ITESO, Universidad Jesuita de Guadalajara

Sistemas Embebidos Basados en Microcontroladores 2

Prof. Rodrigo Aldana López

Práctica 1:
Sistema de Comunicación Cliente-Servidor

José Andrés Hernández Hernández ie704453

ie704453@iteso.mx

Eric Guedea Osuna ie717466

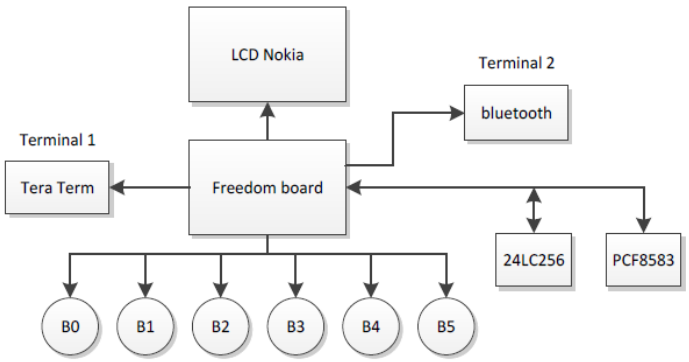
ie717466@iteso.mx

Descripción

Consiste en la implementación de un sistema comunicación cliente-servidor. Este sistema cuenta con 2 terminales de acceso al servidor, una a través de una terminal 1 en la PC (Tera Term) y la terminal 2 que se accede a través de un dispositivo móvil como un Smartphone.

El sistema a diseñar está compuesto por

- Servidor (K64)
- Módulo de bluetooth HC05
- Un display Nokia 5110 (terminal del servidor)
- Botones B0-B5
- Una memoria EEPROM 24LC256
- Un Reloj de tiempo real PCF8583



En cada terminal del sistema de comunicaciones se desplegara el siguiente menú:

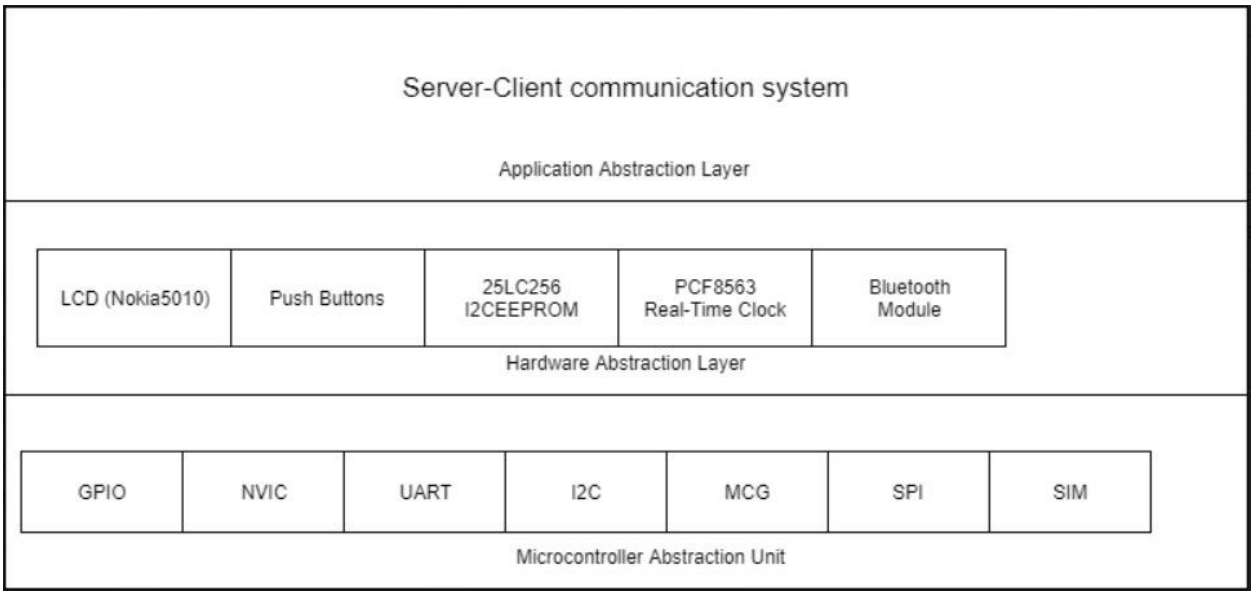
Opciones:

1) Leer Memoria I2C
2) Escribir memoria I2C
3) Establecer Hora
4) Establecer Fecha
5) Formato de hora
6) Leer hora
7) Leer fecha
8) Comunicación con terminal 2
9) Eco en LCD

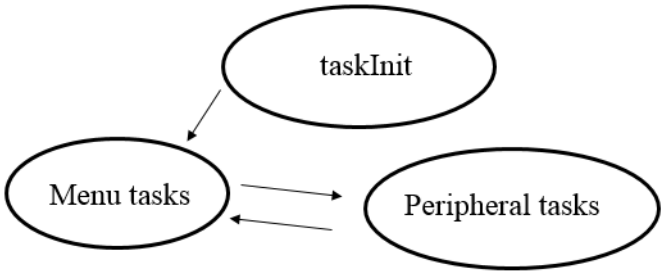
Desarrollo

Para el desarrollo de esta práctica fue necesario realizar el diseño de capas de las diferentes tareas que juegan el rol en esta práctica, en nuestro caso, inicializábamos una tarea que iniciaba a las demás tareas y además creaba los semáforos, las colas y los eventos.

En este caso las primeras tareas que se ejecutaban eran las tareas de los menús que se imprimían en la UART, y además daban apertura a las tareas de los periféricos, en este caso el principal periférico que se utilizó fue la del I2C, con el RTC y la memoria E2PROM. El modelo de capas se describe en la siguiente figura.



Este diagrama mostrado representa a nivel capas los diferentes periféricos y hardware utilizado en el proyecto, en el siguiente diagrama se muestra el flujo pero del sistema operativo.



Para la ejecución del programa se necesito las siguientes librerías que nos permitían hacer uso de los elementos del sistema operativo y de los periféricos utilizados, se cuido bastante el uso de definiciones múltiples con las librerías de manera de eliminar posibles errores.

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>
#include "board.h"
#include "peripherals.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "MK64F12.h"
#include "LCDNokia5110.h"
#include "Buttons.h"
#include "fsl_uart.h"
#include "fsl_port.h"
#include "fsl_debug_console.h"
#include "FreeRTOS.h"
#include "semphr.h"
#include "queue.h"
#include "event_groups.h"
#include "Menu.h"
#include "I2C.h"
```

El uso de callbacks fueron importantes para los periféricos de UART e I2C de manera que funcionaran de manera ágil y fueran lo menos bloqueante posible. La estructura de la creación de tareas fueron de la siguiente manera.

```
/*******************************READ I2C TASKS*****************************/

xTaskCreate(taskREADI2C_Read, "ReadI2C_Read",
        (3*configMINIMAL_STACK_SIZE), NULL, configMAX_PRIORITIES-1, NULL);
```

El tercer parámetro de la tarea es el tamaño de stak utilizado, la macro utilizada para ese parámetro es el mínimo de stack, el cual es 90, en este caso se multiplica por 3, de manera que en total queda de 270. El penúltimo parámetro nos indica la prioridad de la tarea, en este caso tiene un valor por default de 5, por lo que la tarea de menor tamaño tienen mayor prioridad.

Los siguiente por revisar son los callbacks, los cuales tienen una estructura estándar donde espera a que se realice la acción del periférico de manera que activa una bandera que funciona como una variable global y es entonces cuando el periférico puede realizar su acción específica.

```
static void uart0_transfer_callback(UART_Type *base, uart_handle_t *handle,
        status_t status, void *userData)
{
    if (kStatus_UART_RxIdle == status)
    {
        rx0Finished = true;
    }
}
```

Lo siguiente por revisar son las estructuras de las tareas, las cuales tienen cierto stack además de estar encerrados en un *while* infinito, de manera que se ejecute las veces que sea necesario hasta que un semáforo, evento o interrupción lo interrumpa o lo saque de rutina.

```
void taskMENU_SetDate(void *arg)
{
    Time_Type time;
    uint8_t phase = 0;
    for(;;)
    {
        /**Wait the event flag to continue the task*/
        xEventGroupWaitBits(g_eventsMenus,
                (EVENT_DATE_MENU), pdTRUE,
                pdTRUE, portMAX_DELAY);
#if 0
        /**Print the time in LCD*/
        time = getTime();
        printTimeLCD(time);
#endif
        /**Print in the UART for phases*/
        menu_SetDate0(phase);

        /**Set the flag event to jump to the next task*/
        xEventGroupSetBits(g_eventsSetDate, EVENT_DATE_SET);

        xSemaphoreTake(g_semaphore_SetDate, portMAX_DELAY);
        xSemaphoreGive(g_semaphore_Init);
    }
}
```

Como podemos observar en la imagen anterior, podemos ver que la tarea tiene sus variables locales que se alojan en su stack, tiene eventos y semáforos en espera, y a su vez tiene un *give* de un semáforo de manera que le pasa la rutina otra tarea, y es así como el sistema operativo funciona, además es posible que en una tarea corran dos tipos de entidades, es decir, que la tarea puede tener mas de dos stacks pertenecientes a diferentes procesos.