Cátedra de Sistemas Operativos II

FACULTAD DE CIENCIAS EXACTAS,
FÍSICAS Y NATURALES

Trabajo Práctico N°4

Alumno: Oliva Arias, Carlos Agustín
Fecha: 21 de Junio de 2019

Índice

1. Introducción

Toda aplicación de ingeniería que posea requerimientos rigurosos de tiempo, y que esté controlado por un sistema de computación, utiliza un Sistema Operativo de Tiempo Real (RTOS, por sus siglas en inglés). Una de las características principales de este tipo de SO, es su capacidad de poseer un kernel preemptive y un scheduler altamente configurable. Numerosas aplicaciones utilizan este tipo de sistemas tales como aviónica, radares, satélites, etc. lo que genera un gran interés del mercado por ingenieros especializados en esta área.

1.1. Propósito

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

1.2. Ámbito del sistema

Para el siguiente trabajo se utilizaron los siguientes componentes de hardware y software:

- LPC1769 Placa de desarrollo con procesador CORTEX-M3, donde corre FreeRTOS.
- CMSIS Librería de programación de la placa, con funciones que permiten tanto su configuración como uso.
- UART-USB Pequeño hardware de interfaz para leer Uart en un puerto usb.
- FreeRTOS Sistema Operativo de tiempo real para microcontroladores.
- Tracealyzer Software que permite recopilar y analizar un trazo de ejecución en FreeRTOS.

1.3. Definiciones, Acrónimos y Abreviaturas

RTOS: Real Time Operating System (Sistena Operativo de tiempo real)
CMSIS: Cortex Microcontroller Software Interface Standard (Interfaz de Software estándar para Microcontroladores Cortex)
UART: Universal Asynchronous Receiver-Transmitter (Transmisor-Receptor

Asíncrono Universal)
EDIII: Electrónica Digital 3
IDE: Integrated Development Environment (entorno de desarrollo integrado)

1.4. Referencias

Referencias

[1] https://www.freertos.org/

[2] https://percepio.com/tracealyzer/

[3] https://www.freertos.org/wp-content/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

[4] https://www.freertos.org/a00111.html

[5] https://percepio.com/gettingstarted-freertos/

[6] https://www.freertos.org/FAQHelp.html

[7] https://www.nxp.com/docs/en/user-guide/UM10360.pdf

[8] https://percepio.com/docs/FreeRTOS/manual/index.html

1.5. Descripción general del documento

Las siguientes secciones describen el proceso de realización del trabajo, las herramientas y archivos auxiliares utilizados.

2. Descripción general

Se pide que, sobre un sistema embebido de arquitectura compatible con FreeRTOS (ej: ARM Cortex M4): 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. Por otro lado, el consumidor, es una tarea que envía el valor recibido a la terminal de una computadora por puerto serie (UART). Y nuevamente, realizar un análisis del sistema con Tracealyzer.

3. Diseño de solución

Se configuraron un proyecto por aplicación requerida, clonando el proyecto inicial, cuya confección se detalla en la guía del apéndice. Ambos tienen múltiples archivos fuente, de los cuales solo se detallaran aquí los principales. Para el primero que consta de dos tareas simples y una cola, las tareas de productor y consumidor simplemente envía y recibe (respectivamente) un número aleatorio, e imprimen un mensaje en el trazo a analizar. Se adjunta el código utilizado:

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

#include <stdlib.h>

static void vProductorTask( void *pvParameters );
static void vConsumidorTask( void *pvParameters );

traceString logger;
xQueueHandle xQueue;


int main( void )
{
 vTraceEnable(TRC_START);
 logger = xTraceRegisterString("Log");

    xQueue = xQueueCreate( 2, sizeof( long ) );
 if( xQueue != NULL )
 {
  xTaskCreate( vProductorTask, "Productor", 240, NULL, 1,
      NULL );
  xTaskCreate( vConsumidorTask, "Consumidor", 240, NULL, 2,
      NULL );

  vTaskStartScheduler();
 }
 else
 {
```

```
29    vTracePrint(logger,"La cola no pudo ser creada.");
30   }
31   for( ;; );
32   return 0;
33  }
34
35  static void vProductorTask( void *pvParameters )
36  {
37   srand(xTaskGetTickCount());
38   long lValueToSend;
39   portBASE_TYPE xStatus;
40   portTickType xLastWakeTime;
41   for( ;; )
42   {
43    xLastWakeTime = xTaskGetTickCount();
44    lValueToSend = rand() % 100;
45    xStatus = xQueueSend( xQueue, &lValueToSend, 0 );
46    if( xStatus != pdPASS )
47    {
48     vTracePrint(logger, "No se pudo enviar a la cola." );
49    }
50    vTracePrintF(logger,"envie %d",lValueToSend);
51    vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS )
          );
52   }
53  }
54
55  static void vConsumidorTask( void *pvParameters )
56  {
57  long lReceivedValue;
58  portBASE_TYPE xStatus;
59  const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
60
61   for( ;; )
62   {
63    xStatus = xQueueReceive( xQueue, &lReceivedValue,
         xTicksToWait );
64    if( xStatus == pdPASS )
65    {
66     vTracePrintF(logger,"recibi  %d", lReceivedValue );
67    }
68    else
69    {
70     vTracePrint(logger, "No se pudo recibir de la cola");
71    }
72    vTaskDelay( 250 / portTICK_RATE_MS );
73   }
74  }
75
```

```
76 void vApplicationMallocFailedHook( void )
77 {
78  for( ;; );
79 }
80
81 void vApplicationStackOverflowHook( xTaskHandle *pxTask,
       signed char *pcTaskName )
82 {
83  for( ;; );
84 }
85
86 void vApplicationIdleHook( void )
87 {
88
89 }
90
91 void vApplicationTickHook( void )
92 {
93
94 }
95
96 void vConfigureTimerForRunTimeStats( void )
97 {
98 const unsigned long TCR_COUNT_RESET = 2, CTCR_CTM_TIMER = 0
       x00, TCR_COUNT_ENABLE = 0x01;
99
100  /* Power up and feed the timer. */
101  LPC_SC->PCONP |= 0x02UL;
102  LPC_SC->PCLKSEL0 = (LPC_SC->PCLKSEL0 & (~(0x3<<2))) | (0x01
       << 2);
103
104  /* Reset Timer 0 */
105  LPC_TIM0->TCR = TCR_COUNT_RESET;
106
107  /* Just count up. */
108  LPC_TIM0->CTCR = CTCR_CTM_TIMER;
109
110  /* Prescale to a frequency that is good enough to get a
       decent resolution,
111  but not too fast so as to overflow all the time. */
112  LPC_TIM0->PR =  ( configCPU_CLOCK_HZ / 10000UL ) - 1UL;
113
114  /* Start the counter. */
115  LPC_TIM0->TCR = TCR_COUNT_ENABLE;
116 }
```

programa1.c

El segundo proyecto realiza el segundo programa solicitado. Se crean dos tareas como productores, uno de datos periódicos y tamaño fijo de 1 byte (temperatura), y un segundo productor aperiódico con datos de tamaño variable simulando el ingreso de un comando por teclado(user). Además se crea una tarea (Transmisor) como consumidor de los datos para enviar por UART.

Para la utilización de una cola que recibe dos tipos de datos diferentes se implemento una estructura que contiene ambos tipos de datos, llenándose solamente la parte necesaria en cada productor.

Ambas tareas productoras simulan sus actividades, mediante la generación aleatoria de datos. User confecciona una cadena de texto entre 1-29 caracteres en el rango [a-z]. Sensor genera un dato del tipo *uint8_t* cuyo rango varia entre [0-255].

La tarea Tx recoge los datos desde la cola y envía utilizando UART0 a 9600 baudios. La tarea Tx parsea esos datos para enviarlos, así pueden visualizarse por la terminal de la PC conectada.

A continuación el código utilizado

```
/*CMSIS includes*/
#include "LPC17xx.h"
#include "lpc17xx_uart.h"
#include "lpc17xx_pinsel.h"
#include "lpc17xx_gpio.h"

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


/* Parametros */
#define MAX_INPUT_SIZE 30
#define USR_PRIO 4
#define TEMP_PRIO 2
#define TX_PRIO 3
#define MAX_QUEUE_SIZE 5
#define TX_BAUD_RATE 9600



void vUart_config();
void vPinsel_config();
```

```
29 /* Tasks */
30 static void vTaskSensor(void * pvParameters);
31 static void vTaskUsuario(void * pvParameters);
32 static void vTaskTx(void * pvParameters);
33
34 typedef struct{
35  char * msg;
36  uint8_t temp;
37  int msg_len;
38 }xInput;
39
40 xQueueHandle xQueue;
41 traceString logger;
42
43 int main( void )
44 {
45   /* Initialize hardware */
46  vUart_config();
47  vPinsel_config();
48
49  /* Initialize tracealyzer*/
50  vTraceEnable(TRC_START);
51
52  logger = xTraceRegisterString("Log");
53
54  xQueue = xQueueCreate( MAX_QUEUE_SIZE, sizeof( xInput *) );
55
56  if( xQueue != NULL )
57  {
58   /*Crear las 3 tareas*/
59   xTaskCreate( vTaskTx, ( char * ) "Tx", 240, NULL, TX_PRIO,
        NULL );
60   xTaskCreate( vTaskSensor, ( char * ) "Sensor", 240, NULL,
        TEMP_PRIO, NULL );
61   xTaskCreate( vTaskUsuario, ( char * ) "User", 240, NULL,
        USR_PRIO, NULL );
62  }
63
64  vTaskStartScheduler();
65  for( ;; );
66  return 0;
67 }
68
69 static void vTaskSensor(void * pvParameters)
70 {
71  portTickType xLastWakeTime = xTaskGetTickCount();
72
73  const TickType_t xPeriod = pdMS_TO_TICKS(1000);
74
```

```
75  portBASE_TYPE xStatus;
76  for (;;)
77  {
78   xInput *in = pvPortMalloc(sizeof(xInput *));
79
80   in->temp = rand() % 256;
81   in->msg = NULL;
82
83   xStatus = xQueueSendToBack( xQueue, &in, pdMS_TO_TICKS(1));
         //100
84   if( xStatus != pdPASS )
85   {
86    vTracePrint(logger,"No se pudo enviar el dato a la cola."
         );
87   }
88   vPortFree(in);
89   vTracePrint(logger, "temperatura");
90
91   vTaskDelayUntil( &xLastWakeTime, xPeriod ); //Periodico
       cada 1000ms
92  }
93 }
94
95 static void vTaskUsuario(void * pvParameters)
96 {
97  srand(xTaskGetTickCount());
98  xInput * in = pvPortMalloc(sizeof(xInput *));
99
100 for (;;)
101 {
102  int i;
103  int cantidad = (rand() % MAX_INPUT_SIZE) + 1;
104  in -> msg = pvPortMalloc(sizeof(char *) * cantidad);
105  for(i=0; i < cantidad; i++)
106  {
107   char  aux = (rand() % 25) + 97;
108   in -> msg[i] = aux;
109  }
110  in -> msg_len = cantidad;
111  vTracePrint(logger,"usuario");
112  xQueueSend(xQueue, &in, 100);
113  vPortFree(in->msg);
114  vTaskDelay(( (rand() % 2000) + 2000) / portTICK_RATE_MS );
115 }
116 }
117
118
119 static void vTaskTx(void * pvParameters)
120 {
```

```
121  portBASE_TYPE xStatus;
122  xInput *xIn = pvPortMalloc(sizeof(xInput *));
123  char xStr[4];
124  for (;;)
125  {
126   xStatus = xQueueReceive( xQueue, &xIn, portMAX_DELAY );
127   if( xStatus == pdPASS )
128   {
129    if(xIn->msg == NULL)
130    {
131     sprintf(xStr,"%3d\n",(uint8_t)xIn->temp);
132     UART_Send(LPC_UART0,(unsigned char *)xStr,sizeof(xStr),
          NONE_BLOCKING);
133     vTracePrintF(logger,"Uart: %d",(int) xIn->temp);
134    }
135    else
136    {
137     char cMsg [(xIn->msg_len)+1];
138     sprintf(cMsg,"%s\n",xIn->msg);
139     UART_Send(LPC_UART0,(unsigned char *) cMsg,xIn->msg_len
          +1, NONE_BLOCKING);
140     vTracePrintF(logger,"Uart: %s",xIn->msg);
141    }
142   }else
143   {
144    vTracePrint(logger,"No se pudo leer de la cola");
145   }
146  }
147 }
148
149 void vApplicationMallocFailedHook( void )
150 {
151  vTracePrint(logger,"ApplicationMallocFailed");
152 }
153
154 void vApplicationStackOverflowHook( xTaskHandle *pxTask,
     signed char *pcTaskName )
155 {
156  vTracePrint(logger,"ApplicationStackOverflow");
157 }
158
159 void vApplicationIdleHook( void )
160 {
161  /* This example does not use the idle hook to perform any
     processing. */
162 }
163
164 void vApplicationTickHook( void )
165 {
```

```
166  /* This example does not use the tick hook to perform any
         processing. */
167 }
168
169 void vConfigureTimerForRunTimeStats( void )
170 {
171 const unsigned long TCR_COUNT_RESET = 2, CTCR_CTM_TIMER = 0
        x00, TCR_COUNT_ENABLE = 0x01;
172
173  /* Power up and feed the timer. */
174  LPC_SC->PCONP |= 0x02UL;
175  LPC_SC->PCLKSEL0 = (LPC_SC->PCLKSEL0 & (~(0x3<<2))) | (0x01
         << 2);
176
177  /* Reset Timer 0 */
178  LPC_TIM0->TCR = TCR_COUNT_RESET;
179
180  /* Just count up. */
181  LPC_TIM0->CTCR = CTCR_CTM_TIMER;
182
183  /* Prescale to a frequency that is good enough to get a
         decent resolution,
184  but not too fast so as to overflow all the time. */
185  LPC_TIM0->PR =  ( configCPU_CLOCK_HZ / 10000UL ) - 1UL;
186
187  /* Start the counter. */
188  LPC_TIM0->TCR = TCR_COUNT_ENABLE;
189 }
190
191 void vPinsel_config() {
192  PINSEL_CFG_Type pin_config;
193  // pin 0.11 RXD2
194  pin_config.Portnum = PINSEL_PORT_0;
195  pin_config.Pinnum = PINSEL_PIN_3;
196  pin_config.Pinmode = PINSEL_PINMODE_PULLUP;
197  pin_config.Funcnum = PINSEL_FUNC_1;
198  pin_config.OpenDrain = PINSEL_PINMODE_TRISTATE;
199  PINSEL_ConfigPin(&pin_config);
200  GPIO_SetDir(0, (1<<11), 0);
201  // pin 0.10 TXD2
202  pin_config.Portnum = PINSEL_PORT_0;
203  pin_config.Pinnum = PINSEL_PIN_2;
204  pin_config.Pinmode = PINSEL_PINMODE_PULLUP;
205  pin_config.Funcnum = PINSEL_FUNC_1;
206  pin_config.OpenDrain = PINSEL_PINMODE_TRISTATE;
207  PINSEL_ConfigPin(&pin_config);
208  GPIO_SetDir(0, (1<<10), 1);
209 }
210
```

```
211 void vUart_config() {
212  UART_CFG_Type struct_uart;
213  UART_FIFO_CFG_Type struct_fifo;
214  UART_ConfigStructInit(&struct_uart);
215  struct_uart.Baud_rate = TX_BAUD_RATE;
216  struct_uart.Databits = UART_DATABIT_8;
217  UART_Init(LPC_UART0, &struct_uart);
218  UART_FIFOConfigStructInit(&struct_fifo);
219  UART_FIFOConfig(LPC_UART0, &struct_fifo);
220  UART_IntConfig(LPC_UART0,UART_INTCFG_RBR,ENABLE);
221  UART_TxCmd(LPC_UART0, ENABLE);
222 }
```

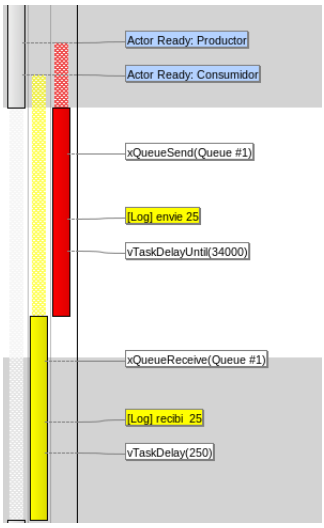
programa2.c

4. Implementación y resultados

4.1. Consumidor y productor

Para el primer programa se analizaron dos variaciones, la primera da igual prioridad a las tareas, y la segunda da mayor prioridad al consumidor.

Un trazo de la ejecución en igual prioridad puede verse en la siguiente figura:



Igual Prioridad

Puede observarse que ambas tareas se encuentran listas y el planificador entrega la ejecución al productor (por estar listo anteriormente). El productor genera el dato, y lo envía a la cola, para luego imprimir un mensaje y bloquearse mediante vTaskDelayUntil() el cual es apropiado para tareas periódicas. Logrado esto, se entrega el procesador al consumidor que retira el dato e imprime también su mensaje, para bloquearse mediante vTaskDelay(). La elección diferenciada se explicara con detalle más adelante.

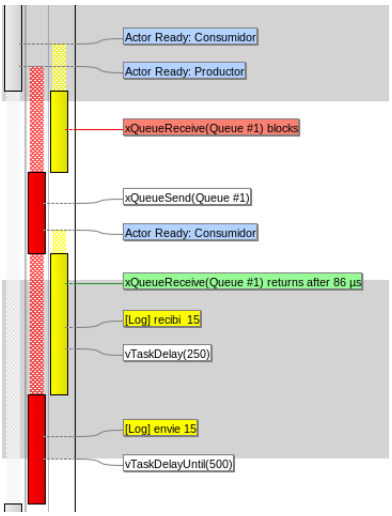
Puede verse en el siguiente fragmento del log la precisión que se logra en el cumplimiento de las tareas.

```
[ 33.750.445]  Productor [Log] envie 25
[ 33.750.525] Consumidor [Log] recibi  25
[ 34.000.445]  Productor [Log] envie 52
[ 34.000.526] Consumidor [Log] recibi  52
[ 34.250.445]  Productor [Log] envie 81
[ 34.250.525] Consumidor [Log] recibi  81
[ 34.500.444]  Productor [Log] envie 2
```

```
[ 34.500.525] Consumidor [Log] recibi  2
[ 34.750.445]  Productor [Log] envie 91
[ 34.750.525] Consumidor [Log] recibi  91
[ 35.000.445]  Productor [Log] envie 80
[ 35.000.526] Consumidor [Log] recibi  80
[ 35.250.445]  Productor [Log] envie 15
[ 35.250.525] Consumidor [Log] recibi  15
```

Al ser este ejemplo demasiado sencillo, es trivial el análisis del heap, puesto que los requerimientos de memoria son estáticos.

Veamos ahora que pasa al cambiar la prioridad del consumidor.



Mayor prioridad a consumidor

A simple vista es notable que ha cambiado significativamente. Entonces vemos que la tarea del consumidor se ejecuta primero, intentando leer un dato aunque no haya ninguno en la cola, a lo cual, según lo codificado, se bloquea esperando hasta 100 ms. Ante este bloqueo se permite la ejecución del productor, de prioridad menor, que escribe el dato en la cola , poblándola y haciendo que el consumidor pase al estado de listo (en 15 us según el log que se adjunta debajo). Esto hace que el productor sea desalojado (incluso antes de escribir en el log del trace) para que pueda leerse el dato escrito y escribirse el log. Una vez bloqueada la tarea del consumidor, vuelve a ejecución el productor que puede finalmente escribir el log, y bloquearse nuevamente.

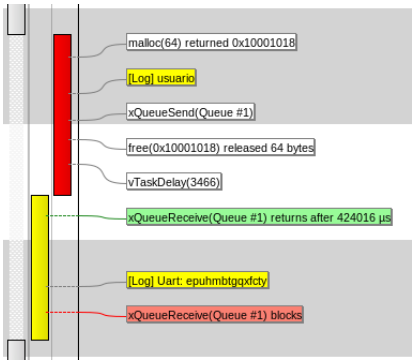
```
1 [    250.368]        IDLE Actor Ready: Consumidor
2 [    250.380]        IDLE Actor Ready: Productor
3 [    250.394] Consumidor Context switch on CPU 0 to
    Consumidor
```

```
[    250.415] Consumidor xQueueReceive(Queue #1) blocks
[    250.439]  Productor Context switch on CPU 0 to Productor
[    250.457]  Productor xQueueSend(Queue #1)
[    250.472]  Productor Actor Ready: Consumidor
[    250.485] Consumidor Context switch on CPU 0 to
   Consumidor
[    250.502] Consumidor xQueueReceive(Queue #1) returns
   after 86 us
[    250.526] Consumidor [Log] recibi  15
[    250.538] Consumidor vTaskDelay(250)
[    250.564]  Productor Context switch on CPU 0 to Productor
[    250.587]  Productor [Log] envie 15
[    250.600]  Productor vTaskDelayUntil(500)
[    250.625]       IDLE Context switch on CPU 0 to IDLE
```

Log distinta prioridad

4.2. Dos productores y un consumidor

Para este programa se configuraron la mayor prioridad a la tarea User, dado que se considero este input de mayor importancia, seguido de Tx y finalmente Sensor. Se analiza una traza de ejecución con esta configuración, donde se encuentra una secuencia de interacción:



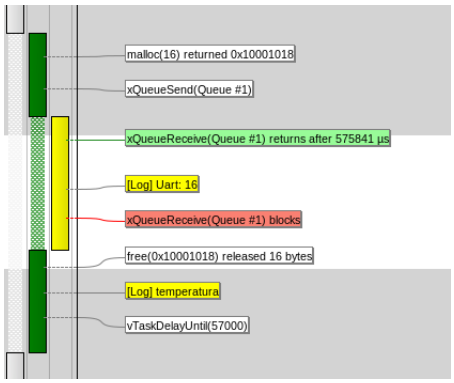
Interacción User-Tx

La tarea User reserva memoria dinámicamente para la escritura del mensaje, escribe en la cola y luego libera la memoria (esto puede hacerse sin perjuicio de perder la información dado que la cola copia el mensaje). Finalmente User se bloquea un tiempo aleatorio.

La tarea Tx que la mayor parte del tiempo se encuentra esperando (su lectura de la cola tiene como argumento el valor máximo de espera) pasa al estado listo en cuanto la cola tiene un dato para procesarlo, enviarlo por

Uart y continuar con su ciclo infinito, que la lleva a esperar otro dato en la cola nuevamente.

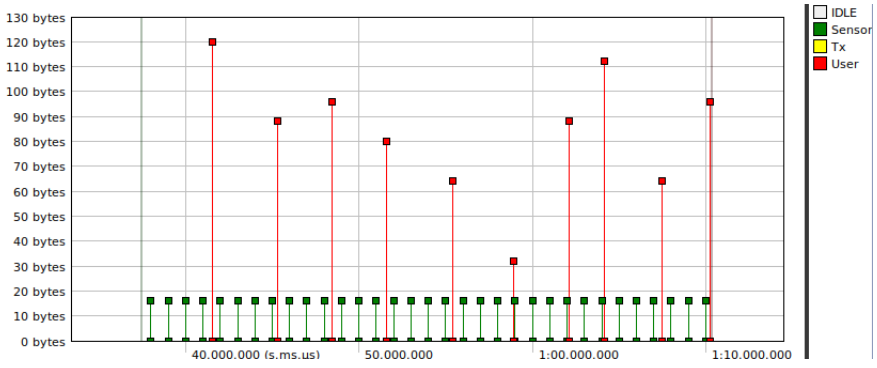
En la siguiente figura vemos otra secuencia:



Interacción Sensor-Tx

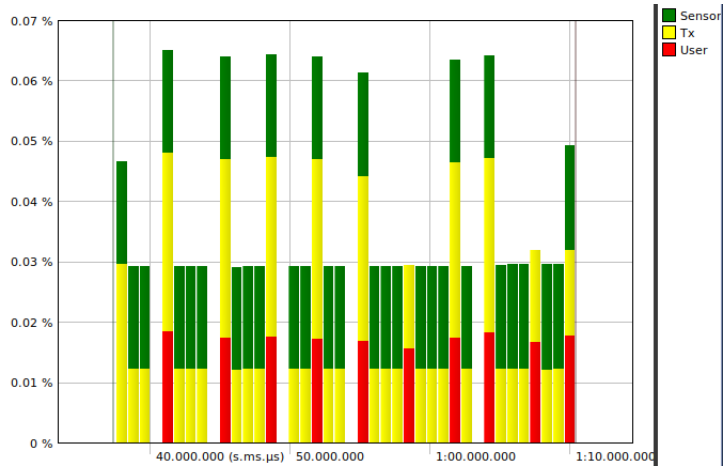
Aquí el sensor reserva memoria, produce el dato y lo encola, haciendo que Tx pase al estado de listo, y logrando ser desalojado por el kernel para dar lugar a Tx que tiene mayor prioridad. La tarea Tx se ejecuta similarmente a lo ya analizado y vuelve a estar bloqueada al no encontrar datos en su cola. Retorna así el procesador a la tarea Sensor que libera la memoria utilizada y se bloquea hasta que reinicie su período.

Aunque el hardware utilizado no tiene problemas de memoria para actividades de esta magnitud, se presentan los gráficos de utilización de procesador y memoria a modo ilustrativo.



Heap

Las reservas dinámicas de memoria para el mensaje del usuario se liberan casi inmediatamente. El trabajo utiliza heap_4.c aunque no es determinista, para evitar la fragmentación que podría introducir esta tarea.



Uso CPU

El procesador nunca ocupa demasiado tiempo para atender a estas tres actividades propuestas.

5. Conclusiones

En el presente trabajo práctico se implemento y analizo un sistema sencillo, con requerimientos en tiempo real, se pudo observar la precisión en los tiempos de ejecución y la rapidez para el cambio de tarea que presenta el kernel. En la realización se revisaron conceptos de sistemas operativos como planificación, reserva de memoria, fragmentación, etc. Se pudo dar cuenta de las estructuras de control más básicas tales como tareas y colas que hacen a la implementación de sistemas embebidos con FreeRTOS.

6. Apéndice

6.1. Guía paso a paso

- A la hora de iniciar el trabajo primero se investigo sobre FreeRTOS [1] así como también se aprendió sobre Tracealyzer [2], porque era la primera vez que se trabajaba con los mismos.
- Posteriormente se instalo MCUXpresso como IDE puesto que ya se había trabajado en EDIII con la misma.
- El primer objetivo propuesto fue hacer correr el primer ejemplo del libro [3] (cuyos códigos de LPC176x se pueden bajar en la red), el mismo es

"semi-hosted.eimprime en la consola de la IDE cadenas de texto. Una vez logrado esto, con cierta familiaridad sobre las nociones mínimas se inicio el trabajo práctico.

- Se instalo además el plugin de tracealyzer para realizar instantáneas (snapshots).
- Como primera medida se configuro un proyecto nuevo, enlazándolo a la librería CMSISv2. Luego se integro todos los archivos fuente de FreeRTOS (task.c, queue.c, etc) y los headers correspondientes, el código para GCC CORTEXM3 (port.c y portmacro.h), MemMang (heap_4.c [4]) y por último de la demo para LPC1768 el header de configuración (FreeRTOSConfig.h).
- Se corrigieron varios errores sencillos (tipográficos, macros mal definidas, utilización de nombres viejos, etc) para lograr la compilación.
- Ambas aplicaciones requeridas son relativamente sencillas, y tienen la forma del ejemplo 10 del libro anteriormente citado. Por lo que se las utilizo a partir de este punto para probar la integración de las herramientas a utilizar.
- En este punto solo faltaba la integración de Tracealyzer para tener las herramientas necesarias para la codificación.
- Siguiendo la guía [5] se completo la configuración necesaria para realizar instantáneas. En las primeras se identifico un problema del port a Cortex-M3 que fue solucionado gracias al FAQ de la página oficial [6].
- Finalmente se codifico la solución para ambos programas, en diferentes iteraciones.