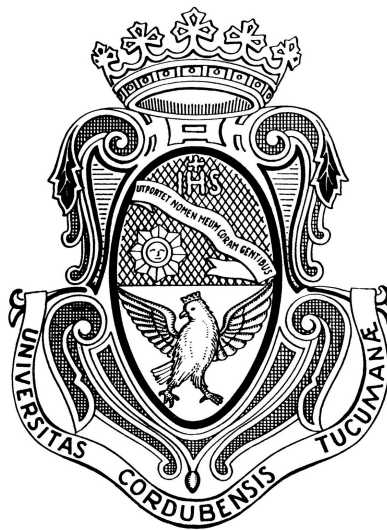


UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS
FÍSICAS Y NATURALES



SISTEMAS OPERATIVOS II

TRABAJO PRÁCTICO N°4: Tiempo real - FreeRTOS

Integrantes:

- Cazajous Miguel A. - 34980294

Córdoba - Argentina
9 de diciembre de 2018

Índice

1. Introducción:	1
1.1. Propósito:	1
1.2. Ambito del sistema:	1
1.3. Definiciones, acrónimos y abreviaturas:	1
1.4. Referencias:	2
1.5. Descripción general del documento:	3
2. Descripción general:	3
2.1. Perspectiva del producto:	3
2.2. Funciones del producto:	4
2.3. Características de los usuarios:	4
2.4. Restricciones:	4
2.5. Suposiciones y dependencias:	5
2.6. Requisitos futuros:	5
3. Requisitos específicos:	5
3.1. Interfaces externas:	5
3.2. Funciones:	6
3.3. Requisitos de rendimiento:	6
3.4. Restricciones de diseño:	6
3.5. Atributos del sistema:	7
4. Otros requisitos:	7
5. Diseño de solución:	7
6. Implementación y resultados:	8
6.0.1. Un productor y un consumidor:	9
6.0.2. Dos productores y un consumidor:	15
7. Conclusiones:	17
8. Apéndices:	19
8.1. Configuración FreeRTOS en Freescale K64:	19
8.2. Tracealyzer:	29
8.2.1. Trace view:	29
8.2.2. Communication flow:	30
8.2.3. CPU load graph:	30
8.2.4. Statistics report:	31
8.2.5. Memory heap utilization:	32

1. Introducción:

1.1. Propósito:

El propósito en este práctico, es la implementación en tiempo real, de una adquisición de datos desde dos dispositivos de entrada que pueden ser sensores, teclados, etc. La transmisión de estos datos debe ser de tal manera que se evite su pérdida o inconsistencia.

1.2. Ambito del sistema:

La aplicación está pensada para correr en una placa de desarrollo Cortex, implementando un comportamiento de tiempo real mediante la incorporación de FreeRTOS.

1.3. Definiciones, acrónimos y abreviaturas:

1. Tiempo real: Sistema que interactúa activamente con un entorno con dinámica conocida en relación con sus entradas, salidas y restricciones temporales, para darle un correcto funcionamiento de acuerdo con los conceptos de tiempo real.
2. FreeRTOS: Es un sistema operativo de tiempo real y es la solución estándar para pequeños procesadores.
3. Software: Referente a programas, aplicaciones.
4. Hardware: Componentes físicos, computadora, placa de desarrollo, etc.
5. Sistema Operativo: Software principal residente en un hardware que permite la gestión de sus partes y su interacción con aplicaciones de usuario. Es una capa intermedia entre aplicaciones de usuario y el hardware de la máquina.
6. Cortex: Forma parte de la familia de procesadores de la arquitectura ARM, que utiliza un reducido conjunto de instrucciones (RISC) de 32 y 64 bits.
7. UART: Transmisor-receptor asíncrono universal. Maneja las interrupciones de dispositivos conectados al puerto serie y convierte los datos en formato paralelo para pasarlo al bus del sistema, y de nuevo en serie para que puedan ser transmitidos a través de los puertos.



8. IDE: Entorno de desarrollo integrado. Es una aplicación que posee facilidades para el desarrollo de software.
9. SDK: Software Development Kit (Kit de desarrollo de software) es un conjunto de herramientas de desarrollo que le permite al programador crear una aplicación informática para un sistema concreto.

1.4. Referencias:

- [1] William Stallings. Sistemas Operativos 5º edición. 2005. ISBN: 84-205-44692-0.
- [2] Wikipedia. Tiempo real. URL: https://es.wikipedia.org/wiki/Tiempo_real.
- [3] freertos.org (Amazon). The FreeRTOS Kernel. URL: <https://www.freertos.org>.
- [4] Mendez Gonzalo. Especificacion de Requisitos segun el estandar de IEEE 830. URL: <https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf>.
- [5] Percepio. Tracealyzer 4. URL: <https://percepio.com/tracealyzer/>.
- [6] Wikipedia. UART. URL: https://es.wikipedia.org/wiki/Universal_Asynchronous_Receiver-Transmitter.
- [7] Wikipedia. Arquitectura ARM. URL: https://es.wikipedia.org/wiki/Arquitectura_ARM.
- [8] NXP semiconductors. MCUXpresso. URL: <https://mcuxpresso.nxp.com/en/welcome>.
- [9] NXP semiconductors. Importing an MCUXpresso SDK into MCUXpresso IDE. URL: <https://community.nxp.com/docs/DOC-334084>.
- [10] Percepio. Tracealyzer for FreeRTOS. URL: <https://percepio.com/docs/FreeRTOS/manual/>.
- [11] Erich Styger. mcuoneclipse. URL: https://github.com/ErichStyger/mcuoneclipse/blob/master/Examples/MCUXpresso/FRDM-K64F/FRDM-K64F_lwip_lwip_mqtt_bm/Generated_Code/trcConfig.h.
- [12] freertos.org - Amazon. FreeRTOS reference manual. URL: https://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V10.0.0.pdf.

1.5. Descripción general del documento:

Este documento se compone de 7 secciones principales, siendo la primera la introducción, donde se explica brevemente el fin del proyecto como así también sus requerimientos de forma muy general. En la segunda sección la descripción general del sistema con el fin de conocer las principales funciones que debe ser capaz de realizar, conocer a quién va dirigido, además de restricciones y supuestos que puedan afectar el desarrollo del mismo. En la sección 3 se definen de manera detallada los requerimientos del sistema a desarrollar, los que definen el comportamiento del sistema como así también otros requerimientos que puedan ser deseados considerando el uso que el sistema va a tener. En la cuarta sección se presenta el diseño del sistema que dará solución a los requerimientos antes enumerados. En la quinta sección se discuten los resultados de implementación del diseño elaborado en la etapa anterior donde se muestra como el sistema opera. En la sección 6 se elabora una breve conclusión acerca de la experiencia en la elaboración del proyecto. Finalmente en la última sección se agrega información alternativa que pueda ser de interés.

2. Descripción general:

2.1. Perspectiva del producto:

El producto debe ser capaz de obtener datos desde diversos dispositivos de entrada, o simular la adquisición de datos mediante software, siempre cumpliendo con las restricciones de tiempo real, evitando que datos se pierdan y/o sean inconsistentes.

2.2. Funciones del producto:

El sistema deberá ser capaz de:

1. Obtener o simular dispositivos de entrada como sensores o ingresos por teclado que generen datos.
2. Gestionar esos datos y transmitirlos por UART sin pérdida. Pueden ser mostrados por pantalla y/o almacenados en archivos.
3. Proveer, mediante el uso de alguna aplicación, información al usuario acerca del rendimiento de las tareas en el entorno de tiempo real.

2.3. Características de los usuarios:

No se especifican usuarios finales del producto, el mismo es una implementación que permita observar el comportamiento de un sistema de tiempo real. Aunque podría decirse que un producto de estas características se encuentran en innumerables dispositivos donde se requiera adquisición de datos e interacción con el medio ambiente. Sensores meteorológicos, dispositivos espaciales, etc. podrían ser productos estrechamente relacionados con la aplicación que aquí se desarrolla.

2.4. Restricciones:

La restricción está en que herramienta de tiempo real se debe usar. En este caso la aplicación debe ser enteramente desarrollada haciendo uso de FreeRTOS.

El lenguaje no tiene restricciones pero dado que la implementación que se obtiene de la página oficial de FreeRTOS está hecha en C, este es el lenguaje a usar. No impidiendo que pueda implementarse algún programa simple, dentro del proyecto, en otro lenguaje.

Los IDE de algunas placas de desarrollo solo tiene soporte para un número reducido de sistemas operativos. Por esa razón la elección de la placa influirá en el IDE a usar, y este afectará la elección del sistema operativo.

A su vez herramientas que se usan para observar el comportamiento de las tareas de tiempo real no tienen soporte para todos los sistemas operativos, siendo en la mayoría de los casos, desarrolladas para correr bajo Windows solamente.

2.5. Suposiciones y dependencias:

Como se mencionó antes hay una dependencia entre SOs, IDEs de las placas de desarrollo y herramientas de rastreo, cambiar un SO causaría que sea imposible correr la aplicación o se encontraría una reducción en sus prestaciones.

2.6. Requisitos futuros:

No se tiene pensado ningún requisito que deba cumplirse o que sería deseable en un futuro. La aplicación de simulación es muy específica, aunque sería deseable que la aplicación permita realizar algunos cambios, por ejemplo en la simulación de los dispositivos de entrada sin tener que rehacer todo el código.

3. Requisitos específicos:

3.1. Interfaces externas:

Interfaz de software:

La interfaz de software será una terminal capaz de mostrar lo que se recibe por el puerto serie, como Putty, Minicom, etc. o utilizar la IDE misma donde se desarrolla la aplicación.

Los dispositivos de entrada pueden ser completamente desarrollados en software sin la necesidad de contar con dispositivos físicos.

Interfaz de hardware y comunicación:

Un par de sensores y/o dispositivos de entrada irán conectados a una placa que implementará las tareas de tiempo real y manda a una computadora la información obtenida. Como se explicó en párrafos anteriores los sensores pueden ser simulados por software.

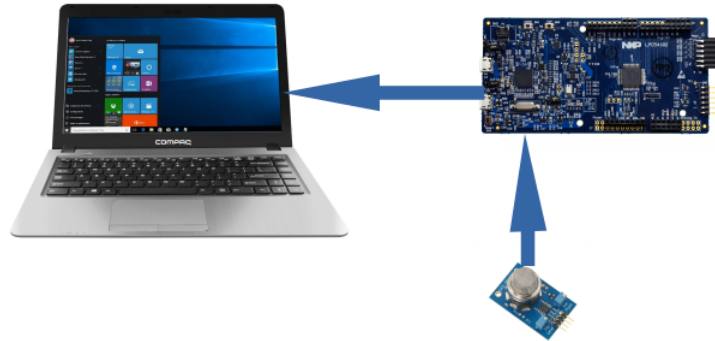


Figura 1: Imagen ilustrativa de ejemplo

3.2. Funciones:

- Leer valores del exterior mediante sensores y teclados o implementar una simulación de estos por software mediante el envío de datos aleatorios o definidos fijos previamente cada cierto tiempo.
- Agregar la lectura de valores como nuevas tareas para el planificador de tiempo real.
- Enviar mediante UART información obtenida a una computadora para que muestre en pantalla o guarde en un archivo los datos recibidos.

Además de las funciones propias de tiempo real se debe utilizar alguna herramienta de seguimiento que permita obtener información acerca de si las restricciones de tiempo real se están respetando o no.

3.3. Requisitos de rendimiento:

Los requisitos de rendimiento están ligados estrechamente con los que el tiempo real define. El sistema no debe ser necesariamente rápido, debe ser determinista y permitir que no hay pérdida de información.

3.4. Restricciones de diseño:

Como se mencionó antes debe utilizarse FreeRTOS y el uso de ciertos IDE están solo presentes en algunos SOs. Si bien existe una amplia gama de placas Cortex que pueden utilizarse siempre debe tenerse presente que el IDE puede no funcionar en el SO que se desee. En Windows se puede encontrar la mayor

compatibilidad general con los IDEs y otras herramientas relacionadas al uso de aplicaciones de tiempo real.

3.5. Atributos del sistema:

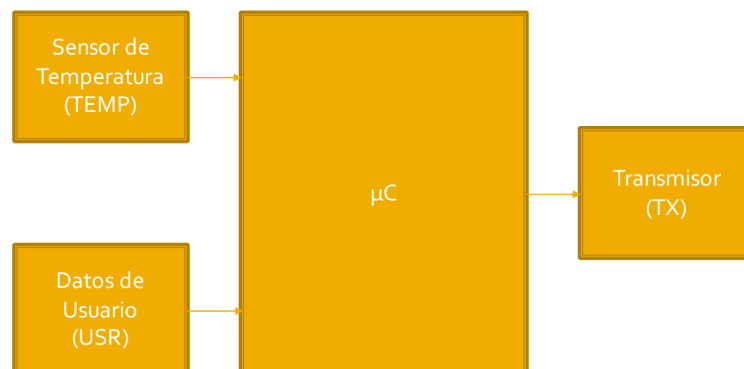
La placa de desarrollo debe ser Cortex, pero no hay restricción acerca de cual se deba usar. El hardware que recibe (computadora) simplemente debe ser capaz de poder conectarse a la placa mediante un IDE y mostrar lo que esta envía mediante Putty o similar.

4. Otros requisitos:

No se tienen otros requisitos presentes a la hora de elaborar este informe que puedan considerarse de importancia.

5. Diseño de solución:

Con un simple diagrama de bloques vemos que es lo que queremos implementar.



6. Implementación y resultados:

Al intentar correr el programa usando FreeRTOS aparecieron, luego de avanzar paso por paso con el debugger algunos errores debido a que no se encontraban ciertos archivos, se movió el proyecto a una carpeta sin espacios, lo que parece solucionó el problema.

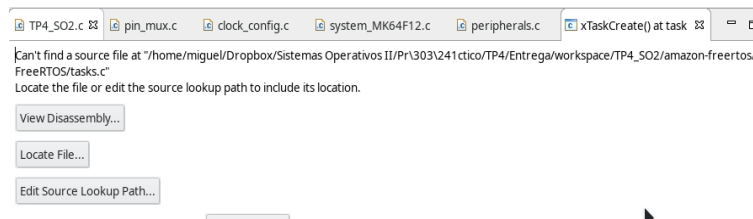


Figura 2: Mensaje de error

Luego se pudo conseguir que funcionara un programa “hello world” usando FreeRTOS mediante la inclusión de los headers de FreeRTOS.

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

Se declaro una función que será nuestra tarea a ser incluida en el planificador de tiempo real.

```
void vPrint(void *pvParameter){
    for(;;){
        printf("Hello World\n");
        vTaskDelay(500);
    }
}
```

El agregado de la siguiente línea fue necesario para que el programa funcionara, de lo contrario terminaba antes de agregar la tarea y la misma cumple la función de comenzar la grabación. Como dice la documentación [10] es la línea típicamente usada para el modo instantáneas (snapshot mode).

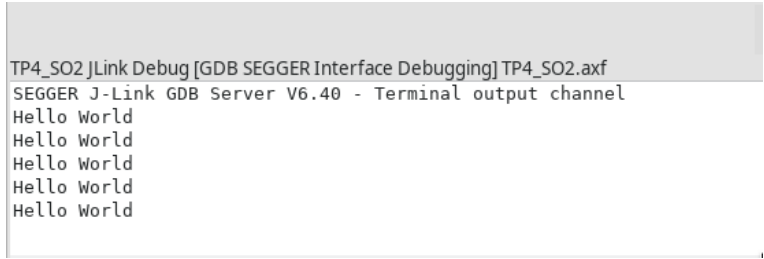
```
vTraceEnable(TRC_START);
```

Para agregar la tarea se usaron las siguientes líneas

```
xTaskCreate(vPrint, "vPrint", 240, NULL, 1, NULL);
vTaskStartScheduler();
```

Más adelante cuando mostremos las tareas principales se verá más de cerca las líneas anteriores.

El resultado del código es el siguiente:



```
TP4_SO2 JLink Debug [GDB SEGGER Interface Debugging] TP4_SO2.axf
SEGGER J-Link GDB Server V6.40 - Terminal output channel
Hello World
Hello World
Hello World
Hello World
Hello World
```

Figura 3: Salida

Luego se grabó una snapshot usando “save snapshot” en la pestaña de Percepio para visualizar los “traces”.

El único inconveniente es que en Linux el programa Tracealyzer se abre mediante un script, el cual no puede ser ejecutado desde el entorno de eclipse, por esa razón al grabar el programa no se abre automáticamente, sino que debemos hacerlo de forma manual. La captura que se tomó fue solo para verificar si el programa estaba funcionando adecuadamente.

6.0.1. Un productor y un consumidor:

Las primeras líneas con las que nos encontramos son para la creación de la cola de mensajes que van a usar las tareas para comunicarse.

```
QueueHandle_t xQueue;
xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
```

Donde las constantes se definen como:

```
#define QUEUE_LENGTH 10
#define QUEUE_ITEM_SIZE sizeof(AMessage)
```

Y “AMessage” es una estructura.

```
typedef struct A_Message
{
    char ucMessageID;
    char ucData[ 20 ];
} AMessage;
```

Las tareas se agregaron usando la misma función que se mostró anteriormente.

```
xTaskCreate(vProductor, "Productor", 240, (void *)xQueue, 1, NULL);
xTaskCreate(vConsumidor, "Consumidor", 240, (void *)xQueue, 2, NULL);
```

De acuerdo a la documentación [12] la definición de la función es:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask );
```

Cada uno de los 6 parámetros se comentan a continuación:

1. Es un puntero a una tarea que corre infinitamente.
2. Un nombre de esa tarea representado por una cadena
3. Le dice al kernel que tan grande hacer la pila para esa tarea. El valor especifica la cantidad de palabras que la pila puede tener, no la cantidad de bytes. El producto de la palabra por el valor no debe exceder el máximo tamaño que una variable tipo `size_t` puede almacenar. En nuestro caso es 240.
4. Parámetros pasados a la tarea. En nuestro caso pasamos la cola.
5. Define la prioridad de la tarea, con 0 para la menor prioridad y (`configMAX_PRIORITIES - 1`) para la mayor. Aquí el consumidor tiene una mayor prioridad.
6. Se pasa un manejador a la tarea para modificarle la prioridad o eliminarla.

Inicio del planificador:

```
vTaskStartScheduler();
```

Usualmente antes de ejecutar el planificador, el main o funciones que se llaman desde main serán ejecutadas, pero luego de ejecutar el planificador, solo las tareas que se implementaron e interrupciones son las que se van a ejecutar.

El planificador asigna la máxima prioridad a las tareas que se han creado.

Las tareas envían y reciben por medio de la cola usando las siguientes sentencias.

```
xQueueSendToBack( xQueue, &xMessage, 2000/portTICK_RATE_MS )  
xQueueReceive( xQueue, &xMessage, portMAX_DELAY )
```

La primera, como indica, se usa para enviar y la misma reemplaza a la anterior y según la documentación es la recomendada para usar.

Lo anterior se incluye dentro de un bloque condicional para evaluar si ocurrió o no un error. Si (!= pdPASS) ocurrió un error. Las definiciones son:

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,  
                             const void * pvItemToQueue,  
                             TickType_t xTicksToWait );
```

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,  
                          void *pvBuffer,  
                          TickType_t xTicksToWait );
```

1. Un manejador de la cola que se está usando.
2. Un puntero a memoria para leer o escribir lo que se envía.
3. Cantidad de ticks. Para envío es el tiempo que espera hasta que la cola sea vaciada. Para recepción es el tiempo que estará en estado de bloqueado esperando por información en la cola.

Communication flow:

Vemos a continuación que la comunicación entre productor y consumidor se lleva a cabo a través de una cola.

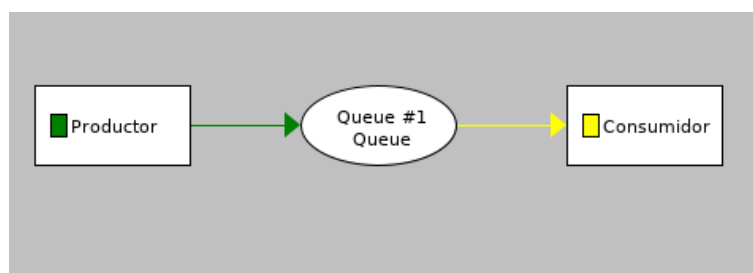


Figura 4: Communication flow

Trace view:

Podemos ver una serie de eventos representados cada uno con un color específico (ver anexo) entre los que se ven cuando los actores están listos, cuando se bloquean o cuando retornan exitosamente de una operación.

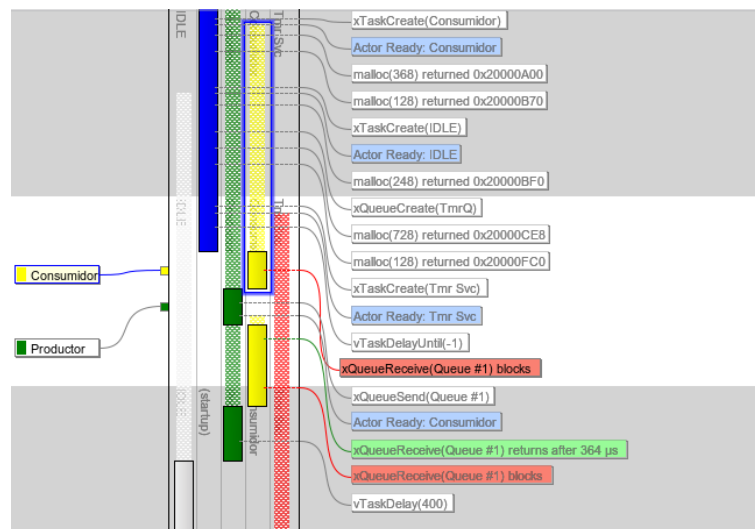


Figura 5: Trace view

CPU load:

Como podemos ver el uso de la CPU en este programa es extremadamente mínimo, donde gran parte se usa en el arranque, los valores después se reparten equitativamente entre productor y consumidor.

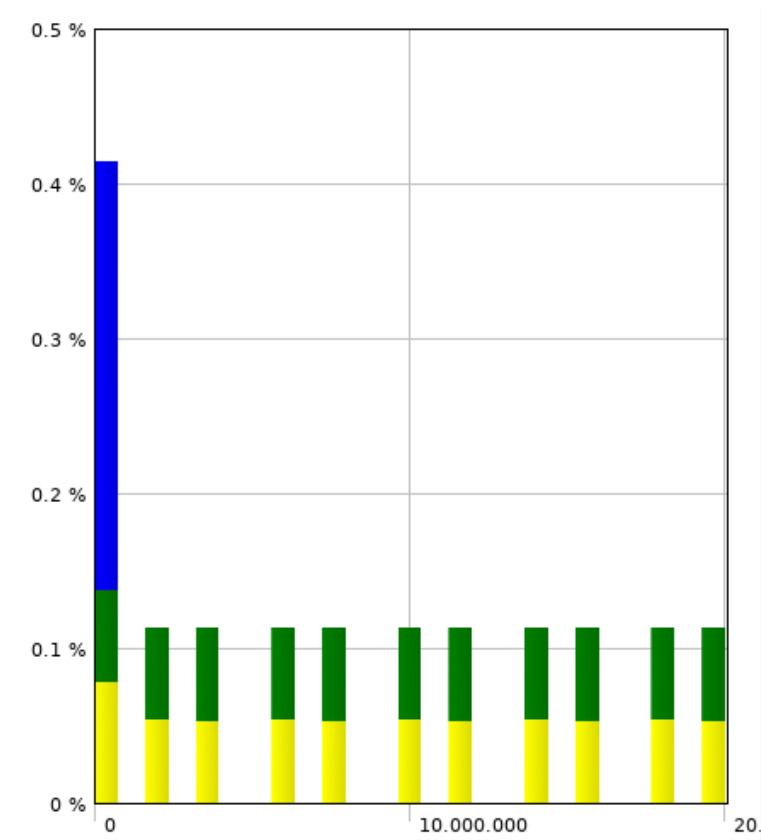


Figura 6: CPU load

Memory heap:

Lo importante en este gráfico es observar que la línea de uso de memoria no se dispare, lo que indicaría que hay una pérdida de memoria en la ejecución del programa.



Figura 7: Memory heap.

Statistics report:

Se muestra en el cuadro siguiente valores mínimos, promedios y máximos del tiempo de respuesta y el tiempo de ejecución. El primero es mayor que el segundo ya que también considera el tiempo en que el actor está listo para ejecutarse pero no lo está.

Actor	Count	CPU Usage	Execution Time			Response Time		
			Min	Avg	Max	Min	Avg	Max
IDLE	1	9.710	20.074.528	20.074.528	20.074.528	20.085.566	20.085.566	20.085.566
(startup)	1	48.555	2.222	2.222	2.222	2.222	2.222	2.222
Productor	1	18.129	480	480	480	2.711	2.711	2.711
Consumidor	2	23.607	196	312	429	479	934	1.390
Tmr Svc	1	0.000	0	0	0	20.084.936	20.084.936	20.084.936

Figura 8: Statistics report

6.0.2. Dos productores y un consumidor:

Igualmente a los casos anteriores las tareas se agregaron líneas similares.

Communication flow:

En este caso podemos ver que a diferencia del anterior encontramos un nuevo productor que escriben información en una cola, que el consumidor se encarga de leer.

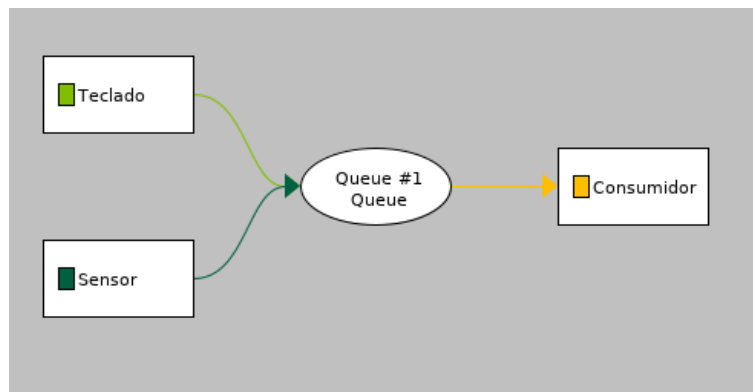


Figura 9: Communication flow.

Trace view:

Podemos ver igual que en el caso anterior los diferentes eventos que ocurren y se evidencia un poco más el solapamiento de tareas, al tener una más. La prioridad del consumidor es mayor que la de los dos productores.

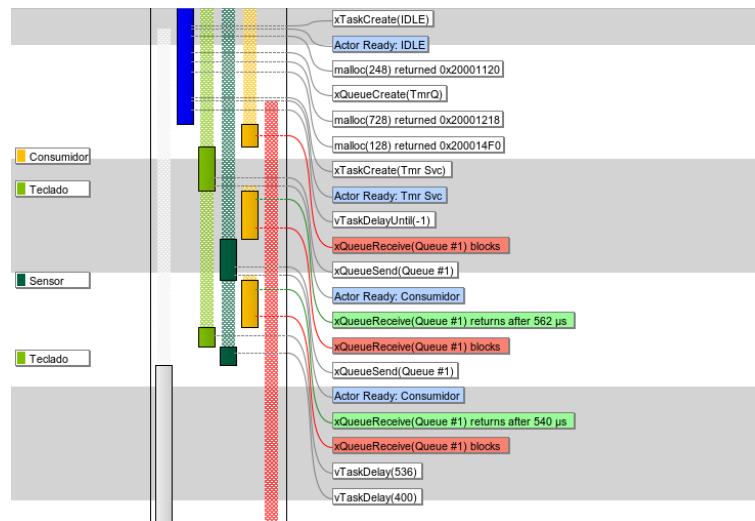


Figura 10: Trace view.

CPU load:

Es este caso vemos un gráfico de cpu un poco más complejo que anteriormente. Vemos que en intervalos donde ocurren las dos escrituras el consumidor que es el encargado de leerlas aumenta al doble.

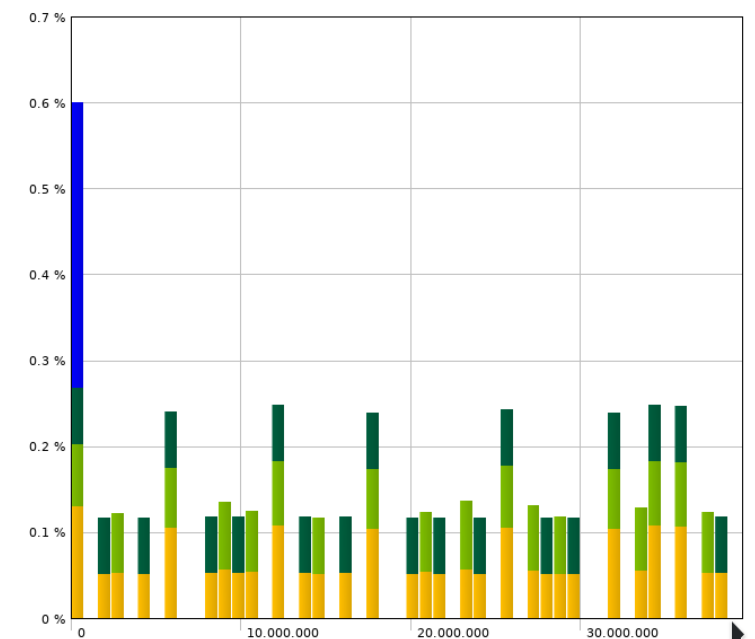


Figura 11: CPU load

Memory heap:

Como se puede ver en la siguiente imagen al igual que en el ejemplo anterior se mantiene constante.



Figura 12: Memory heap.

Statistics report:

Al igual que en el caso anterior podemos ver los valores máximo, mínimo y promedios de los tiempos de respuesta y ejecución

Actor	Count	CPU Usage	Execution Time			Response Time		
			Min	Avg	Max	Min	Avg	Max
IDLE	1	27.922	39.457.930	39.457.930	39.457.930	39.495.446	39.495.446	39.495.446
(startup)	1	23.782	2.629	2.629	2.629	2.629	2.629	2.629
Sensor	1	11.893	521	521	521	3.731	3.731	3.731
Teclado	1	12.868	564	564	564	3.977	3.977	3.977
Consumidor	3	23.534	197	344	421	463	777	1.396
Tmr Svc	1	0.000	0	0	0	39.494.813	39.494.813	39.494.813

Figura 13: Statistics report

7. Conclusiones:

Se pusieron en práctica diversos conceptos aprendidos teóricamente y se hizo frente a una gran cantidad de problemas y/o inconvenientes que se presentaron a la hora de utilizar placas de desarrollo, mayormente relacionados a la correcta configuración del dispositivo con el IDE, que se usa para el desarrollo de la aplicación, con el SDK que incluye FreeRTOS y el plugin/herramienta para realizar el rastreo.

El trabajo sirvió para comprender el campo de aplicación de los sistemas operativos de tiempo real y de su importancia en áreas donde la pérdida de datos puede ser un problema con consecuencias graves.

Si bien las tareas programadas no representan una gran carga y pueden ser implementadas tranquilamente en un sistema de no tiempo real, se logró visualizar como estas interactúan.

8. Apéndices:

8.1. Configuración FreeRTOS en Freescale K64:

A continuación se mostrarán los pasos necesarios para tener funcionando el IDE en conjunto con el plugin y aplicación de rastreo, incluyendo los archivos y programas necesarios y su posterior configuración.

El procedimiento es el siguiente para configurar lo necesario bajo Linux.

1. En primer lugar descargamos el IDE que se va a utilizar. En este caso se eligió MCUXPRESSO [8]. Teniendo presente que se debe bajar la versión de Linux 64.
2. Descargamos luego Tracealyzer [5] llenando el formulario que nos proponen.

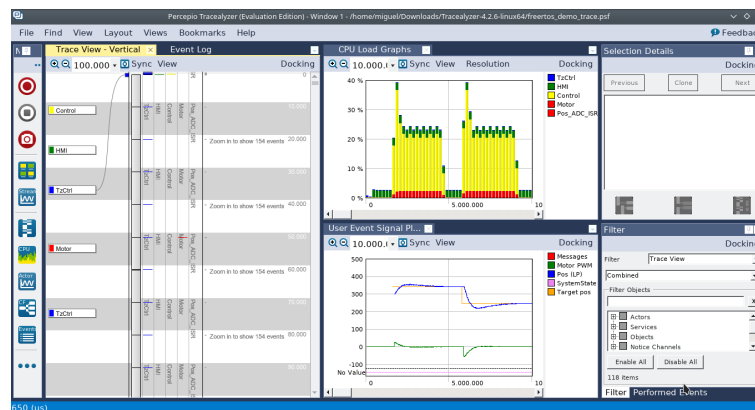
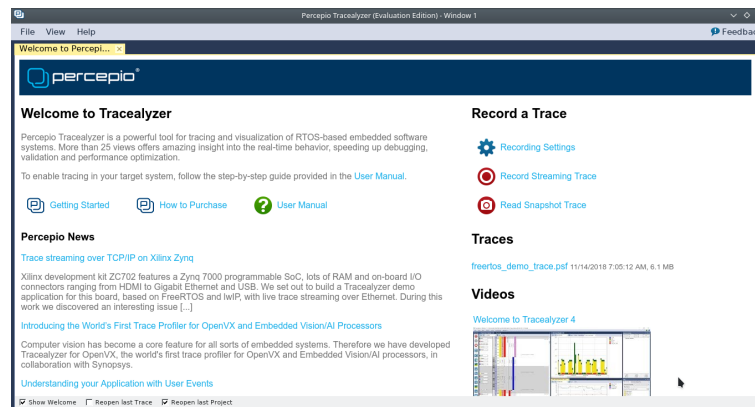
Target Platform*	FreeRTOS (Tracealyzer v4 ▾)
Host OS*	Linux (64-bit) ▾
Name (First, Last)*	<input type="text"/>
Email*	<input type="text"/>
Company	<input type="text"/>
Phone	<input type="text"/>
Country	Argentina ▾

(* required field)

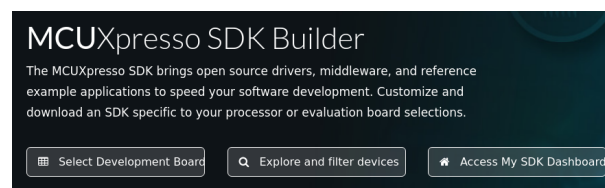
Llenamos además nuestros datos personales (nombre, apellido y dirección de correo electrónico).

Luego de que se llena el formulario se nos envía al correo electrónico un link de descarga y un serial como licencia de prueba que dura solo unos días.

Al finalizar la instalación se puede abrir el programa corriendo el script **launch-tz.sh** habiendo instalado antes la aplicación **mono**, y ejecutar un ejemplo que nos brinda la aplicación.



3. Procedemos ahora a descargar el SDK, también desde [8]. Seleccionamos “Select Development Board”



Lo anterior nos redirecciona a la siguiente dirección, en donde elegimos “FRDM-K64F”

Select Development Board

Search for your board or kit to get started.


Search by Name


Select a Device, Board, or Kit


▼ Boards
▼ Kinetis
FRDM-K22F
FRDM-K28F
FRDM-K28FA
FRDM-K64F
FRDM-K66F
FRDM-K82F
FRDM-KE02Z40M
FRDM-KE04Z

Luego presionamos en “Build MCUXpresso SDK”

Actions

 [Build MCUXpresso SDK](#)

 [Explore selection with Clocks tool](#)

 [Explore selection with Pins tool](#)

Finalmente seleccionamos el sistema operativo host (Linux) y en “Add software component” agregamos **Amazon-FreeRTOS Kernel**


Developer Environment Settings

Selections here will impact files and examples projects included in the SDK and Generated Projects

Host OS	Toolchain / IDE
<div>Linux ▼</div>	<div>MCUXpresso IDE ▼</div>

Select Optional Middleware

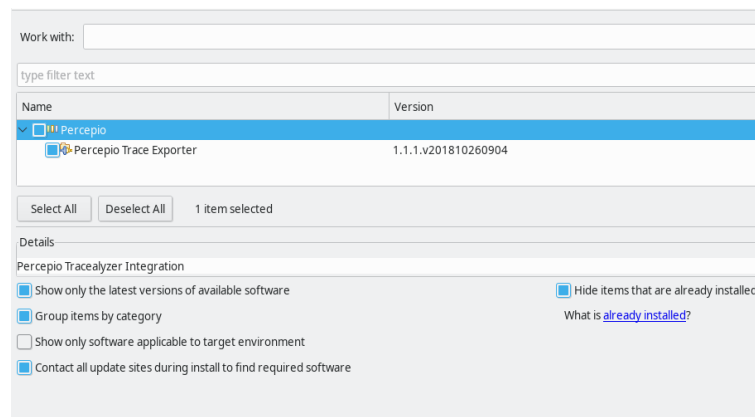
Add middleware, operating systems, and software libraries to your SDK.

 Add software component

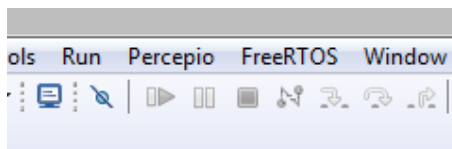
Luego podemos descargar el archivo.

4. Dentro del IDE MCUXpresso podemos incluir un plugin del Tracealyzer.

Nos dirigimos a **Help-Install new software...** ponemos la dirección <http://percepio.com/exporter> e instalamos.



Cuando termine la instalación nos muestra una nueva pestaña llamada **Percepio**.

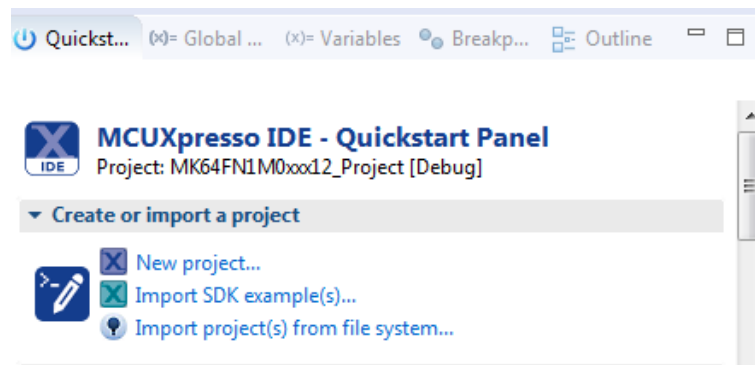


5. Antes de crear un proyecto debemos realizar lo siguiente. Como indica en el siguiente enlace [9] importamos el SDK descargado anteriormente al IDE MCUXpresso. Siguiendo lo que dice en el enlace copiamos el archivo en la ruta.

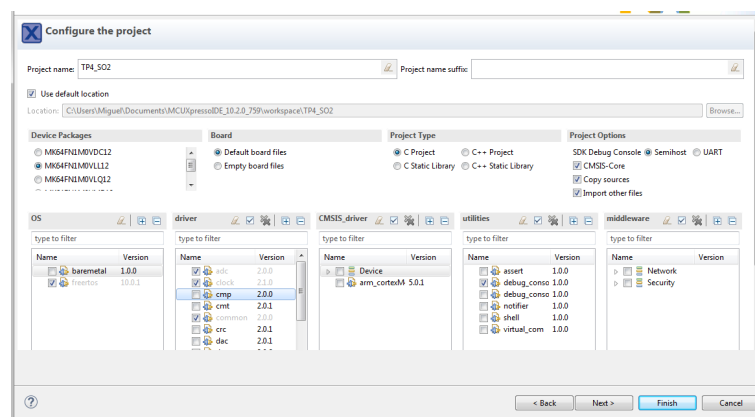
PATH

/home/user_name/mcuxpresso/01/SDKPackages/

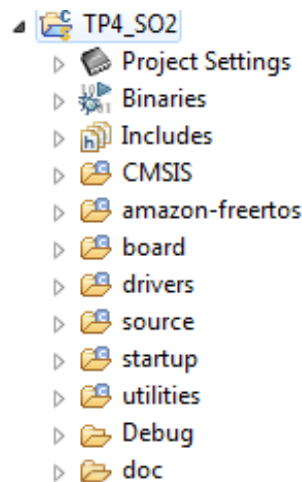
6. Para crear un nuevo proyecto vamos a **New project** en el panel de acceso rápido que nos muestra la IDE abajo a la izquierda.



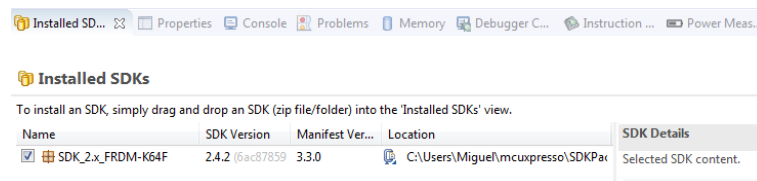
Ahora que tenemos el SDK instalado podemos elegir la placa **frdmk64f** y nos saldrá lo siguiente.



Elegimos un nombre de proyecto, cambiamos el OS de baremetal a freertos y finalizamos. Ya tendremos así nuestro proyecto en el workspace.

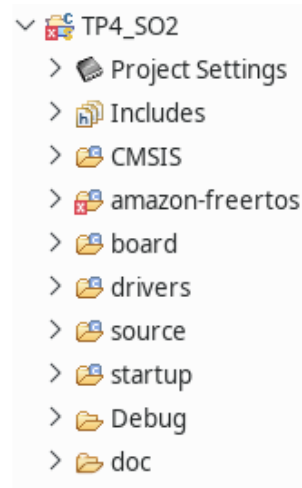


Vemos en el IDE también el SDK instalado



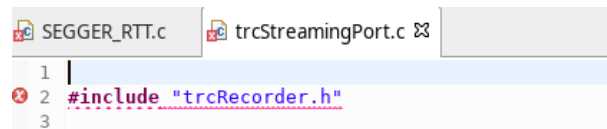
7. Debemos ahora permitir que tracealyzer grabe eventos mientras ejecutamos el debug en la freescale k64. Para ellos hacemos lo siguiente, basándonos en el manual [10].

Debemos primero descargar el archivo y descomprimirlo, de acuerdo a lo que propone el manual en la carpeta de **amazon-freertos**. Algunas carpetas dentro de **streamports** no son necesarias pues no vamos a realizar un seguimiento de esa forma en nuestro proyecto, sino que vamos a hacer uso de las snapshots.



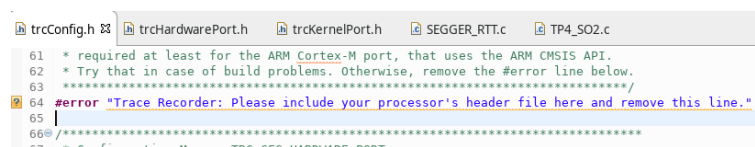
Como se puede ver al refrescar y compilar el proyecto surgen una serie de errores provenientes de la carpeta **TraceRecorder** que acabamos de agregar.

Específicamente, los errores vienen dados por la mala inclusión de headers como se puede apreciar en la siguiente figura.



Debemos colocar la ruta donde se encuentran esos archivos a partir de la carpeta de proyecto (TP4_SO2).

Luego de arreglar las rutas de los headers vemos otros inconvenientes que se presentan.



Buscando el header de la placa dentro del archivo TP4_SO2.c arreglamos el problema.

```
trcConfig.h 83 trcHardwarePort.h trcKernelPort.h SEGGER_RTT.c TP4_SO2.c
61 * required at least for the ARM Cortex-M port, that uses the ARM CMSIS API.
62 * Try that in case of build problems. Otherwise, remove the #error line below.
63 *****
64 #include "MK64F12.h"
65 /*****
66 * Configuration Macro: TRC_CFG_HARDWARE_PORT
67 *
```

Veamos ahora unas capturas de los siguientes y últimos errores con que nos enfrentamos al compilar el proyecto.

En el archivo **trcHardwarePort.h**

```
51 #if (TRC_CFG_HARDWARE_PORT == TRC_HARDWARE_PORT_NOT_SET)
52 #error "TRC_CFG_HARDWARE_PORT not selected - see trcConfig.h"
53 #endif
54 |
55 /*****
```

En el archivo **trcHardwarePort.h**

```
124 #if (TRC_CFG_HARDWARE_PORT == TRC_HARDWARE_PORT_NOT_SET)
125 #error "TRC_CFG_HARDWARE_PORT not selected - see trcConfig.h"
126 #endif
127 |
```

Los dos problemas anteriores se pueden solucionar modificando el archivo trcConfig.h

```
82 *****
83 #define TRC_CFG_HARDWARE_PORT TRC_HARDWARE_PORT_NOT_SET
84
85 /*****
```

cambiando “TRC_HARDWARE_PORT_NOT_SET” por “TRC_HARDWARE_PORT_ARM_Cortex_M” de acuerdo a lo que se indica en este ejemplo de configuración [11]

En el archivo **trcKernelPort.h**

```
47 #if (defined(TRC_USE_TRACEALYZER_RECORDER) && configUSE_TRACE_FACILITY == 1)
48 #error Trace Recorder: You need to include trcRecorder.h at the end of your FreeRTOSConfig.h!
49 #endif
50
```

Este mensaje es bastante claro, nos indica que debemos incluir algo en el archivo de configuración de FreeRTOS, y volviendo al manual [10] en la parte de integración del tracealyzer en el IDE, específicamente el punto 7 y 8 nos dice que debemos agregar las siguientes líneas.

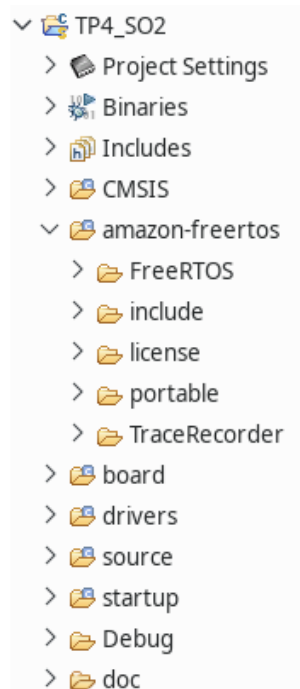
```
/* Integrates the Tracealyzer recorder with FreeRTOS */  
#if ( configUSE_TRACE_FACILITY == 1 )  
#include "trcRecorder.h"  
#endif
```

Recordando que debemos corregir la ruta colocamos lo anterior en el archivo FreeRTOSConfig.h

```
142  
143 /* Integrates the Tracealyzer recorder with FreeRTOS */  
144 #if ( configUSE_TRACE_FACILITY == 1 )  
145 #include <amazon-freertos/TraceRecorder/include/trcRecorder.h>  
146 #endif  
147
```

Al finalizar estas configuraciones, salieron algunos problemitas más respecto al mal seteo de las rutas de los headers, que se solucionaron como hicimos al principio.

Finalmente obtenemos el proyecto compilado con los archivos de **TraceRecorder** agregados.



De esta manera se solucionan todos los problemas de configuración y se está en condiciones de empezar un proyecto.

8. Al momento de conectar la placa y apretar en el acceso rápido para debuggear que posee la IDE.



Nos saldra un mensaje que dice.

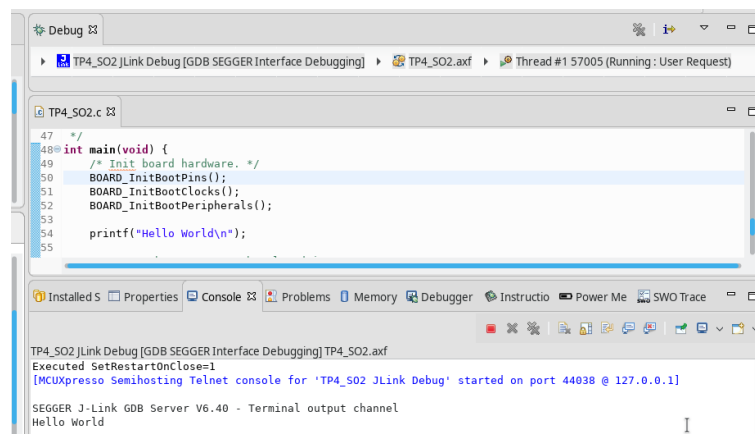
ERROR!

J-Link support unavailable

Apretamos el logo azul de debug en el acceso rápido y nos pide que ubiquemos un archivo.

`/opt/SEGGER/JLink/JLinkGDBServerCLExe`

Y finalmente podemos ejecutar nuestro programa.



8.2. Tracealyzer:

Vamos a dar un breve resumen de algunas de las diferentes capturas que pueden obtenerse con el software indicando que es lo que está representando.

8.2.1. Trace view:

Muestra todos los eventos grabados en una línea temporal vertical en donde se muestran los principales actores representados por bloques de colores según su prioridad (rojo mayor prioridad, azul menor prioridad). En general siguiente el espectro de la luz).

Al alejarse se hace un filtrado donde se muestran los más importantes, además de que se ocultan las repeticiones de los eventos para dejar lugar a los menos frecuentes.

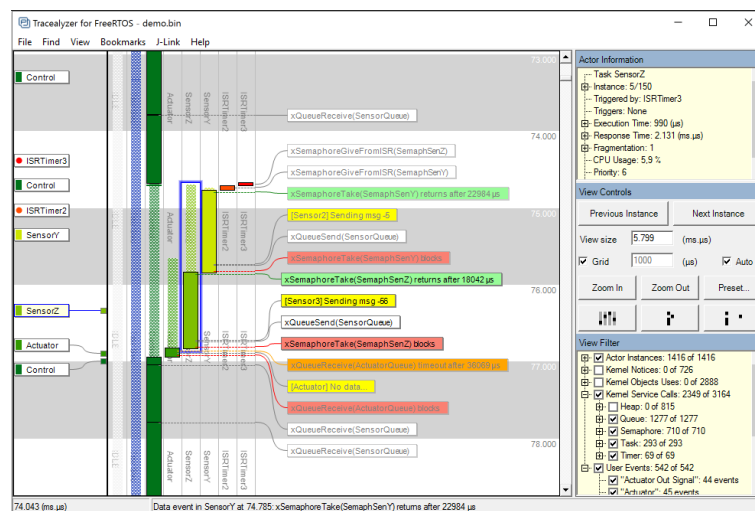


Figura 14: Vista principal

Seleccionando en cada actor podemos ver los diferentes eventos, también representados por colores.

1. Rojo: Llamadas bloqueantes al kernel
2. Verde: Retorno exitoso de la anterior
3. Blanco: Llamadas al kernel que se completaron sin bloquearse
4. Anaranjado: Llamadas al kernel que retornaron debido a un timeout

5. Amarillo: Eventos de usuario.
6. Celeste: Notificaciones de kernel, por ejemplo “actor listo”, cuando una tarea está lista para ejecutar.

8.2.2. Communication flow:

Ofrece una vista rápida de de la comunicación y sincronización entre actores en una traza, a través de mensajes por colas, mutex y/o semáforos.

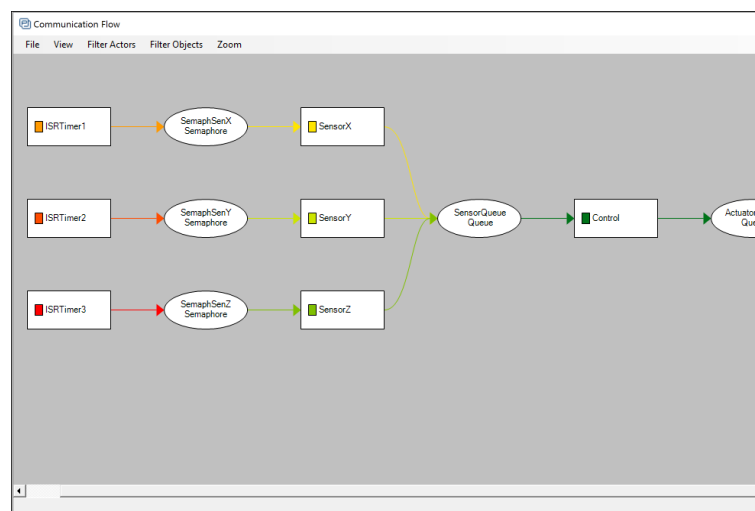


Figura 15: Communication flow

8.2.3. CPU load graph:

Muestra el uso de CPU a través del tiempo por actor y en total. Por defecto muestra a todos los actores.

Evalúa la traza total dividiéndola en intervalos. Por defecto se divide en 100 intervalos.

Los rectángulos que muestran la carga por los actores y la altura de cada uno representa la carga para ese actor en particular.

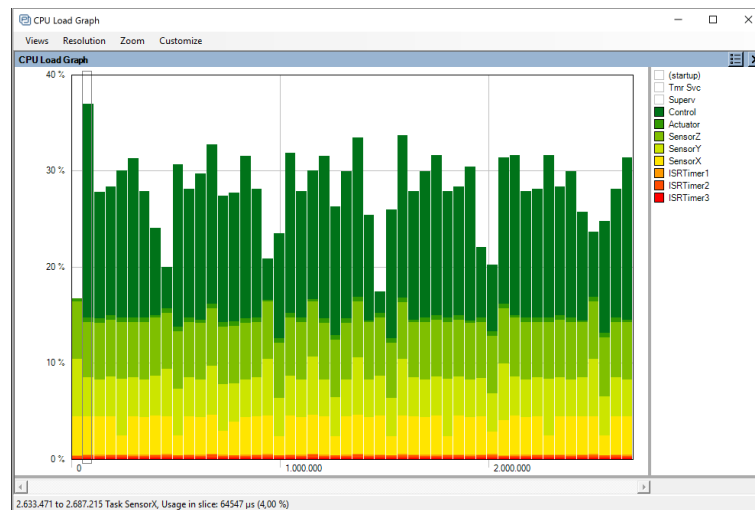


Figura 16: CPU load graph.

8.2.4. Statistics report:

Provee información acerca del máximo, mínimo y promedio de una serie de propiedades de las tareas en ejecución, como tiempo de respuesta y tiempo de ejecución.

Se pueden generar reportes estadísticos usando la opción “View” en el menú. Las propiedades son:

1. Prioridad: La prioridad del actor.
2. Count Número de instancia del actor en la traza o en el intervalo seleccionado.
3. Uso de CPU: El uso relativo del procesador por cada actor en porcentaje.
4. Tiempo de ejecución: Tiempo en microsegundos para ejecutar una instancia del actor, excluyendo tiempo gastado en expulsión de tareas.
5. Tiempo de respuesta: El tiempo desde el comienzo hasta que finaliza una instancia del actor. El evento “Actor ready” es considerado como comienzo de la instancia.
6. Periodicidad: El tiempo entre dos instancias adyacentes del mismo actor, medido desde el comienzo de una hasta el comienzo de la otra.
7. Separación: Tiempo entre dos instancias adyacentes del mismo actor, medido desde el final de una hasta el comienzo de la otra.

8. Fragmentación: El número de fragmentos planificados de la instancia. El valor indica el número de interrupciones, por ejemplo ISR, expulsiones por tareas de mayor prioridad, bloqueo en mutex, etc.

Statistics Report Viewer

Actor	Priority		Count	CPU Usage		Execution Time			Response Time			Periodicity		Separation			Fragmentation		
	Min	Max		%	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Task Superv	2	3	68	4.750	108	1.877	2.081	108	4.980	12.097	39.729	39.996	40.010	27.895	34.974	39.621	1	1,51	3
Task Control	4	4	393	13.111	40	896	1.087	40	990	3.255	745	6.760	59.770	0	5.768	59.729	1	1,21	3
Task Actuator	5	5	90	0.429	60	128	150	60	245	1.278	4.188	29.775	35.000	2.910	29.530	34.845	1	1	1
Task SensorZ	6	6	150	5.904	990	1.058	1.112	1.000	1.571	3.148	15.869	17.986	18.002	14.852	16.412	16.872	1	1	1
Task SensorY	7	7	112	4.447	1.001	1.067	1.100	1.002	1.311	2.130	22.870	23.990	24.043	21.868	22.678	22.913	1	1	1
Task SensorX	8	8	90	3.592	1.016	1.072	1.124	1.052	1.107	1.130	29.923	30.000	30.077	28.870	28.893	28.947	1	1	1

Figura 17:

Las expresiones para el tiempo se expresan en microsegundos.

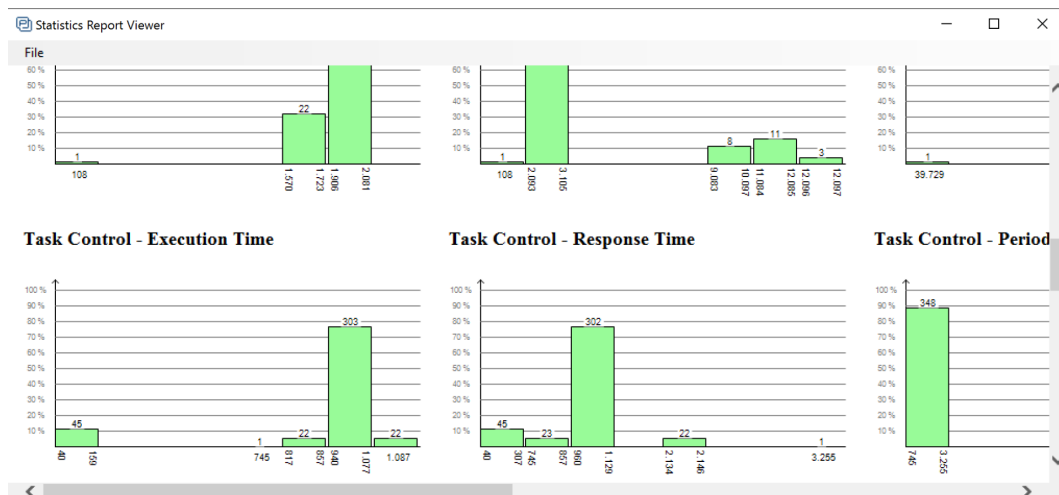


Figura 18:

8.2.5. Memory heap utilization:

Muestra la utilización de memoria a lo largo del tiempo. Cada punto corresponde a una operación de memoria dinámica como una llamada a malloc()

o free()).

Se usa para observar si no hay alguna pérdida de memoria, por ejemplo si el gráfico sigue creciendo indefinidamente.

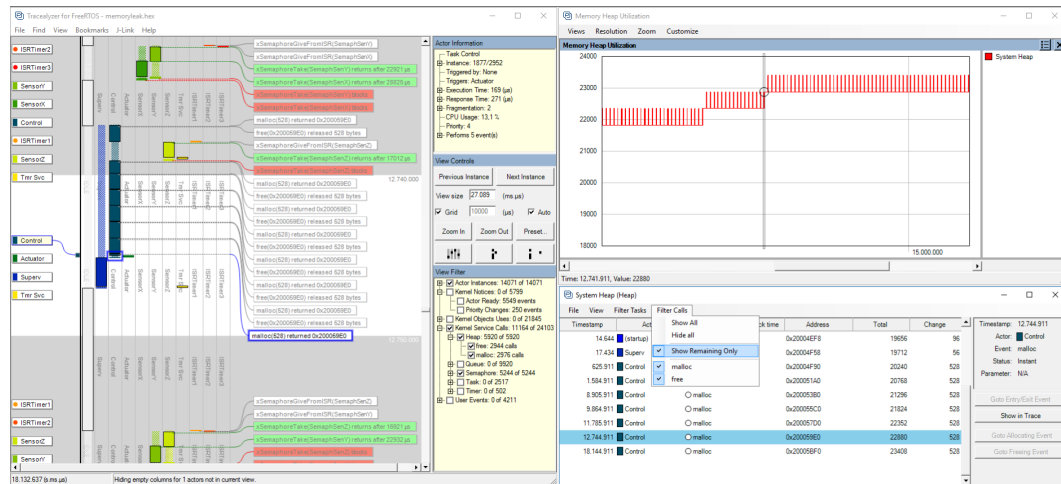


Figura 19: Memory heap utilization.