /* Reset Timer 0 */
182   LPC_TIM0->TCR = TCR_COUNT_RESET;
183
184   /* Just count up. */
185   LPC_TIM0->CTCR = CTCR_CTM_TIMER;
186
187   /* Prescale to a frequency that is good enough to get a decent resolution,
188   but not too fast so as to overflow all the time. */
189   LPC_TIM0->PR =  ( configCPU_CLOCK_HZ / 10000UL ) - 1UL;
190
191   /* Start the counter. */
192   LPC_TIM0->TCR = TCR_COUNT_ENABLE;
193 }
```

Programa con dos productores y un consumidor

Ya familiarizado con el software, se procedió a realizar la aplicación principal de este trabajo.

Se debe de implementar 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). Para emular esto, se realizo una función que genera strings de longitud aleatoria, y la misma es utilizada por la tarea aperiódica implementada. Se opto esta solucion para no agregar un periférico como es un teclado, y evitar emergentes adicionales, ya que no es el objetivo del trabajo,

La segunda tarea tarea, es un valor numérico de longitud fi
ja, proveniente del sensor de temperatura del embebido. De la misma manera, se emula esta situacion enviando un valor numérico entre 0 y 255 generado, representando las medidas.

El consumidor, es una tarea que envía el valor recibido a la terminal de una computadora por puerto serie (UART).

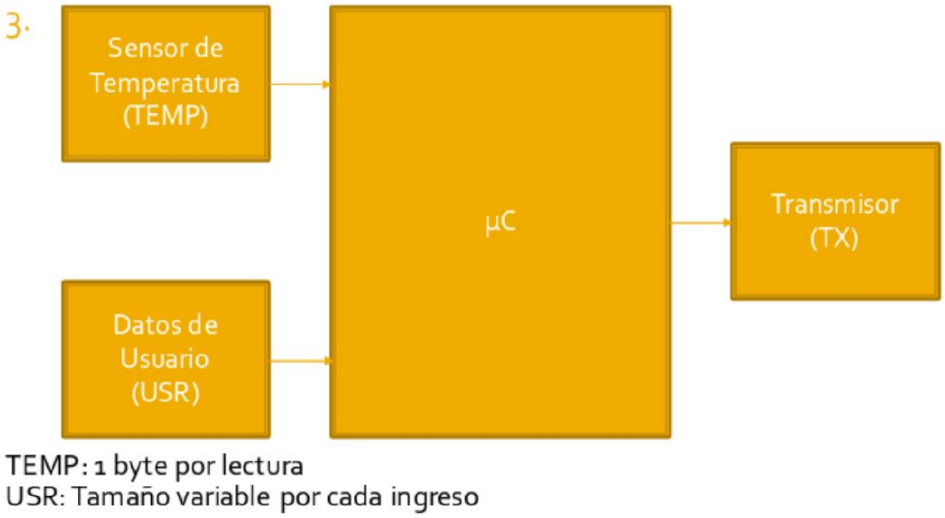

Figura 19: **Arquitectura por componentes del sistema a implementar**

Solución al uso de una sola cola y datos de tamaño variable

La solución a utilizar una sola cola donde ademas, sus elementos no tienen tamaño fijo, se realizo con el uso de estructuras.

Se utilizo una cola para transferir estructuras, donde un campo *char* y otro campo *char* $*$ (*string*) permiten llevar a cabo la implementacion requerida.

La cola entonces alberga estructuras, donde de tratarse de el ingreso del usuario, el campo *msg* tendrá contenido de longitud variable, mientras que el campo *num* no se considera. De tratarse de un dato proveniente del sensor, el campo *msg* ahora torna a ser un *NULL pointer*, y no se considera, permitiendo identificar las diferentes fuentes (o tareas) que escriben en la cola.

```c
struct msg_struct {
    char * msg;
    char num;
    int len_str;
};
```

Luego declaramos una cola de punteros a estructura, y la creamos:

```c
#define MAX_POINTERS     10

/* Declare a variable of type QueueHandle_t to hold the handle of the queue being created
. */
QueueHandle_t xPointerQueue

/* Create a queue that can hold a maximum of MAX_POINTERS pointers*/
xPointerQueue = xQueueCreate(MAX_POINTERS, sizeof(struct msg_struct *));
```

Debug en entorno Semihosted - versión intermedia

Semihosting es un mecanismo que permite que el código que se ejecuta en un destino ARM se comunique y use las facilidades de entrada / salida en una computadora host que ejecuta un depurador.

Ejemplos de estas funciones incluyen entrada de teclado, salida de pantalla y E / S de disco. Por ejemplo, puede utilizar este mecanismo para habilitar funciones en la biblioteca de C, como *printf*() y *scanf*(), para usar la pantalla y el teclado del host en lugar de tener una pantalla y teclado en el sistema de destino.

Se creo un proyecto *semihost*, para primero poder ver la correcta emulación tanto del sensor, como del ingreso del usuario (teclado). Vemos que ambas tareas emulan correctamente al sensor y a la entrada variable:

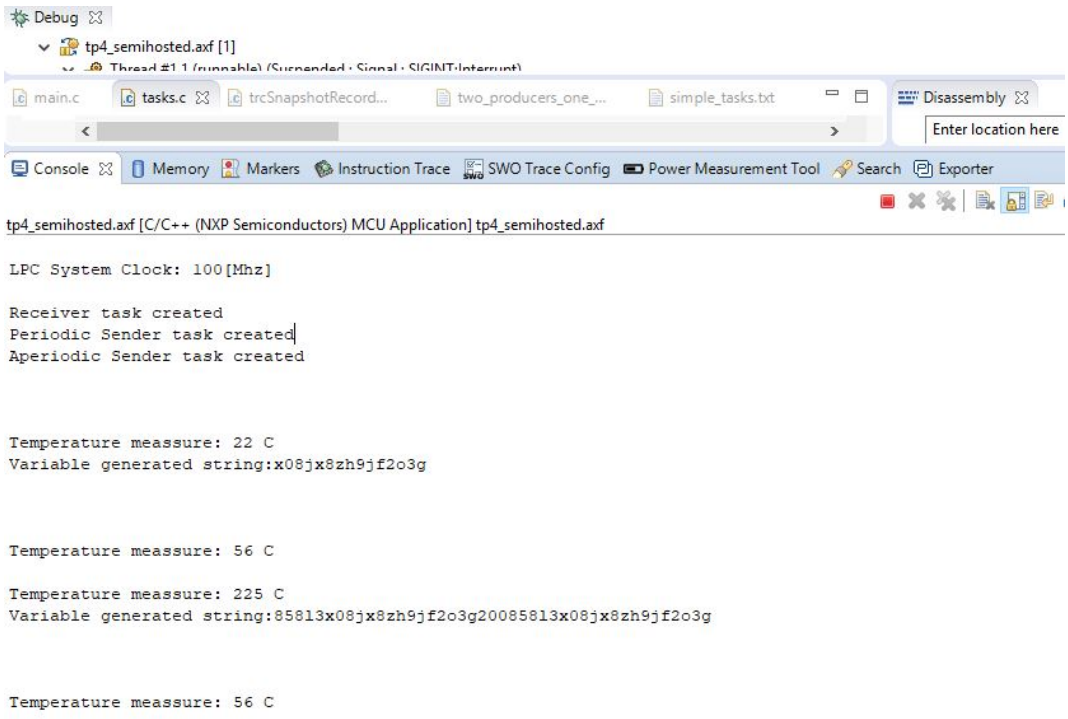

Figura 20: **Prueba de tareas productoras en entorno semihost**

En cuanto a la versión final, se utilizo el mismo código a excepción de una función adicional para que la tarea consumidora pueda enviar los datos via UART.

En esta versión (semihsoted), en lugar de proceder a enviar cada dato que se lee, simplemente se imprimen por pantalla, por ende, la tarea consumidora solo lee y muestra resultados por pantalla.

Versión final

La versión final ya consta de la aplicación receptora interactuando con la PC mediante el envio de los datos leidos a traves del puerto serie.

Prioridades: Si se configura que la tarea consumidora sea la de mayor prioridad, y las dos tareas productoras tengan la misma, la cola nunca tendrá mas que 1 ítem (estructura) en ella.

Esto es causado porque la tarea consumidora se antepone (*pre-empting*) a las tareas productoras ni bien se coloca un dato en la cola.

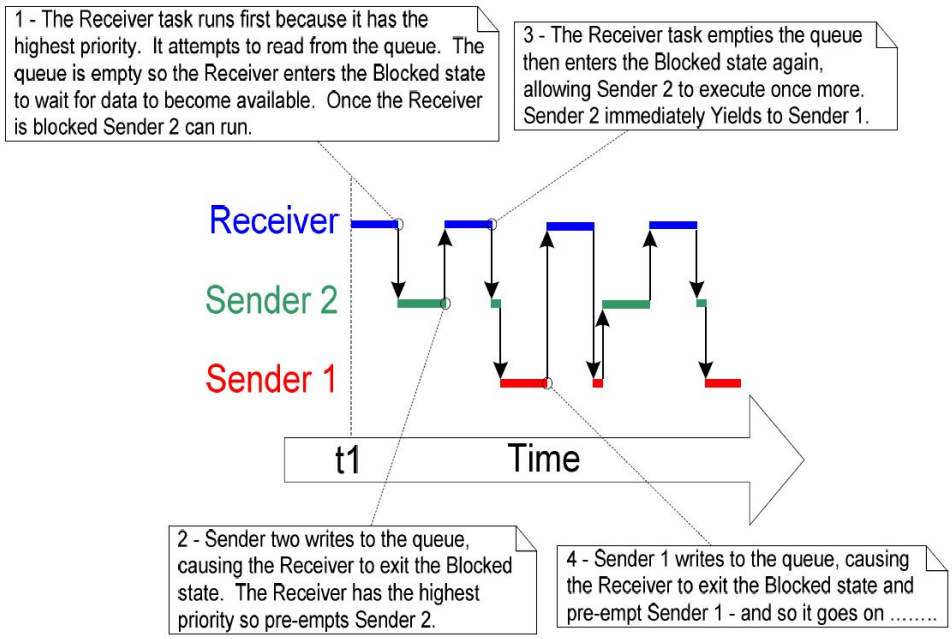

Figura 21: Secuencia de ejecución con tarea consumidora de mayor prioridadd

Por otro lado, si las tareas productoras son las de mayor prioridad, la cola estará normalmente llena.

Esto se debe a que ni bien la tarea consumidora saque un ítem de la cola, se le antepone alguna de las dos tareas productoras que luego vuelve a llenar la cola.

La tarea productora vuelve a bloquearse para asi esperar que se libere espacio y volver a disponer de la cola nuevamente.

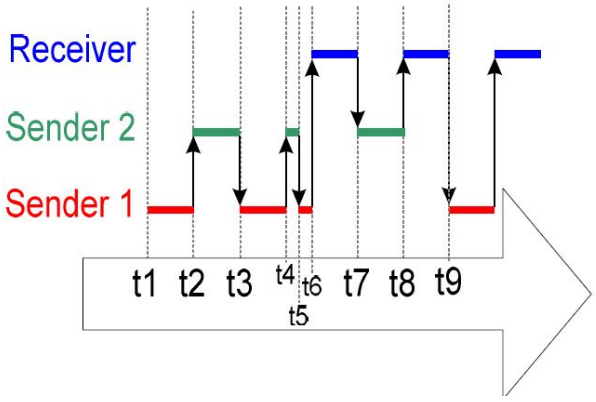

Figura 22: **Secuencia de ejecución con tareas productoras de mayor prioridad**

Se configuro de menor prioridad a la tarea consumidora, mientras que las tareas productoras tienen ambas la misma prioridad. Se procede a mostrar su funcionamiento y luego se agregan los códigos empleados junto con el análisis vía Tracealyzer.

Se tiene un script en Python que configura el puerto con los parámetros necesarios, y se encarga de recibir los datos que la tarea consumidora (Tx en el diagrama en bloques de la arquitectura anexado) envía (hacer zoom a captura en caso que no se vea nitidamente):

```
Run:  freeRTOS_serial ×
C:\Users\kzax\freeRTOS\venv\Scripts\python.exe C:/Users/kzax/freeRTOS/freeRTOS_serial.py
('Puerto abierto: ', 'COM5')
Serial<id=0x2cd5550, open=True>(port='COM5', baudrate=115200, bytesize=8, parity='N', stopbits=1, timeout=None, xonxoff=False, rtscts=False, dsrdtr=False)

3x08jx8zh9jf2o3g20085813x08jx8zh9jf2o3g20085813
250
T$37
x8zh9jf2o3g2008
82
13x08jx8zh9jf2o3g2008
82
13x08jx8zh9jf2o3g20085
119
3x08
171
8zh9jf2o3g2008
82
13x08jx8zh9jf2o3g20085813
250
8jx8zh9
69
2o3g2008
82
13x08jx8zh9jf2o3g20085813x08
171
8zh9jf2o3g20085813x
52
```

Figura 23: **Recepción de datos vía Pyserial**

Análisis con Tracealyzer

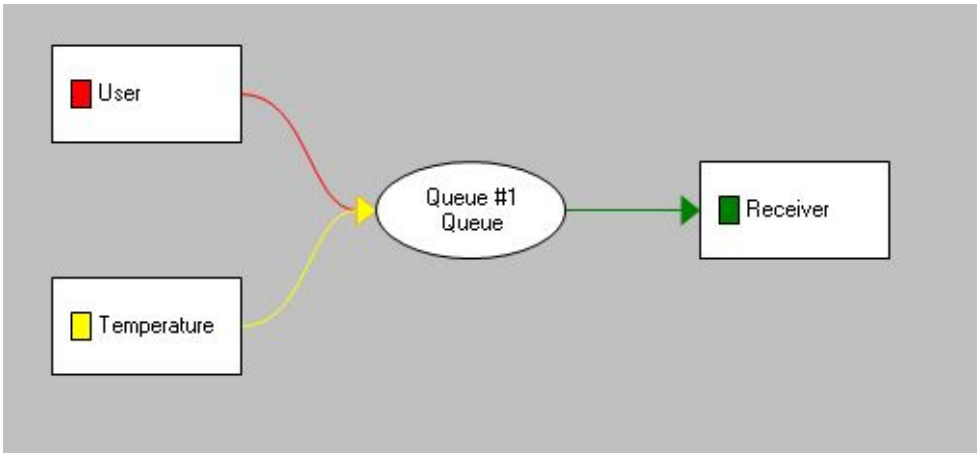

Figura 24: **Actores y objetos involucrados**

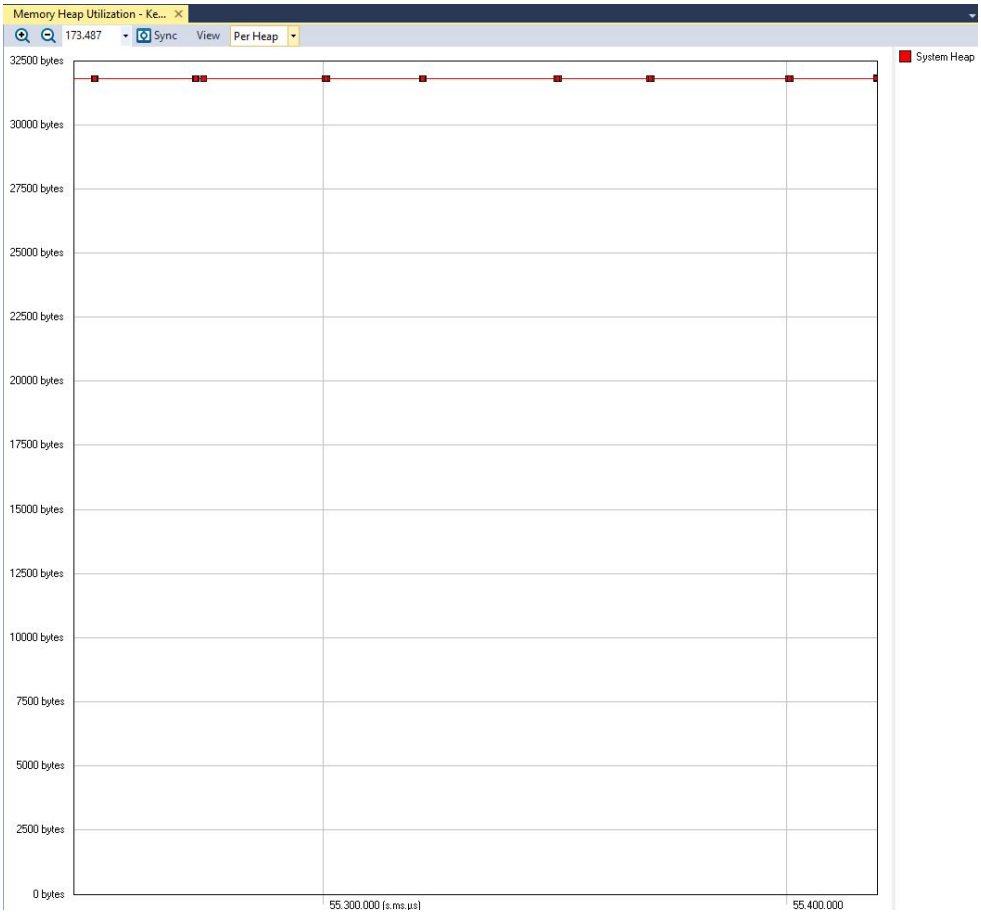

Figura 25: **Memory heap**

Figura 26: Context switching

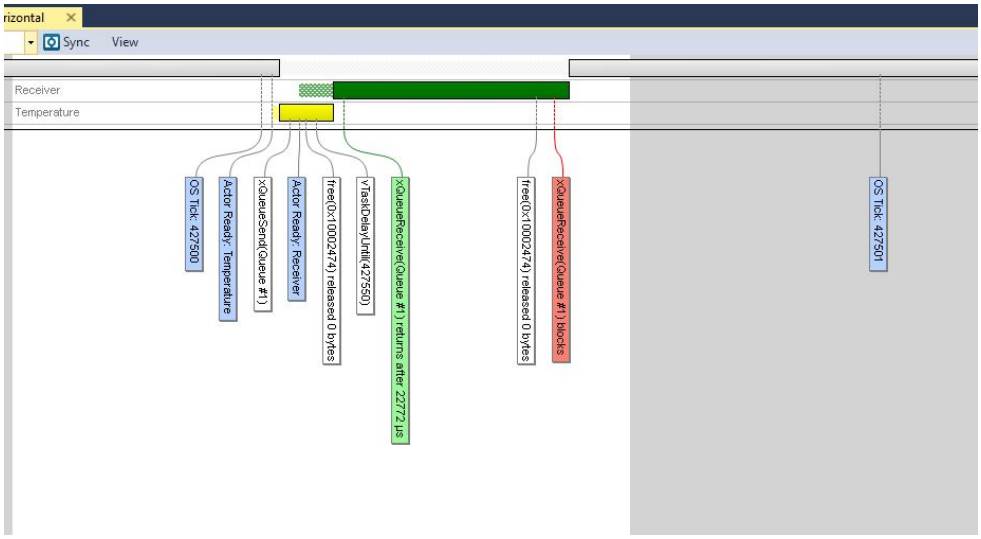

Figura 27: Secuencia temperature - receiver

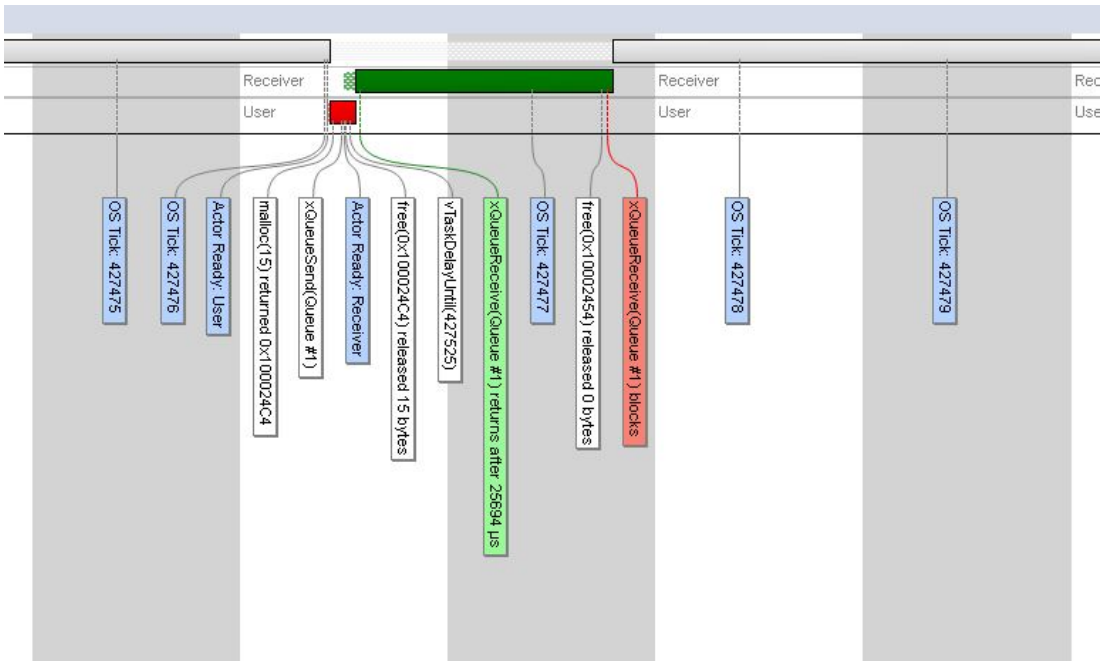

Figura 28: **Secuencia user - receiver**

Figura 29: **Tiempo de ejecución de cada actor**

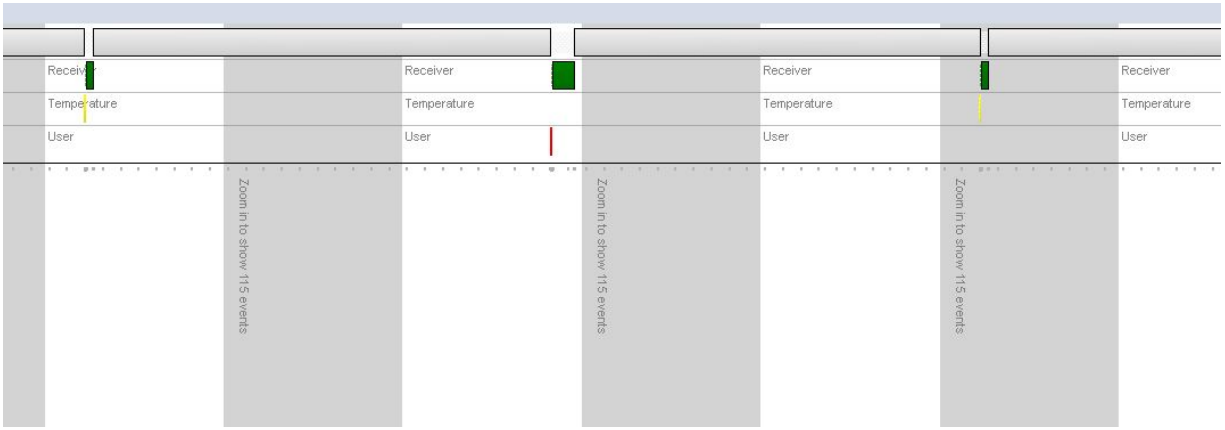

Figura 30: **Interleaving**

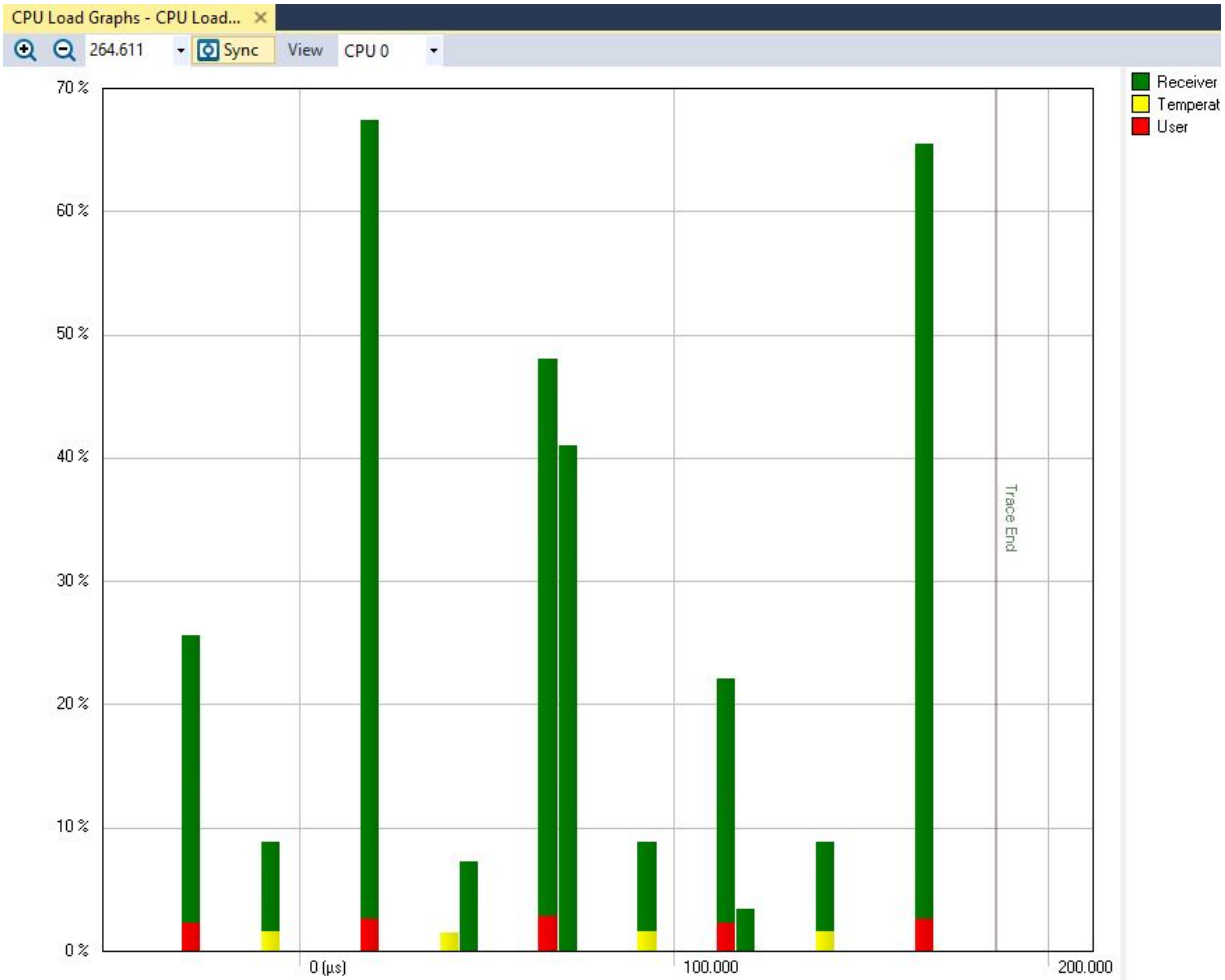

Figura 31: **Grafo de carga del CPU**

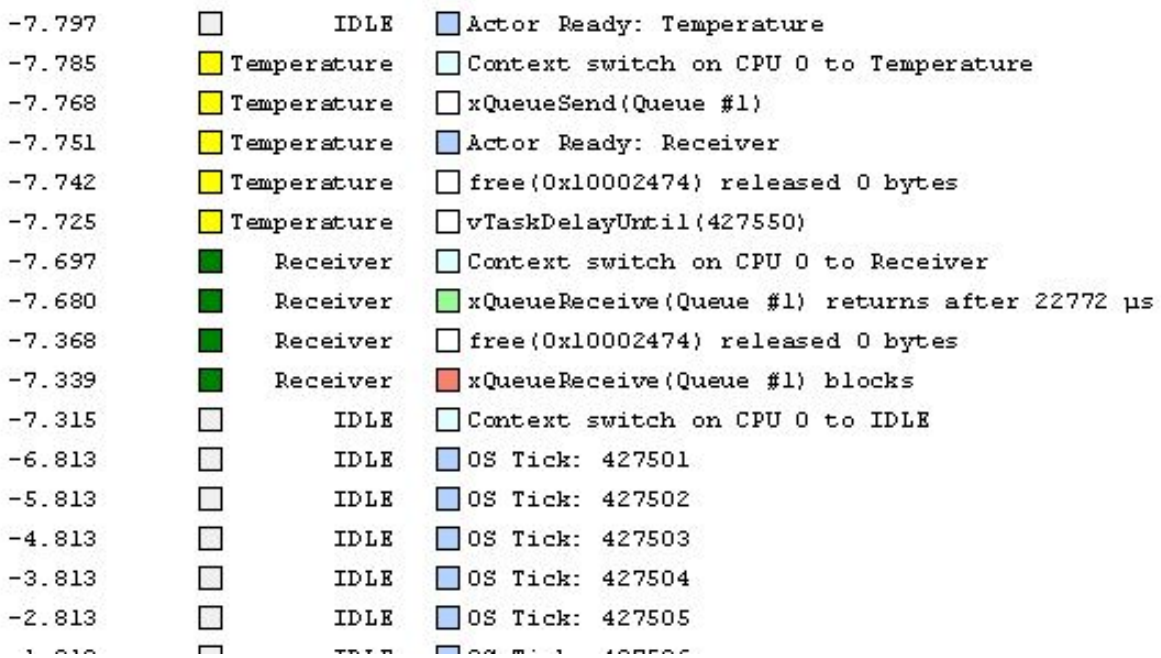

Figura 32: **Event log - secuencia Temperature - Receiver**

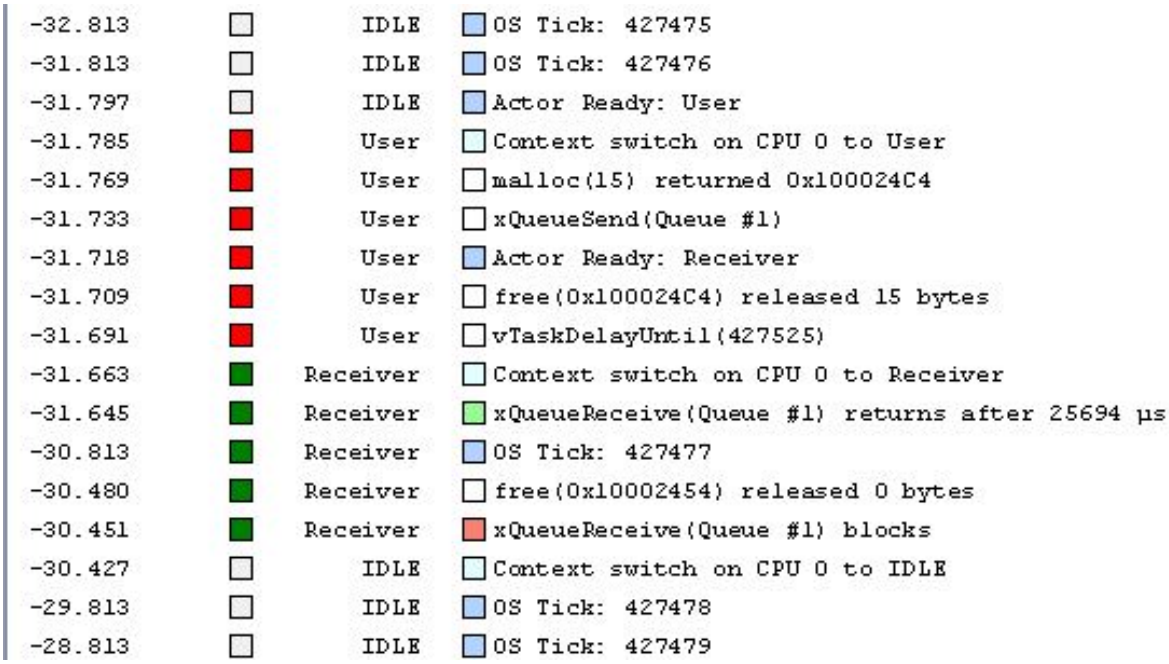

Figura 33: **Event log - secuencia User - Receiver**

Código

```
/*
 * TP4 OSII – FreeRTOS
 *
 * @author Casabella Martin *
 *
 *
 * */

#include <string.h>
#include <math.h>
/* Kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Priorities at which the tasks are created. */
#define RECEIVE_TASK_PRIORITY   ( tskIDLE_PRIORITY + 1 )
#define PERIODIC_TASK_PRIORITY    ( tskIDLE_PRIORITY + 4 )
#define APERIODIC_TASK_PRIORITY   ( tskIDLE_PRIORITY + 4 )
/* The bit of port 0 that the LPCXpresso LPC13xx LED is connected. */
#define mainLED_BIT             ( 22 )
#define MAX_POINTERS    10

/* The rate at which data is sent to the queue, specified in milliseconds. */
#define PERIODIC_SEND_FREQUENCY_MS      ( 50 / portTICK_RATE_MS )

#define MAX_STRING_LEN  51

#define BUF_TEMP 10
#define portNEW_DELAY 200

/*Prototipos de funciones*/
static void vReceiverTask(void *pvParameters);
static void vPeriodicSenderTask(void *pvParameters);
static void vAperiodicSenderTask(void *pvParameters);

static void vToggleLED(void);
char * vRandomString(int len_string);

void UART3_Init(void);
void UART_Send(char* datos, int size);

/* Declare a variable of type QueueHandle_t to hold the handle of the queue being created. */
QueueHandle_t xPointerQueue;

/*Estructura para uso de strings variables*/
struct msg_struct {
  char * msg;
  char num;
  int len_str;
};

int main(void) {
  /* P0_22 for the LED. */
  LPC_PINCON->PINSEL1 &= (~(3 << 12));
  LPC_GPIO0->FIODIR |= (1 << mainLED_BIT);

  /* Enable traceanalycer snapshot */
  vTraceEnable(TRC_START);

  /* Create a queue that can hold a maximum of MAX_POINTERS pointers*/
```

```
65    xPointerQueue = xQueueCreate(MAX_POINTERS, sizeof(struct msg_struct *)); //creo cola de 10
        punteros a struct
66
67    if (xPointerQueue != NULL) {
68
69      /*
70       * The size of the stack used by the idle task is defined by the applicationdefined
        constant configMINIMAL_STACK_SIZE . The value assigned to
71       * this constant in the standard FreeRTOS Cortex-M3 demo applications is the minimum
        recommended for any task. If your task uses a lot of stack
72       * space, then you must assign a larger value.     *
73       *
74       *For example, the Cortex-M3 stack is 32 bits wide so, if
75       *usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 *
        4 bytes).
76       *
77       */
78
79      //taskcode, name, stacksize, parameters, handletask
80      xTaskCreate(vReceiverTask, "Receiver", configMINIMAL_STACK_SIZE,
81      NULL, RECEIVE_TASK_PRIORITY, NULL);
82
83      xTaskCreate(vPeriodicSenderTask, "TemperatureSensor",
84      configMINIMAL_STACK_SIZE, NULL, PERIODIC_TASK_PRIORITY,
85      NULL);
86
87      xTaskCreate(vAperiodicSenderTask, "User", configMINIMAL_STACK_SIZE,
88          NULL, APERIODIC_TASK_PRIORITY , NULL);
89
90      /*Dimos prioridades iguales a ambas tareas productoras, mientras que la lectora tiene
        prioridad mas alta*/
91
92      /**
93       * Passing a uxPriority value above (configMAX_PRIORITIES - 1) will result in the priority
        assigned to the task being capped silently to the
94       * maximum legitimate value.
95       *
96       */
97
98      vTaskStartScheduler();
99
100   }
101
102   /* If all is well we will never reach here as the scheduler will now be
103    running.  If we do reach here then it is likely that there was insufficient
104    heap available for the idle task to be created. */
105   for (;;)
106     ;
107   return 0;
108 }
109
110
111 /**
112  * Tarea de envio periodico representando el seensor de temperatura. Se genera
113  * un valor entre 0 y 255, emulando medicion, y esta tarea se encarga de comunicarla
114  * con el proceso central.
115  *
116  *
117  * @param void*  pvParameters
118  */
119 static void vPeriodicSenderTask(void *pvParameters) {
120
121   /*  This parameter is named on the assumption that vTaskDelayUntil() is
122    being used to implement a task that executes periodically and with a
123    fixed frequency. */
124
125   /* The xLastWakeTime variable needs to be initialized with the current tick
126    count. Note that this is the only time the variable is written to explicitly.
```

```
127     After this xLastWakeTime is updated automatically internally within
128     vTaskDelayUntil(). */
129    portTickType xLastWakeTime;
130    xLastWakeTime = xTaskGetTickCount();
131
132    /* Define a task that performs an action every 50 milliseconds. */
133    const TickType_t xPeriod = pdMS_TO_TICKS(50);
134
135    struct msg_struct *msg = pvPortMalloc(sizeof(struct msg_struct *));
136
137    /* Enter the loop that defines the task behavior. */
138    for (;;) {
139
140      /* This task should execute every 50 milliseconds. Time is measured
141       in ticks. The pdMS_TO_TICKS macro is used to convert milliseconds
142       into ticks. xLastWakeTime is automatically updated within vTaskDelayUntil()
143       so is not explicitly updated by the task. */
144
145      /* Place this task in the blocked state until it is time to run again.
146       The block state is specified in ticks, the constant used converts ticks
147       to ms.  While in the blocked state this task will not consume any CPU
148       time.
149
150       The parameters to vTaskDelayUntil() specify, instead, the exact tick count value at which
        the
151       calling task should be moved from the Blocked state into the Ready state. vTaskDelayUntil
        ()
152       is the API function that should be used when a fixed execution period is required (where
        you
153       want your task to execute periodically with a fixed frequency), as the time at which the
        calling
154       task is unblocked is absolute, rather than relative to when the function was called (as is
        the
155       case with vTaskDelay()).
156
157       */
158      vTaskDelayUntil(&xLastWakeTime, xPeriod);
159
160      msg->msg = NULL;
161      msg->num = (char) (rand() % 255); //genero valor aleatorio de 0 a 255 y lo casteo a char
162
163      /* Send to the queue - causing the queue receive task to flash its LED.
164       *
165       *  0 is used as the block time so the sending operation will not block -
166       *  it shouldn't need to block as the queue should always be empty at this
167       *  point in the code.
168       *
169       *  */
170
171      /*(pdMS_TO_TICKS(5): The maximum amount of time the task should remain in the
172       *Blocked state to wait for space to become available on the queue,
173       *should the queue already be full.*/
174      xQueueSend(xPointerQueue, &msg, pdMS_TO_TICKS(5));
175      //espera 5 [ms] si la cola esta llena
176
177
178      /*Free a estructura*/
179      vPortFree(msg);
180
181    }
182  }
183
184  /**
185   * Funcion Aperiodica representa el ingreso de caracteres de un usuario. Se genera una cadena
       de longitud
186   * variable, de manera aleatoria, y se envia aperiodicamente (i,e cada cierto tiempo que
       tambien es variable)
187   *
```

```
188  * @param: void*  pvParameters
189  *
190  */
191 static void vAperiodicSenderTask(void *pvParameters) {
192
193   portTickType xLastWakeTime;
194   /*Aloco estructura donde colocara cada string generado para colocar en la cola*/
195   struct msg_struct *msg = pvPortMalloc(sizeof(struct msg_struct *));
196   msg->msg = pvPortMalloc(sizeof(char *));
197   msg->num = 0; //le asigno 0 para que no tenga mugre el campo
198
199   int random_len = 0;
200
201   xLastWakeTime = xTaskGetTickCount();
202
203   /* Define a task that performs an action every random milisecons starting (at least 20)*/
204   const TickType_t xPeriod = pdMS_TO_TICKS((rand() %(150 - 20) + 20));
205   /*Esta tarea dura entre 20 ms y 150     rand()%(nMax-nMin) + nMin; */
206
207   for (;;) {
208     /* Place this task in the blocked state until it is time to run again.
209      The block state is specified in ticks, the constant used converts ticks
210      to ms.  While in the blocked state this task will not consume any CPU
211      time. */
212     vTaskDelayUntil(&xLastWakeTime, xPeriod);
213
214     random_len = (rand() + xTaskGetTickCount()) %(MAX_STRING_LEN - 3);
215
216     /* Send to the queue - causing the queue receive task to flash its LED.*/
217     msg->msg = vRandomString(random_len);
218     msg->len_str = random_len + 2; //+2 porque no incluye el \0
219
220     /*Envio string, esperando hasta 5[ms]  si la cola esta llena
221      *
222      * xQueueSend(), xQueueSendToFront() and xQueueSendToBack() will
223      * return immediately if xTicksToWait is zero and the queue is already full.
224      *
225      * */
226     xQueueSend(xPointerQueue, &msg, pdMS_TO_TICKS(5));
227
228     /*Free a estructura*/
229     vPortFree(msg->msg);
230   }
231
232 }
233
234 /**
235  * Tarea de recepcion. Se fija en la cola si hay algo y lo recibe. Compara campos de la
        estructura, y lo
236  * reenvia por puerto serie.
237  *
238  *
239  * @param void*  pvParameters
240  */
241
242 static void vReceiverTask(void *pvParameters) {
243
244   struct msg_struct * pcReceivedValue; //creo puntero a estructura
245
246   //char cReceivedValue;
247   char tmp_buf[BUF_TEMP];
248
249   /*Modulo UART*/
250   UART3_Init();
251
252   for (;;) {
253     /* Wait until something arrives in the queue - this task will block
254      indefinitely provided INCLUDE_vTaskSuspend is set to 1 in
```

```
255       FreeRTOSConfig.h. */
256
257     /* Receive the address of a buffer. */
258     if (xQueueReceive(xPointerQueue, /* The handle of the queue. */
259     &pcReceivedValue, /* Store the 'buffers address in pcReceivedString. */
260     portNEW_DELAY) == pdTRUE) {
261
262       if (pcReceivedValue->msg == NULL) {
263         /*Vino valor de temperatura*/
264
265         /*Convierto valor recibido, a base 10, y lo pongo en tmp buf*/
266         itoa(pcReceivedValue->num, tmp_buf, 10);
267
268         /*Concateno \r\n*/
269         strcat(tmp_buf, "\r\n");
270
271         /*Mando la temperatura recibida a traves del puerto UART*/
272         UART_Send(tmp_buf, strlen(tmp_buf));
273
274       } else {
275
276         /*Vino string de longitud variable (msg tiene algo)*/
277
278         /*Lo mando directamente (ya es char*, no necesito casteo)*/
279         UART_Send(pcReceivedValue->msg, pcReceivedValue->len_str);
280
281       }
282
283       /* toggle LED 22 */
284       vToggleLED();
285       /*Free a estructura*/
286       vPortFree(pcReceivedValue);
287
288     }
289
290 //    } else {
291 //      /* Returned if data cannot be read from the queue because the queue is
292 //       * already empty.
293 //       *
294 //       * If a block time was specified (xTicksToWait was not zero) then the
295 //       * calling task will have been placed into the Blocked state to wait for
296 //       * another task or interrupt to send data to the queue, but the block time
297 //       * expired before this happened.
298 //       *
299 //       */
300 //    }
301
302   }
303
304 }
305
306 /*
307  * Funcion para generar string de longitud variable. No solo varia la longitud de
308  * la cadena, sino que la posicion dentro del arreglo fijo, i.e, si genero una cadena
309  * de largo 5, no implica que sean los primeros 5 caracteres siempre.
310  *
311  * @param: len_str longitud de la cadena devuelta
312  * @return: char* cadena generada
313  *
314  * */
315
316 char * vRandomString(int len_string) {
317   static char* leters = "a0b1c2d3e4f5g6h7i8j9kol1m2n3o4p5q6r7s8t9u0v1w2x3y4z5";
318
319   /* +3 para el agregado de \r\n */
320   char* var_string = pvPortMalloc(len_string += 3);
321
322   for (int i = 0; i < len_string - 3; i++) {
```

```
323     var_string[i] = leters[rand() % MAX_STRING_LEN]; //elige elemento con indice de 0 a 50
324   }
325   var_string[len_string - 3] = '\r';
326   var_string[len_string - 2] = '\n';
327   var_string[len_string - 1] = '\0';
328 
329   return var_string;
330 }
331 
332 /**
333  * toggle LED 22
334  */
335 static void vToggleLED(void) {
336   unsigned long ulLEDState;
337 
338   /* Obtain the current P0 state. */
339   ulLEDState = LPC_GPIO0->FIOPIN;
340 
341   /* Turn the LED off if it was on, and on if it was off. */
342   LPC_GPIO0->FIOCLR = ulLEDState & (1 << mainLED_BIT);
343   LPC_GPIO0->FIOSET = ((~ulLEDState) & (1 << mainLED_BIT));
344 }
345 
346 /**
347  * configuracion UART
348  */
349 void UART3_Init(void) {
350 
351   /*UART3, chau UART1*/
352   LPC_SC->PCONP |= (1 << 25);
353   /*Deshabilito tambien UART0 y UART1*/
354   LPC_SC->PCONP &= ~(3 << 3);
355   /*PCLKSEL1*/
356   LPC_SC->PCLKSEL1 |= (1 << 18);
357 
358   /*Palabra de 8 bits*/
359   LPC_UART3->LCR = 0x03;
360   /*Bit de stop*/
361   LPC_UART3->LCR |= (1 << 2);
362   /*Habilito configuarcion*/
363   LPC_UART3->LCR |= 0b10000000;
364   LPC_UART3->DLL = 54; //*U3LCR 0b10100001 ; // 115200
365   LPC_UART3->DLM = 0; //*U3LCR
366   /*Deshabilito cfg baudrate*/
367   LPC_UART3->LCR &= ~(1 << 7);
368 
369   //pin 0 TXD0 pin 1 RXD0 puerto 0
370   LPC_PINCON->PINSEL0 = 0b1010; // *PINSEL0 configurar los pines port 0
371   LPC_PINCON->PINMODE0 = 0; // *PINMODE0 pin a pull up
372 
373   /*
374    * LPC_UART3->IER = 1; // *U3IER habilito la interrupcion por Recive Data Aviable
375    * *ISER0 |= 1<<8; //activate interrup uart3
376    */
377 }
378 
379 void UART_Send(char* data, int size) {
380   for (int i = 0; i < size; i++) {
381     while ((LPC_UART3->LSR & (1 << 5)) == 0) {
382     } //*U3LSR // Wait for Previous transmission
383     LPC_UART3->THR = data[i]; //*U3THR
384   }
385 }
386 
387 void vConfigureTimerForRunTimeStats(void) {
388   const unsigned long TCR_COUNT_RESET = 2, CTCR_CTM_TIMER = 0x00,
389       TCR_COUNT_ENABLE = 0x01;
390 
```

```
391   /* This function configures a timer that is used as the time base when
392    collecting run time statistical information - basically the percentage
393    of CPU time that each task is utilising.  It is called automatically when
394    the scheduler is started (assuming configGENERATE_RUN_TIME_STATS is set
395    to 1). */
396
397   /* Power up and feed the timer. */
398   LPC_SC->PCONP |= 0x02UL;
399   LPC_SC->PCLKSEL0 = (LPC_SC->PCLKSEL0 & (~(0x3 << 2))) | (0x01 << 2);
400
401   /* Reset Timer 0 */
402   LPC_TIM0->TCR = TCR_COUNT_RESET;
403
404   /* Just count up. */
405   LPC_TIM0->CTCR = CTCR_CTM_TIMER;
406
407   /* Prescale to a frequency that is good enough to get a decent resolution,
408    but not too fast so as to overflow all the time. */
409   LPC_TIM0->PR = ( configCPU_CLOCK_HZ / 10000UL) - 1UL;
410
411   /* Start the counter. */
412   LPC_TIM0->TCR = TCR_COUNT_ENABLE;
413 }
414
415 /*
416  * Necessary functions for FreeRTOS
417  */
418 void vApplicationStackOverflowHook(TaskHandle_t pxTask, char *pcTaskName) {
419   /* This function will get called if a task overflows its stack. */
420   (void) pxTask;
421   (void) pcTaskName;
422   for (;;)
423     ;
424 }
```

Script Python para comunicación serial

```
import serial  # PySerial


#Configuracion de puerto
ser=serial.Serial(
    port = 'COM5',
    baudrate = 115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS
 )

print ('Puerto abierto: ', ser.name)
print(ser)
print('')
#Flush entrada y salida
ser.flushInput()
ser.flushOutput()

#Variable para ir handleando lecturas
out_lpc = ''

#Loop de lectura
while(1):
```

```
    #Leo byte, lo parseo a string
    read_byte = str(ser.read())

    #Si llego \n (salto de linea) imprimo recepcionn completa
    if (read_byte=='\n'):
        print(out_lpc)
    else:
        #No llego \n, entonces sigo recibiendo bytes
        out_lpc+=read_byte
```

Referencias

[1] Desarrollo de aplicaciones en LPCXpresso basadas en RTOS. http://www.sase.com.ar/2011/files/2010/11/SASE2011-Desarrollo-Apliaciones-RTOS-LPCXpresso.pdf.

[2] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd, 2010.

[3] freeRTOS. https://www.freertos.org.