SETR 1 - Memoria de prácticas

Bermudo Garay, Pablo Manuel
López Moreno, José Manuel

Índice

1. Práctica 1 **4**
- 1.1. Objetivos . . . 4
- 1.2. Introducción . . . 4
- 1.3. Desarrollo de la práctica . . . 4
- 1.4. Conclusiones . . . 6

2. Práctica 2 **6**
- 2.1. Objetivos . . . 6
- 2.2. Introducción . . . 7
- 2.3. Desarrollo de la práctica . . . 7
- 2.4. Conclusiones . . . 9

3. Práctica 3 **9**
- 3.1. Objetivos . . . 9
- 3.2. Introducción . . . 10
  - 3.2.1. Clasificación general de los dispositivos USB . . . 10
  - 3.2.2. Los dispositivos de clase HID (Human Interface Device) . . . 10
  - 3.2.3. Acelerómetro/inclinómetro . . . 12
- 3.3. Desarrollo de la práctica . . . 13
- 3.4. Conclusiones . . . 19

4. Práctica 4 **19**
- 4.1. Objetivos . . . 19
- 4.2. Introducción . . . 19
- 4.3. Desarrollo de la práctica . . . 20
- 4.4. Conclusiones . . . 22

5. Práctica 5 **22**
- 5.1. Objetivos . . . 22
- 5.2. Introducción . . . 22
- 5.3. Desarrollo de la práctica . . . 23
- 5.4. Conclusiones . . . 29

6. Práctica 6 **29**
- 6.1. Objetivos . . . 29
- 6.2. Introducción . . . 29
- 6.3. Desarrollo de la práctica . . . 30
- 6.4. Conclusiones . . . 33

7. Práctica 7 **34**
- 7.1. Objetivos . . . 34
- 7.2. Introducción . . . 34
- 7.3. Desarrollo de la práctica . . . 34
- 7.4. Conclusiones . . . 39

8. Práctica 8 **40**
- 8.1. Objetivos . . . 40

8.2. Introducción 40
8.3. Desarrollo de la práctica 40
8.4. Conclusiones 42

1. Práctica 1

1.1. Objetivos

- Entender el esquema de la placa de Innovación Docente de ATC - Manejar el software de desarrollo/depurador - Manejar la utilidad de configuración de Silabs - Entender y manejar los displays LCDs. - Probar algunos dispositivos del microcontrolador: puertos, timer y conversores A/D.

Un aspecto importante en el desarrollo con microcontroladores es saber utilizar y aprovechar el software elaborado por otros y que nos pueda venir bien al proyecto que se pretenda hacer. Este software elaborado previamente puede tener diferentes orígenes: bajado de internet de la página de ejemplos del fabricante, sintetizado por una herramienta de generación de código automática (como la utilidad de configuración), comprado a otro desarrollador, etc. Como es natural la documentación y claridad del código depende mucho de su origen, de la complejidad del mismo e incluso de su tamaño. En esta práctica vamos a utilizar código de diferentes fuentes y adaptarlo para ir desarrollando los ejemplos que se proponen. Por tanto va a ser más una labor de análisis y adaptación del software existente que no el escribir muchas líneas en una pantalla en blanco.

1.2. Introducción

Para el desarrollo de la práctica en primer lugar el profesor nos pone en contexto sobre los elemento que vamos a utilizar y las principales características del microcontrolador 8051 de Silabs el cual es el que vamos a usar en la mayor parte de las prácticas de la asignatura. Como elemento a destacar el 8051 viene provisto de un "crossbar" el cual es el elemento que nos permite configurar de forma muy flexible los puertos de entrada/salida del microcontrolador, evitando así tener que usar pines predeterminados para utilizar los recursos de los que provee el dispositivo.

Además del manejo del 8051 el profesor nos introdujo el método de funcionamiento de los Liquid-crystal displays (LCD's). Este dispositivo consta de dos filas y aunque la memoria puede almacenar hasta cuarenta columnas de caracteres en cada fila, en la pantalla solo muestran dieciséis. Las columnas mostradas se seleccionan mediante un puntero que apunta a la primera columna que se desea mostrar.

1.3. Desarrollo de la práctica

Durante la práctica conocimos los fundamentos básicos básicos del trabajo con microcontroladores, para ello interactuamos con los leds y botones de la placa de innovación docente del ATC. Además de esto nos comunicamos con un dispositivo externo, en concreto un display LCD al cual le enviábamos cadenas de texto usando las funciones proporcionadas por la biblioteca destinada a tal efecto.

```
#include "C8051F340.h"
#include "lcdalfanum340.h"
```

```
// Peripheral specific initialization functions,
// Called from the Init_Device() function
void PCA_Init()
{
    PCA0CN    = 0x40;
    PCA0MD    &= ~0x40;
    PCA0MD    = 0x00;
    PCA0CPM0  = 0x42;
    PCA0CPH0  = 0x80;
}

void Timer_Init()
{
    TCON      = 0x10;
    TMOD      = 0x02;
    TL0       = 0x0A;
    TH0       = 0x0A;
}

void Port_IO_Init()
{
    P3MDOUT   = 0xEE;
    P0SKIP    = 0xFF;
    P1SKIP    = 0xFF;
    P2SKIP    = 0xFF;
    P3SKIP    = 0x07;
    XBR1      = 0x41;
}

void Oscillator_Init()
{
    OSCICN    = 0x83;
}

// Initialization function for device,
// Call Init_Device() from your main program
void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
}

sbit led1 = P3^1;
sbit led2 = P3^2;
sbit led3 = P3^3;
sbit bot1 = P2^5;
sbit bot2 = P2^6;
sbit bot3 = P2^7;
```

```
main()
{
    char hola[16] = {"texto1"};
    char hola2[16] = {"texto2"};
    char ind = 0;

    Init_Device();
    LCD_Init();
    LCD_DisplayString (1, 1, hola);
    LCD_DisplayString (2, 1, hola2);

    while (1)
    {
        if (bot1 == 0)
        {
            ind += 15;
            PCA0CPH0 = ind;
            while (bot1 == 0);
        }
        led2 = !bot2;
        led3 = !bot3;
    }
}
```

1.4. Conclusiones

Sí, se consiguieron los objetivos, llegando hasta el final de la práctica, manejando los leds correspondientes y consiguiendo interactuar con el display LCD. Como valoración personal la práctica al ser introductoria ha sido muy poco didáctica. Esto es debido a que el objetivo de la práctica en cuestión es familiarizarse con los elementos de trabajo con los que vamos a interactuar durante el resto de la asignatura, siendo la manipulación de leds y la comunicación con el display LCD tareas básicas dentro del marco de trabajo de la asignatura.

2. Práctica 2

2.1. Objetivos

Después de una primera práctica en la que tomamos contacto con el mundo del desarrollo con microcontroladores, usando la placa de innovación docente del departamento, que monta un microcontrolador 8051 de Silabs, es hora de conocer otros entornos de desarrollo. En esta ocasión vamos a trabajar con microcontroladores Freescale Flexis y el entorno de desarrollo CodeWarrrior.

2.2. Introducción

Para familiarizarnos con este entorno se pretende desarrollar una especie de indicador de actitud que nos indique la horizontalidad de la placa a través de leds y sonido.

Para realizar esto se dispone de una placa DEMOQE128 con un microcontrolador Flexis 8 bits (MC9S08QE128). Esta placa tiene además de un acelerómetro, 8 LEDs en una línea y un pequeño altavoz. Se pretende hacer un sistema que instalado, por eje mplo, en un avión nos indique cuando se pierde la horizontalidad, porque viremos a derecha o izquierda o porque subamos o bajemos (la parte delantera de la placa sería el morro del avión, la trasera la cola). Se trata de marcar la inclinación lateral encendiendo un solo LED, de forma que cuando la placa se encuentre horizontal sea unos de los LEDs centrales el encendido (o mejor los dos LEDs centrales), y cuando se incline a la izquierda sea uno de los LEDs de la izquierda el que se encienda, cada vez más a la izquierda, dependiendo de la inclinación (al contrario de lo que haría un nivel de burbuja). Por otra parte, la inclinación hacia delante y hacia atrás (subir o bajar el avión) se marcará por un sonido, de forma que cuando se encuentre horizontal no suene, cuando se suba genere un sonido cada vez más grave, cuanto más se incline en subida; y en bajada un sonido agudo, cada vez más, cuanto más inclinada sea la bajada. Por último, si hay turbulencias (movimientos verticales bruscos, cambios bruscos eje z del acelerómetro) o volamos invertidos (cambio eje z) debe sonar un doble tono intermitente. Aunque no es estrictamente necesario, se debe utilizar un interruptor, de los que dispone la DEMOQE, para inicializar la horizontalidad (es decir, indicarle al sistema el "cero").

2.3. Desarrollo de la práctica

Código de la práctica con la función de los LEDs realizada.

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include "MCUinit.h"

#define LED0 PTCD_PTCD0
#define LED1 PTCD_PTCD1
#define LED2 PTCD_PTCD2
#define LED3 PTCD_PTCD3
#define LED4 PTCD_PTCD4
#define LED5 PTCD_PTCD5

#define LED6 PTED_PTED6
#define LED7 PTED_PTED7

#define BTN PTDD_PTDD3
int readADC(int ch);
void ejeX(int ac);
void ledsoff();

void main(void)
{
    int i;
```

```
    int samples[] = {0, 0, 0};
    const int adch[] = {0x01, 0x08, 0x09};

    EnableInterrupts; /* enable interrupts */
    /* include your code here */
    MCU_init();

    /* loop forever */
    for (;;) {
        for (i = 0; i < 3; i++) {
            samples[i] = readADC(adch[i]);
        }

        ejeX(samples[0]);

        __RESET_WATCHDOG(); /* feeds the dog */
    }
    /* please make sure that you never leave main */
}

int readADC(int ch)
{
    int adc;
    ADCSC1 = ch;

    while (!ADCSC1_COCO);
    adc = ADCR;
    return adc;
}

void ledsoff()
{

    LED0 = 1;
    LED1 = 1;
    LED2 = 1;
    LED3 = 1;
    LED4 = 1;
    LED5 = 1;
    LED6 = 1;
    LED7 = 1;
}

void ejeX(int ac)
{
    int nled = (ac - 64) / 16;

    ledsoff();
    switch (7 - nled) {
        case 0:
            LED0 = 0;
```

```
            break;

        case 1:
            LED1 = 0;
            break;

        case 2:
            LED2 = 0;
            break;

        case 3:
            LED3 = 0;
            break;

        case 4:
            LED4 = 0;
            break;

        case 5:
            LED5 = 0;
            break;

        case 6:
            LED6 = 0;
            break;

        case 7:
            LED7 = 0;
            break;
    }
}
```

2.4. Conclusiones

Se consiguieron todos los objetivos de la práctica. Además fue una sesión interesante ya que en ella conocimos el entorno CodeWarrrior, más potente y a la vez más sencillo en cuanto a la utilización del asistente de configuración que que el empleado para el 8051 de Silabs.

3. Práctica 3

3.1. Objetivos

- Conocer como conectar un microcontrolador como periférico del PC mediante USB, obteniendo una expansión hardware fácil en los PCs.
- Conocer los fundamentos de la conexión USB, en concreto términos como LowSpeed, FullSpeed, SuperSpeed, USB 2.0, que son los descriptores y los endpoints, datos con

los que durante el desarrollo de las practicas trataremos en mas de una ocasión ya que son los aspectos mas básicos de los dispositivos USB. Con los descriptores y los endpoinds nos permiten configurar y comunicarle al equipo el tipo de dispositivo que le conectamos,

- Realizar un dispositivo USB de clase HID para usar el inclinómetro como mando para PC.

3.2. Introducción

Para comenzar la práctica, el profesor nos recuerda los diferentes tipos de dispositivos USB dependiendo de su uso, y como hay que configurarlos para que trabajen de una forma u otra para con el PC.

3.2.1. Clasificación general de los dispositivos USB

Los dispositivos USB se pueden clasificar de muy diversas formas, pero una posible clasificación atendiendo a su relación con el host sería:

- **Dispositivos de clase específica de vendedor:** Para éstos se necesita desarrollar un driver que se cargue en el sistema operativo, normalmente lo suministra el fabricante. Aunque también se pueden utilizar un driver general como el USBexpress. La principal ventaja de este tipo de dispositivos es que se pueden adaptar perfectamente a la aplicación (no tienen limitaciones de velocidad de transferencia, como algunos dispositivos de clase genérica, por ejemplo los HID)
- **Dispositivos de clase genérica:** Estos dispositivos tienen ya el driver en el sistema operativo. El fabricante no precisa desarrollar el driver, ni el usuario cargarlo. Los casos típicos son los HID, almacenamiento masivo, sonido, etc. A su vez podemos clasificar estos dispositivos de clase genérica en:
  - Los que son utilizados directamente por el SO, por ejemplo un ratón o un teclado, no precisa ningún tipo de software adicional.
  - Los que no son utilizados directamente por el SO, estos son definidos por el vendedor, en este caso aunque no precise driver, es necesario desarrollar algún tipo de aplicación para hacer uso del dispositivo USB.

La placa desarrollada sirve para probar cualquiera de los tipos de dispositivos USB comentados. Un aspecto importante para todos los dispositivos USB son los descriptores, en particular para los dispositivos de clase genérica. Por ejemplo, el ratón HID, que se suministra, se puede cambiar por un joystick, sólo hay que modificar el "report descriptor" utilizando la herramienta "HID descriptor tool".

3.2.2. Los dispositivos de clase HID (Human Interface Device)

La clase HID engloba principalmente a los dispositivos que son usados por los humanos para el control de los computadores: ratones, teclados, joysticks ... Pero también puede

englobar a otros tipos dispositivos no directamente relacionados con estos: displays, lectores de código, termómetros, voltímetros ... Tres son los aspectos a tener en cuenta con respecto a la clase HID:

1. **Qué hace que un dispositivo sea de clase HID:**

   Como en el resto de dispositivos USB los descriptores juegan un papel muy importante en los de clase HID. En el Interface descriptor el campo bInterfaceClass debe ser 03h para indicar que el dispositivo es de la clase HID. Para este tipo de dispositivos sólo hay dos subclases: 1 para dispositivos soportados por la BIOS (Boot interface subclass), 0 para el resto. El protocolo puede ser: 1 para teclado, 2 para ratones y 0 para ninguno.Existen además tres estructuras de descriptores específicos para esta clase:

   - **HID descriptor:** describe el tipo y la longitud de los descriptores subordinados (los dos siguientes).
   - **Report descriptor:** su presencia es obligatoria para los dispositivos HID, examinando este descriptor el driver del host puede determinar el tamaño y la composición de los datos mandados desde el dispositivo USB, también determina si es tipo joystick, ratón, etc. y por tanto como es visto por el sistema operativo.
   - **Physical descriptor:** este tipo de descriptores es opcional en la clase HID, informa sobre que parte o partes del cuerpo humano son utilizados para controlar el dispositivo USB. En esta práctica no lo vamos a usar ni describir.

2. **Qué datos y cómo se transfieren en los dispositivos de clase HID:**

   En los dispositivos de clase HID se transfiere unas estructuras denominadas REPORT. Un report está compuesto por una o más transacciones de datos entre el dispositivo y el host, hay tres tipos: Input, Output y Feature (configuración). Los dispositivos HID obligatoriamente utilizan el Endpoint 0 de control y además un Endpoint tipo interrupt de entrada, opcionalmente pueden usar un Endpoint tipo interrupt de salida, si éste no existiese se utiliza para transmitir información del host al dispositivo USB el Endpoint0 de control contestando a la petición Set Report.

3. **Cómo se especifican la configuración y en particular el formato de los datos en la clase HID:**

   Esto se hace mediante los descriptores, en particular en los dispositivos de clase HID el report descriptor es el encargado de indicar el formato de los datos (reports) que se van a transferir. Es importante entender que el Host lee una vez el report descriptor y no tiene porque volver a leerlo. A partir de ese momento cada vez que el Host lea (o escriba) datos se entiende que el formato es el descrito por dicho descriptor. El papel de los descriptores en general es sólo de configuración. Para describir la estructura del report descriptor comenzamos mostrando el elemento mínimo que lo compone, de forma genérica se denomina item. Los ítems tienen un byte de prefijo que en el caso general contiene bits que indican el tamaño del mismo, el tipo y la etiqueta. En el caso del item largo el valor está definido y es fijo (el que se muestra).

   Los ítems son al final una serie de números de significado para el dispositivo e interpretables por el Host pero de muy difícil lectura por el desarrollador humano.

Para manejar dichas claves por el desarrollador se asocia una especie de lenguaje de descripción, que es el que vamos a resumir en los siguientes párrafos. Afortunadamente existe una herramienta pública (Hid descriptor tool) que permite "compilar" y obtener la serie numérica del report descriptor. Los Items lo podemos dividir en tres tipos: Main, Global y local. Los ítems main a su vez son:

- **Input:** describe información que pasa del dispositivo al master.
- **Output:** describe la información que pasa del master al dispositivo.
- **Feature:** describe la información de configuración que puede ser enviada al dispositivo.
- **Collection:** agrupa varios item correspondientes a un dispositivo.
- **End Collection:** señala el fin del agrupamiento.

Realmente los tres primeros tipos de mains indican que hay una transferencia. Si entendemos que el report es una sucesión de datos lo que hacen es ir sumando los datos que componen dicha sucesión. Cómo se componen o para qué sirven dichos datos lo describen los ítems locales y globales, que siempre se encuentran precediendo a los ítems mains.

En general, en un report descriptor, los primeros item son Usage Page y Usage, ambos describen de forma general la funcionalidad de los campos siguientes. El siguiente item es el de Collection, éste agrupa datos del mismo tipo contenidos desde Collection hasta End Collection. Hay varios tipos de Collection por ejemplo: Application, Logical y Physical.

3.2.3. Acelerómetro/inclinómetro

El acelerómetro que vamos a utilizar es el ADXL311, este dispositivo se encuentra soldado en la placa PCB. El ADXL311 dispone de dos salidas analógicas que dependiendo de la aceleración proporciona una tensión que podremos medir con el conversor a/d del microcontrolador (en la PCB esas dos salidas están conectadas al conversor A/D del Cygnal). El funcionamiento teórico de los acelerómetros es relativamente simple, aunque su construcción es compleja. En este caso podríamos considerar que consiste en una masa suspendida por un "muelle de silicio" de forma que si no está sometida a ninguna aceleración hace que las placas de un condensador se encuentren alineadas, pero cuando hay una aceleración (fuerza = masa x aceleración) las placas del condensador dejan de alinearse disminuyendo la capacidad.

Esta disminución de capacidad es el parámetro eléctrico que se trata en el interior del ADXL311 para obtener la salida en tensión en función de la aceleración. Si la única aceleración a la que se somete este chip es la de la gravedad puede servir para medir la inclinación, si estuviera sometido además a otras aceleraciones el caso se complica, por tanto es conveniente mover nuestro ratón de forma que la aceleración a que lo sometemos sea despreciable frente a la gravedad.

3.3. Desarrollo de la práctica

Para el desarrollo de la practica nos centrábamos principalmente en dos actividades, adaptar los descriptores para usar nuestra placa como ratón y/o mando USB, indicándole entre otras cosas el tamaño de datos a enviar, el tipo de dispositivo(ratón, o joystick según la "altura" a la que estuviésemos de la realización de la practica y en el código principal el tratamiento del acelerómetro adaptándolo a los ejes X e Y que requiere el PC como entrada de ratón/joystick.

```
//-------------------------------------------------------------------------
// Includes
//-------------------------------------------------------------------------
#include "F3xx_USB0_Register.h"
#include "F3xx_USB0_InterruptServiceRoutine.h"
#include "F3xx_USB0_Descriptor.h"

//-------------------------------------------------------------------------
// Descriptor Declarations
//-------------------------------------------------------------------------

const device_descriptor DEVICEDESC =
{
   18,                                 // bLength
   0x01,                               // bDescriptorType
   0x1001,                             // bcdUSB
   0x00,                               // bDeviceClass
   0x00,                               // bDeviceSubClass
   0x00,                               // bDeviceProtocol
   EP0_PACKET_SIZE,                    // bMaxPacketSize0
   0xC410,                             // idVendor
   0xB986,                             // idProduct cambio
   0x0000,                             // bcdDevice
   0x01,                               // iManufacturer
   0x02,                               // iProduct
   0x00,                               // iSerialNumber
   0x01                                // bNumConfigurations
}; //end of DEVICEDESC

// From "USB Device Class Definition for Human Interface Devices (HID)".
// Section 7.1:
// "When a Get_Descriptor(Configuration) request is issued,
// it returns the Configuration descriptor, all Interface descriptors,
// all Endpoint descriptors, and the HID descriptor for each interface."
const hid_configuration_descriptor HIDCONFIGDESC =
{

{ // configuration_descriptor hid_configuration_descriptor
   0x09,                               // Length
   0x02,                               // Type
   0x2200,                             // Totallength (= 9+9+9+7)
   0x01,                               // NumInterfaces
```

```
    0x01,                               // bConfigurationValue
    0x00,                               // iConfiguration
    0x80,                               // bmAttributes
    0x20                                // MaxPower (in 2mA units)
},

{ // interface_descriptor hid_interface_descriptor
    0x09,                               // bLength
    0x04,                               // bDescriptorType
    0x00,                               // bInterfaceNumber
    0x00,                               // bAlternateSetting
    0x01,                               // bNumEndpoints
    0x03,                               // bInterfaceClass (3 = HID)
    0x01,                               // bInterfaceSubClass
    0x02,                               // bInterfaceProcotol
    0x00                                // iInterface
},

{ // class_descriptor hid_descriptor
     0x09,                                 // bLength
     0x21,                                 // bDescriptorType
     0x0101,                              // bcdHID
     0x00,                                 // bCountryCode
     0x01,                                 // bNumDescriptors
     0x22,                                // bDescriptorType
     HID_REPORT_DESCRIPTOR_SIZE_LE        // wItemLength (tot. len. of report
                                         // descriptor)
},

// IN endpoint (mandatory for HID)
{ // endpoint_descriptor hid_endpoint_in_descriptor
    0x07,                               // bLength
    0x05,                               // bDescriptorType
    0x81,                               // bEndpointAddress
    0x03,                               // bmAttributes
    EP1_PACKET_SIZE_LE,                 // MaxPacketSize (LITTLE ENDIAN)
    10                                  // bInterval
},

// OUT endpoint (optional for HID)
{ // endpoint_descriptor hid_endpoint_out_descriptor
    0x07,                               // bLength
    0x05,                               // bDescriptorType
    0x01,                               // bEndpointAddress
    0x03,                               // bmAttributes
    EP2_PACKET_SIZE_LE,                 // MaxPacketSize (LITTLE ENDIAN)
    10                                  // bInterval
}

};
```

```
const hid_report_descriptor HIDREPORTDESC =
{
    0x05, 0x01,                     // Usage Page (Generic Desktop)
    0x09, 0x04,                     // Usage (Mouse)->4
    0xA1, 0x01,                     // Collection (Application)
    0x09, 0x01,                     //   Usage (Pointer)
    0xA1, 0x00,                     //   Collection (Physical)
    0x05, 0x09,                     //     Usage Page (Buttons)
    0x19, 0x01,                     //     Usage Minimum (01)
    0x29, 0x03,                     //     Usage Maximum (01)->3
    0x15, 0x00,                     //     Logical Minimum (0)
    0x25, 0x01,                     //     Logical Maximum (1)
    0x95, 0x03,                     //     Report Count (1)->3
    0x75, 0x01,                     //     Report Size (1)
    0x81, 0x02,                     //     Input (Data, Variable, Absolute)
    0x95, 0x01,                     //     Report Count (1)
    0x75, 0x05,                     //     Report Size (7)->5
    0x81, 0x01,                     //     Input (Constant) for padding
    0x05, 0x01,                     //     Usage Page (Generic Desktop)
    0x09, 0x30,                     //     Usage (X)
    0x09, 0x31,                     //     Usage (Y)
    0x15, 0x81,                     //     Logical Minimum (-127)
    0x25, 0x7F,                     //     Logical Maximum (127)
    0x75, 0x08,                     //     Report Size (8)
    0x95, 0x02,                     //     Report Count (2)
    0x81, 0x06,                     //     Input (Data, Variable, Relative)
    0xC0,                           //   End Collection (Physical)
    0xC0                            // End Collection (Application)
};

#define STR0LEN 4

code const unsigned char String0Desc [STR0LEN] =
{
   STR0LEN, 0x03, 0x09, 0x04
}; //end of String0Desc

#define STR1LEN sizeof ("SILICON LABORATORIES") * 2

code const unsigned char String1Desc [STR1LEN] =
{
   STR1LEN, 0x03,
   'S', 0,
   'I', 0,
   'L', 0,
   'I', 0,
   'C', 0,
   'O', 0,
   'N', 0,
   ' ', 0,
   'L', 0,
```

```
   'A', 0,
   'B', 0,
   'O', 0,
   'R', 0,
   'A', 0,
   'T', 0,
   'O', 0,
   'R', 0,
   'I', 0,
   'E', 0,
   'S', 0
}; //end of String1Desc

#define STR2LEN sizeof ("C8051F320 Development Board") * 2

code const unsigned char String2Desc [STR2LEN] =
{
   STR2LEN, 0x03,
   'C', 0,
   '8', 0,
   '0', 0,
   '5', 0,
   '1', 0,
   'F', 0,
   '3', 0,
   'x', 0,
   'x', 0,
   ' ', 0,
   'D', 0,
   'e', 0,
   'v', 0,
   'e', 0,
   'l', 0,
   'o', 0,
   'p', 0,
   'm', 0,
   'e', 0,
   'n', 0,
   't', 0,
   ' ', 0,
   'B', 0,
   'o', 0,
   'a', 0,
   'r', 0,
   'd', 0
}; //end of String2Desc

unsigned char* const STRINGDESCTABLE [] =
{
   String0Desc,
   String1Desc,
```

```
   String2Desc
};

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include "c8051f3xx.h"
#include "F3xx_USB0_InterruptServiceRoutine.h"
#include "F3xx_USB0_Mouse.h"
#include "F3xx_USB0_Register.h"

////////////////////////////////////
//  Generated Initialization File  //
////////////////////////////////////

//#include "C8051F340.h"

// Peripheral specific initialization functions,
// Called from the Init_Device() function
void PCA_Init()
{
    PCA0MD    &= ~0x40;
    PCA0MD    = 0x00;
}

void Timer_Init()
{
    TMR2CN    = 0x04;
    TMR2RLH   = 0xF0;
}

void ADC_Init()
{
    AMX0P     = 0x14;
    AMX0N     = 0x1F;
    ADC0CF    = 0xFC;
    ADC0CN    = 0x82;
}

void Port_IO_Init()
{
    P1MDIN    = 0xF9;
    P3MDOUT   = 0xEE;
    P1SKIP    = 0x06;
    XBR1      = 0x40;
}

void Oscillator_Init()
{
    int i = 0;
    CLKMUL    = 0x80;
```

```
    for (i = 0; i < 20; i++);    // Wait 5us for initialization
    CLKMUL    |= 0xC0;
    while ((CLKMUL & 0x20) == 0);
    CLKSEL    = 0x03;
    OSCICN    = 0x83;
}

void Interrupts_Init()
{
    EIE1      = 0x08;
    IE        = 0x80;
}

// Initialization function for device,
// Call Init_Device() from your main program
void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    ADC_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}

void USB0_Init (void)
{

   POLL_WRITE_BYTE (POWER, 0x08);      // Force Asynchronous USB Reset
   POLL_WRITE_BYTE (IN1IE, 0x07);      // Enable Endpoint 0-1 in interrupts
   POLL_WRITE_BYTE (OUT1IE,0x07);      // Enable Endpoint 0-1 out interrupts
   POLL_WRITE_BYTE (CMIE, 0x07);       // Enable Reset, Resume, and Suspend
                                       // interrupts
   USB0XCN = 0xE0;                     // Enable transceiver; select full speed
   POLL_WRITE_BYTE (CLKREC,0x80);      // Enable clock recovery, single-step
                                       // mode disabled

   EIE1 |= 0x02;                       // Enable USB0 Interrupts

                                       // Enable USB0 by clearing the USB
   POLL_WRITE_BYTE (POWER, 0x01);      // Inhibit Bit and enable suspend
                                       // detection
}

//-----------------------------------------------------------------------------
// Main Routine
//-----------------------------------------------------------------------------
char IN_PACKET[4];

void main(void)
{
```

```
   Init_Device ();
   USB0_Init ();

   EA = 1;
   while (1)
   {
      SendPacket (0);
   }
}


void miadc(void) interrupt 10
{
    AD0INT = 0;
    IN_PACKET[1] = ADC0H;
    IN_PACKET[1] = (char) (IN_PACKET[1]-128);
}
```

3.4. Conclusiones

La práctica se consiguió realizarse en su totalidad llevando a cabo tanto el ratón como el joystick, ilustrándonos en la iniciación a la creación de alternativas USB, el manejo y funcionamiento de los descriptores, para el manejo de dispositivo usando métodos diferentes como puede ser el acelerómetro, dando ideas de aplicación a personas con diferentes necesidades.

4. Práctica 4

4.1. Objetivos

Esta práctica pretende ser una introducción a la técnica conocida como modulación por ancho de pulsos, PWM por sus siglas en inglés. Concretamente la usaremos para controlar motores de corriente continua y servos en función a las lecturas de un acelerómetro. Con las información de un eje del acelérometro controlaremos el motor y con la del otro el servo. En un hipotético coche de radiocontrol con el servo controlaríamos la dirección y con el motor la tracción.

4.2. Introducción

La técnica PWM consiste en la modulación del ancho (duración) de pulsos eléctricos y se usa principalmente para controlar la potencia suministrada a un dispositivo eléctrico o para codificar la información enviada a través de un canal de comunicaciones. Tiene multitud de aplicaciones prácticas como el control de motores eléctricos y servos o la regulación de voltaje, esta última usada en distintos tipos de fuentes de alimentación.

Un servomotor es un dispositivo capaz de orientar su eje de salida hacia una posición arbitraria dentro de su rango de actuación. Estos elementos se usan principalmente en modelismo. Están compuestos por un pequeño motor de corriente continua y un controlador de posición integrado, al que sólo hay que especificarle la posición (ángulo) en la que ha de situarse. Estos dispositivos son muy fáciles de usar, ya que el controlador interno se encarga de posicionar el servo con alta precisión, y además contiene la circuitería necesaria para proporciona la potencia necesaria al motor de su interior. Nosotros sólo hemos de transferirle la posición en que queremos situar el servo, para ello usan señales de PWM con periodos de entre 50/60Hz, codificándose la posición del servo en el tiempo en alto de la señal PWM.

Los motores de DC son dispositivos que requieren cantidades de potencia, muy elevadas en comparación con la potencia que es capaz de suministrar un microcontrolador por sus puertos de entrada/salida, así que para proporcionarles la potencia necesaria se ha de recurrir a controladores externos, como son los puentes en H. Los puentes en H se suelen componer de 4 transistores de potencia, capaz de proporcionar varios amperios (4A) a altos voltajes (12V). La idea es que estos transistores sean conmutados desde el microcontrolador, haciendo que la intensidad fluya entre los bornes del motor, además, gracias a su topología, el flujo de intensidad se puede invertir, pudiendo hacer rotar el motor en ambos sentidos. La velocidad de los motores de DC es proporcional al voltaje aplicado en sus bornes.

Para obtener velocidades intermedias se suelen usar señales de PWM con frecuencias altas para conmutar los puentes en H. De manera que el puente no está conmutado constantemente, si no que se usa una señal de PWM para "encenderlo" y "apagarlo" muy rápidamente, pareciendo desde el punto de vista del motor que la tensión aplicada a sus bornes es proporcional duty-cycle de la señal de PWM, gracias a que el motor se comporta como un filtro paso de baja.

4.3. Desarrollo de la práctica

Durante la práctica se desarrollo un pequeño programa para el microcontrolador 8051 que lee dos ejes del acelerómetro y usa estos valores para controlar la posición del servo y la velocidad del motor de corriente continua.

```
#include "C8051F340.h"

sfr16 ADC0 = 0xbd;

sbit MOT_DIR = P2^1;

unsigned int acel[2];

bit xy;

void setServoPos(unsigned int grados);
void setMotorSpeed(unsigned char velocidad, bit direccion);

void main(void)
{
```

```
    Init_Device();
    TR0 = 1;

    while (1) { }
}

void ADC_ISR(void) interrupt 10
{
    unsigned int grados;
    unsigned char velocidad;
    bit direccion;

    acel[xy] = ADC0;
    xy = !xy;

    if (xy)
    {
        AMX0P = 0x13;

    } else {
        AMX0P = 0x14;
    }

    // Servo
    grados = (acel[0] - 440) * (180 / 128);
    setServoPos(grados);

    // Motor
    if (acel[1] < 505)
    {
        velocidad = 505 - acel[1];
        direccion = 1;
    } else {
        velocidad = acel[1] - 505;
        direccion = 0;
    }

    velocidad = velocidad * 4;
    setMotorSpeed(velocidad, direccion);

    AD0INT = 0;
}

void setServoPos(unsigned int grados)
{
    unsigned int Talto;
    unsigned int Tbajo;

    Talto = 1200 + (grados * 80 / 18) * 10;
    Tbajo = 0xFFFF - Talto;
```

```
    PCA0CPL0 = Tbajo % 256;
    PCA0CPH0 = Tbajo / 256;
}

void setMotorSpeed(unsigned char velocidad, bit direccion)
{
    MOT_DIR = direccion;

    if (direccion)
    {
        PCA0CPH1 = velocidad;
    } else {
        PCA0CPH1 = 0xFF - velocidad;
    }

}
```

4.4. Conclusiones

Fue una práctica muy ilustrativa sobre los usos prácticos de modulación por anchura de pulso. Se consiguió completarla entera, no sin un poco de desconcierto por culpa del material de laboratorio (el motor DC concretamente) que estaba defectuoso y presentaba un comportamiento errático.

5. Práctica 5

5.1. Objetivos

La práctica 5 pretende introducirnos en el campo de las comunicaciones inalámbricas mediante modems XBEE, enviando y recibiendo mensajes entre nodos y mostrándolos posteriormente en un display LCD. Para ello conoceremos el método de comunicación con el que trabajan estos modems.

Como objetivo práctico se implementa un sistema de localización de interiores basado en la potencia en la que reciben los datos los diferentes nodos situados en el aula.

5.2. Introducción

Para la práctica 5 disponemos de 3 nodos fijos con los que nos podemos comunicar, dos balizas, situadas en esquinas contrarias del laboratorio que nos permitirán en la segunda parte de la práctica conocer la localización de nuestro puesto y un nodo situado en el PC del profesor, el cual dispone de una aplicación que mostrará los mensajes enviados por los dispositivos de los alumnos a ese terminal.

El modem XBEE se comunica a través del puerto serie asíncrono y tiene dos modos de funcionamiento, aunque el modo API no lo usamos.

El modo transparente consiste en enviar la información únicamente entre nodo emisor y nodo destino. Este modo, nos permite pasar al modo AT, el cual nos da la posibilidad de comunicarnos con nuestro propio dispositivo, para así poder darle al dispositivo las instrucciones correspondientes de ejecución que necesita recibir. Este cambio de modo se hace mediante el envío de tres "+" consecutivos (+++).

En nuestro caso, para comunicarnos con las dos balizas, debemos entrar en modo AT, indicarle al XBEE que nos vamos a comunicar con la primera baliza (con atdl), salir del modo AT, realizar la comunicación con la primera baliza, y realizar la misma tarea para la baliza siguiente.

Durante nuestra práctica, desarrollada durante dos sesiones, se divide en dos partes, la primera realizando al comunicación con el PC del profesor y enviarle un mensaje, y la segunda, mediante la comunicación con las balizas comentadas anteriormente, y con una tabla previamente elaborada, conocer en que posición del laboratorio nos encontramos, comparando la potencia de la señal recibida por cada una de las balizas con la tabla de valores ya muestreada permitiendo conocer dependiendo de la potencia que da cada una, la distancia a ellas y a partir de ellos, la posición.

5.3. Desarrollo de la práctica

Para el desarrollo de la práctica en primer lugar configuramos el microcontrolador para usarlo con 9600 baudios, que es el bit rate en el cual trabajan los modems.

Posteriormente creamos el código para comunicarnos con la baliza del profesor y el mensaje que enviaremos. De forma análoga se prepara la recepción del mensaje de confirmación que nos indica el PC del profesor y mostrarlos por el LCD. En la segunda parte de la práctica tratamos mediante cambios entre modo AT y modo transparente la alternancia en la comunicación con una baliza y otra, y el envío y recepción de mensajes.

```
//////////////////////////////////////
//  Generated Initialization File  //
//////////////////////////////////////

#include "C8051F340.h"

// Peripheral specific initialization functions,
// Called from the Init_Device() function
void PCA_Init()
{
    PCA0MD    &= ~0x40;
    PCA0MD    = 0x04;
    PCA0CPL4  = 0x3C;
    PCA0MD    |= 0x40;
}

void Timer_Init()
{
    TMOD      = 0x02;
    CKCON     = 0x02;
}
```

```
void UART_Init()
{
    SBRLL1     = 0x3C;
    SBRLH1     = 0xF6;
    SCON1      = 0x10;
    SBCON1     = 0x43;
}

void Port_IO_Init()
{
    P2MDOUT    = 0x01;
    P3MDOUT    = 0xEE;
    P0SKIP     = 0xFF;
    P1SKIP     = 0xFF;
    XBR1       = 0x40;
    XBR2       = 0x01;
}

void Oscillator_Init()
{
    int i = 0;
    CLKMUL     = 0x80;
    for (i = 0; i < 20; i++);     // Wait 5us for initialization
    CLKMUL     |= 0xC0;
    while ((CLKMUL & 0x20) == 0);
    CLKSEL     = 0x03;
    OSCICN     = 0x83;
}

// Initialization function for device,
// Call Init_Device() from your main program
void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    UART_Init();
    Port_IO_Init();
    Oscillator_Init();
}
```

```
#include <C8051F340.h>
#include "lcd.h"
#include "utils.h"

sbit LED1 = P3^3;
sbit LED2 = P3^2;
sbit LED3 = P3^1;


void Init_Device(void);
void sendString(char string[]);
```

```
void receiveString(char v[]);

void main(void)
{
    char v[16];
    Init_Device();
    LCD_Init();

    // Habilitar WDT
    PCA0CPH4 = 0;
    TR0 = 1;

    LED1 = 0;
    LED2 = 0;
    LED3 = 0;

    while (1) {
        // Limpiar el WDT
        PCA0CPH4 = 0;
        delayTx();
        sendString("Tere llama");
        receiveString(v);
        LCD_DisplayString (1, 1, v);
    }
}

void sendString(char string[])
{
    int i = 0;

    while (string[i] != 0) {
        SBUF1 = string[i];
        while ((SCON1 & 0x20) != 0x20);
        i++;
    }

    SBUF1 = 0x0d;
    while ((SCON1 & 0x20) != 0x20);
}

void receiveString(char v[])
{
    char i, rx_data = 0;

    while (rx_data != 0x0d) {
        while ((SCON1 & 0x01) != 0x01);
        rx_data = SBUF1;
        v[i] = rx_data;
        i++;
        SCON1 = SCON1 & (0xfe); // Limpiar RI1
    }
```

```
}
```

```
#include <C8051F340.h>
#include <math.h>
#include "lcd.h"
#include "utils.h"

sbit LED1 = P3 ^ 3;
sbit LED2 = P3 ^ 2;
sbit LED3 = P3 ^ 1;

void Init_Device(void);
void sendString(char string[]);
void receiveString(char v[]);
void enterAT(void);
int string2dec(char string[]);
int searchPosition(int db1, int db2);

void main(void)
{
    char v[16];
    char mensaje[16];
    char decibelios[16];
    int db1, db2, puesto;
    bit baliza = 0;
    Init_Device();
    LCD_Init();

    // Habilitar WDT
    PCA0CPH4 = 0;
    TR0 = 1;

    LED1 = 0;
    LED2 = 0;
    LED3 = 0;

    while (1) {
        // Limpiar el WDT
        PCA0CPH4 = 0;
        delayTx();

        delayAT();
        enterAT(); // Entrar en modo AT
        receiveString(v); // Esperar OK

        // Seleccionar baliza
        if (baliza) {
            sendString("ATDL11");
        } else {
            sendString("ATDL12");
        }
```

```
        receiveString(v); // Esperar OK

        sendString("ATCN"); // Salir del modo AT
        receiveString(v); // Esperar OK

        sendString("123456"); // Saludo a la baliza
        receiveString(mensaje); // Respuesta de la baliza

        delayAT();
        enterAT(); // Entrar en modo AT
        receiveString(v); // Esperar OK

        sendString("ATDB"); // Solicitar decibelios
        receiveString(decibelios); // Esperar decibelios

        sendString("ATCN"); // Salir del modo AT
        receiveString(v); // Esperar OK

        if (baliza) {
            db1 = string2dec(decibelios);
        } else {
            db2 = string2dec(decibelios);
            puesto = searchPosition(db1, db2);
            LCD_DisplayString(1, 1, "Puesto");
            LCD_DisplayNumber(2, 1, puesto); // Escribir puesto
        }

        baliza = !baliza;
    }
}

void sendString(char string[])
{
    int i = 0;

    while (string[i] != 0) {
        SBUF1 = string[i];
        while ((SCON1 & 0x20) != 0x20);
        i++;
    }

    SBUF1 = 0x0d;
    while ((SCON1 & 0x20) != 0x20);
}

void receiveString(char v[])i
{
    char i = 0;
    char rx_data = 0;

    while (rx_data != 0x0d) {
```

```
        while ((SCON1 & 0x01) != 0x01);
        rx_data = SBUF1;
        v[i] = rx_data;
        i++;
        SCON1 = SCON1 & (0xfe); // Limpiar RI1
    }
}

void enterAT()i
{
    int i = 0;
    for (i = 0; i < 3; i++) {
        SBUF1 = '+';
        while ((SCON1 & 0x20) != 0x20);
    }
}

int string2dec(char string[])i
{
    int n;

    if (string[0] <= '9') {
        n = string[0] - '0';
    } else {
        n = string[0] - 'A' + 10;
    }

    n = n * 16;

    if (string[1] <= '9') {
        n += string[1] - '0';
    } else {
        n += string[1] - 'A' + 10;
    }

    return n;
}

int searchPosition(int db1, int db2)i
{
    int i, puesto, min_error, tmp;
    min_error = 120;

    for (i = 0; i < 8; i++) {
        tmp = abs(db1 - balizaDB[i][0]);
        tmp += abs(db2 - balizaDB[i][1]);
        if (tmp < min_error) {
            min_error = tmp;
            puesto = i + 1;
        }
    }
```

```
    return puesto;
}
```

5.4. Conclusiones

La práctica se consiguió hacer en su totalidad y pese a una lectura del puesto en el que estábamos algo difusa, consiguió su cometido dándonos una introducción muy interesante del trabajo con comunicaciones inalámbricas y estos modems XBEE, el trabajo de comunicación interna con el propio dispositivo y con dispositivos externos y nos da una ligera idea de las diferentes aplicaciones que se le pueden dar en sistemas de comunicación con microcontroladores sin la necesidad de cableado.

6. Práctica 6

6.1. Objetivos

Esta práctica pretende introducir el trabajo con dos tipos de puertos serie desde el microcontrolador. Por un lado se va a usar el clásico puerto serie RS232, en la actualidad casi en desuso. Además se explicará como hacer uso de los puerto USB de clase específica de fabricante. Tambien realizaremos un ejercicio híbrido ya que usaremos puertos COM virtuales que emulan el viejo RS232 sobre una interfaz física USB. Para ello usaremos un firmware USB de clase HID.

6.2. Introducción

A la hora de acceder a los puertos y dispositivos del PC bajo windows se pueden utilizar diferentes mecanismos. No es objetivo de esta práctica explicar como se desarrollan drivers bajo windows, sólo se va a mostrar aquellos aspectos básicos que nos ayuden a realizar la práctica. El sistema operativo se puede considerar dividido en múltiples niveles, sin embargo, para nuestros objetivos basta considerar sólo dos: kernel y usuario. Sólo el núcleo del sistema (kernel) puede realizar las operaciones de entrada/salida, en él se debe localizar el manejador (driver) que permita a las aplicaciones utilizar los dispositivos o puertos. Las aplicación es se pueden comunicar con los drivers (modo Kernel) de diferentes maneras, pero hay una que es básica y se utiliza prácticamente en todos los sistemas operativos (incluido UNIX, norma ANSI), es mediante las funciones de la API para el acceso a ficheros: CreateFile, ReadFile y WriteFile. En esta práctica se va a utilizar dicho mecanismo. En el caso del puerto serie COM real el driver está ya en el kernel del windows, basta utilizar estas funciones como se muestra en el apéndice A. Sin embargo, el puerto USB no implementa dicho driver de forma completa, sí hay un USB, pero falta una parte en modo kernel que hay que desarrollar e instalar: el driver de clase. Para facilitar dicha labor de desarrollo en el caso del USB se va a utilizar un “wizard” que automáticamente nos va a generar el esqueleto del driver de clase, que completaremos con unas pocas líneas. En la explicación previa a la práctica se detallará cómo se prepara el driver, ahora es importante darse cuenta que nuestra aplicación para el PC puede ser casi la misma

para todos los casos, ya que se utilizan las mismas funciones para conectarse con el driver. Tanto para rs232 como para USB hay diferentes alternativas a las expuestas. Por ejemplo en el caso de rs232 se podría hacer bajo msdos y acceder directamente al puerto. En el caso del USB se puede diseñar el sistema microcontrolador como un HID (dispositivo de interface humana), como se ve en otras prácticas de la asignatura (esto se recordará al principio de la práctica). La estructura, desde el punto de vista del sistema operativo, del software de manejo de los puertos USB es la que se muestra en la siguiente figura. El USB DRIVER ya está en el sistema operativo, lo que se tiene que construir en nuestro caso es Class Device Driver, si fuese de clase genérica, como la HID, no se necesitaría construir dicho driver.

Como hemos indicado antes el driver de clase será "interpelado" desde la aplicación mediante las llamadas a la API de manejo de ficheros (CreateFile, ReadFile, etc.). Una forma simple de ver un driver es como un conjunto de subrutinas o funciones (parecido a una dll), cada vez que la aplicación "llama" al driver realmente se lanza una (o varias) subrutina de éste. Lo que realmente ocurre es que se construye una estructura denominada IRP (I/O Request Packet) que se le pasa a la función correspondiente del driver, observad la siguiente figura. El driver de clase que se va a construir tendrá que "llamar" al USB Driver y compartir de alguna forma los datos con él , esto se hace mediante una especie de IRP denominada URB (USB Request Block). En la práctica veremos que lo que tenemos que hacer es simplemente "construir" el URB con la información necesaria y "presentarlo o entregarlo" al USB driver.

6.3. Desarrollo de la práctica

A continuación se muestra el archivo con el código desarrollado para el apartado uno, la conexión sobre RS232.

```
#include "C8051F340.h"

// Peripheral specific initialization functions,
// Called from the Init_Device() function
void PCA_Init()
{
    PCA0MD    &= ~0x40;
    PCA0MD    = 0x00;
}

void Timer_Init()
{
    TCON      = 0x40;
    TMOD      = 0x20;
    TH1       = 0x30;
}

void UART_Init()
{
    SCON0     = 0x10;
}
```

```
void ADC_Init()
{
    AMX0P     = 0x14;
    AMX0N     = 0x1F;
    ADC0CF    = 0x5C;
    ADC0CN    = 0x80;
}

void Port_IO_Init()
{
    P1MDIN    = 0xFB;
    P0MDOUT   = 0x10;
    P3MDOUT   = 0xEE;
    P1SKIP    = 0x04;
    XBR0      = 0x01;
    XBR1      = 0x40;
}

void Oscillator_Init()
{
    int i = 0;
    CLKMUL    = 0x80;
    for (i = 0; i < 20; i++);    // Wait 5us for initialization
    CLKMUL    |= 0xC0;
    while ((CLKMUL & 0x20) == 0);
    CLKSEL    = 0x03;
    OSCICN    = 0x83;
}

// Initialization function for device,
// Call Init_Device() from your main program
void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    UART_Init();
    ADC_Init();
    Port_IO_Init();
    Oscillator_Init();
}

main()
{
    Init_Device();

    while(1) {
        AD0BUSY = 1;
        while(AD0BUSY);
        SBUF0 = ADC0H;
        while(!TI0);
        TI0 = 0;
```

```
    }
}
```

Y a continuación el archivo con los descriptores USB sobre el que se trabajaba en los apartados dos y tres.

```
#include "usb_main.h"
#include "usb_structs.h"
#include "usb_regs.h"
#include "usb_request.h"
#include "usb_desc.h"

//---------------------------
// Descriptor Declarations
//---------------------------
// All descriptors are contained in the global structure <gDescriptorMap>.
// This structure contains BYTE arrays for the standard device descriptor
// and all configurations. The lengths of the configuration arrays are
// defined by the number of interface and endpoint descriptors required
// for the particular configuration (these constants are named
// CFG1_IF_DSC and CFG1_EP_DSC for configuration1).
//
// The entire gDescriptorMap structure is initialized below in
// codespace.

DESCRIPTORS code gDescriptorMap = {

//---------------------------
// Begin Standard Device Descriptor (structure element stddevdsc)
//---------------------------
   18,                      // bLength
   0x01,                    // bDescriptorType
   0x00, 0x02,              // bcdUSB (lsb first)
   0x00,                    // bDeviceClass
   0x00,                    // bDeviceSubClass
   0x00,                    // bDeviceProtocol
   64,                      // bMaxPacketSize0
   0xC4, 0x10,              // idVendor (lsb first)
   0x00, 0x00,              // idProduct (lsb first)
   0x00, 0x00,              // bcdDevice (lsb first)
   0x00,                    // iManufacturer
   0x00,                    // iProduct
   0x00,                    // iSerialNumber
   0x01,                    // bNumConfigurations

//---------------------------
// Begin Configuration 1 (structure element cfg1)
//---------------------------

   // Begin Descriptor: Configuration 1
   0x09,                    // Length
   0x02,                    // Type
```

```
    0x20, 0x00,                 // TotalLength (lsb first)
    0x01,                       // NumInterfaces
    0x01,                       // bConfigurationValue
    0x00,                       // iConfiguration
    0x80,                       // bmAttributes (no remote wakeup)
    0x0F,                       // MaxPower (*2mA)

    // Begin Descriptor: Interface0, Alternate0
    0x09,                       // bLength
    0x04,                       // bDescriptorType
    0x00,                       // bInterfaceNumber
    0x00,                       // bAlternateSetting
    0x02,                       // bNumEndpoints
    0x00,                       // bInterfaceClass
    0x00,                       // bInterfaceSubClass
    0x00,                       // bInterfaceProcotol
    0x00,                       // iInterface

    // Begin Descriptor: Endpoint1, Interface0, Alternate0
    0x07,                       // bLength
    0x05,                       // bDescriptorType
    0x81,                       // bEndpointAddress (ep1, IN)
    0x03,                       // bmAttributes (Interrupt)
    0x40, 0x00,                 // wMaxPacketSize (lsb first)
    0x05,                       // bInterval

    // Begin Descriptor: Endpoint2, Interface0, Alternate0
    0x07,                       // bLength
    0x05,                       // bDescriptorType
    0x02,                       // bEndpointAddress (ep2, OUT)
    0x03,                       // bmAttributes (Interrupt)
    0x40, 0x00,                 // wMaxPacketSize (lsb first)
    0x05,                       // bInterval

//---------------------------
// End Configuration 1
//---------------------------

};
```

6.4. Conclusiones

Aunque un poco apurado debido a su longitud se consiguieron realizar los distintos apartados de la práctica con éxito.

Es una práctica muy interesante ya que se explica un tema muy importante en el mundo de los microcontroladores como es el uso de puertos para comunicarse con otros dispositivos. Pero precisamente lo "densa" que es la práctica hace la mayoría de los conceptos queden un poco cogidos con pinzas a pesar de haberse comprendido.

7. Práctica 7

7.1. Objetivos

El objetivo de esta practica es la introducción y el trabajo con los sistemas empotrados en tiempo real y el gobierno de este mediante un sistema operativo Freertos. Como medio de desarrollo para conseguir esta inicialización dispondremos de un sistema basado en ARM7 sobre una placa de desarrollo Makingthings, dándonos las ventajas de las que nos proporciona esa placa de desarrollo propias de ellas mismas que nos permite manejar de forma simple el sistema operativo en tiempo real Freertos.

Nos introduciremos también en el entorno de desarrollo para este tipo de dispositivos Crossworks, el cual mediante el sistema JLINK nos permite depurar el código introducido en la placa. Como objetivo de la práctica intentaremos conseguir comunicarnos con un display LCD y el manejo de un servo, mediante la realización de tareas.

7.2. Introducción

El sistema operativo Freertos nos permite mediante una serie de bibliotecas, y su core, interactuar en tiempo real con dispositivos y salidas de la placa de desarrollo, usando en nuestro caso la placa Makingthings, y ARM7 de Atmel. Este sistema operativo nos ofrece diferentes paquetes dependiendo de las funcionalidades que queramos obtener de base, siendo estos más o menos pesados según nuestra elección. Para nuestras practicas utilizaremos el paquete Heavy para no preocuparnos en las limitaciones ya que este paquete es el más completo del SO. Como primer ejercicio desarrollaremos mediante el uso de tareas el manejo de un display LCD, creando tareas que ejecuten dos mensajes diferentes usando la misma función, en una linea diferente del display. Al hacerlo de directa, como no se gestiona el acceso a los comandos y caracteres del display, este imprime los datos de forma errónea, solapándose unos con otros, ya que no hay ningún medio que gestione que una tarea haya acabado antes de cambiar de fila de escritura, para ellos, debemos utilizar semáforos para proteger esa escritura, habilitando así el correcto orden de escritura. Para el segundo ejercicio, utilizaremos el manejo de un servo para entrar en detalle un poco más del uso de las tareas, ya que con las tareas compartidas en tiempo real, podremos usar ese servo y dos potenciómetros para, mediante la medida de ellos, en tiempo real establecer la velocidad a la que gira el servo y con el otro potenciómetro la dirección de giro. Cabe decir, que en todo el trabajo de Tiempo real, siempre esta bien tener una tarea con la mínima prioridad que controle un parpadeo constante de uno de los LEDs de la placa, para mediante el cual, poder controlar el correcto funcionamiento de las tareas, ya que si esta tarea de menor nivel funciona correctamente, significa que las de mayor nivel también tienen tiempo para funcionar de forma apropiada.

7.3. Desarrollo de la práctica

La práctica se desarrolla en primer lugar configurando las tareas del LCD, añadiéndole posteriormente el tratamiento con semáforos para controlar la correcta ejecución y tratamiento del recurso compartido, y siempre tener en cuenta la función blinktask, que nos

permite usar el LED como depurador "visual". Para esto usamos las bibliotecas que nos proporciona el SO sobre semáforos, pudiéndose localizar fácilmente su funcionamiento en la documentación facilitada por el SO.

```
/*
make.c

make.c is the main project file.  The Run( ) task gets called on bootup, so
stick any initialization stuff in there.  In Heavy, by default we set the USB,
OSC, and Network systems active, but you don't need to if you aren't using
them.  Furthermore, only register the OSC subsystems you need - by default, we
register all of them.
*/

#include "config.h"

// include all the libraries we're using
#include "appled.h"
#include "dipswitch.h"
#include "servo.h"
#include "digitalout.h"
#include "digitalin.h"
#include "motor.h"
#include "pwmout.h"
#include "stepper.h"
#include "xbee.h"
#include "webserver.h"
#include "lcd_4bit.h"
#include "AT91SAM7X256.h"
void BlinkTask( void* parameters );
void pruebalcd(void* parametro);
void pruebalcd1( void* parameters);
void* mySemaphore;

void Run( ) // this task gets called as soon as we boot up.
{
  mySemaphore = SemaphoreCreate();
  initializeLCD();
  TaskCreate( BlinkTask, "Blink", 400, 0, 1 );
  TaskCreate(pruebalcd,"Prulcd",1000,"Tarea1+++++",5);
  TaskCreate(pruebalcd1,"Prulcd",1000,"Tarea2-----",5);
  // Do this right quick after booting up - otherwise we won't be recognised
  Usb_SetActive( 1 );

  // Fire up the OSC system and register the subsystems you want to use
  Osc_SetActive( true, true, true, true );
  // make sure OSC_SUBSYSTEM_COUNT (osc.h) is large enough to accomodate
  // them all
  Osc_RegisterSubsystem( AppLedOsc_GetName(), AppLedOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( DipSwitchOsc_GetName(), DipSwitchOsc_ReceiveMessage,
          DipSwitchOsc_Async );
  Osc_RegisterSubsystem( ServoOsc_GetName(), ServoOsc_ReceiveMessage, NULL );
```

```
  Osc_RegisterSubsystem( AnalogInOsc_GetName(), AnalogInOsc_ReceiveMessage,
          AnalogInOsc_Async );
  Osc_RegisterSubsystem( DigitalOutOsc_GetName(), DigitalOutOsc_ReceiveMessage,
          NULL );
  Osc_RegisterSubsystem( DigitalInOsc_GetName(), DigitalInOsc_ReceiveMessage,
          NULL );
  Osc_RegisterSubsystem( MotorOsc_GetName(), MotorOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( PwmOutOsc_GetName(), PwmOutOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( LedOsc_GetName(), LedOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( DebugOsc_GetName(), DebugOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( SystemOsc_GetName(), SystemOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( SerialOsc_GetName(), SerialOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( IoOsc_GetName(), IoOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( StepperOsc_GetName(), StepperOsc_ReceiveMessage,
          NULL );
  Osc_RegisterSubsystem( XBeeOsc_GetName(), XBeeOsc_ReceiveMessage,
          XBeeOsc_Async );
  Osc_RegisterSubsystem( XBeeConfigOsc_GetName(), XBeeConfigOsc_ReceiveMessage,
          NULL );

  // Starts the network up.  Will not return until a network is found...
  //Network_SetActive( true ); //no red
}

// A very simple task...a good starting point for programming experiments.
// If you do anything more exciting than blink the LED in this task, however,
// you may need to increase the stack allocated to it above.
void BlinkTask( void* p )
{
 (void)p;
  Led_SetState( 1 );
  Sleep( 1000 );

  while ( true )
  {
    Led_SetState( 0 );
    Sleep( 900 );
    Led_SetState( 1 );
    Sleep( 5 );
  }
}


//Tarea simple para escribir Hola mundo en el LCD, se podría haber hecho en la
//tarea anterior antes del while
void pruebalcd( void* p )
{
 (void)p;
 int contador=0;
  while ( true )
  {
```

```
    if(SemaphoreTake (mySemaphore,1000)){

      moveToXY(0,0);
      writeStringToLCD(p);
      writeIntegerToLCD(contador);
      SemaphoreGive(mySemaphore);
    }
    contador++;
    Sleep(700);

  }
}
void pruebalcd1( void* p )
{
 (void)p;
  int contador=0;
  while ( true )
  {
      if(SemaphoreTake (mySemaphore,1000)){
        moveToXY(1,0);
        writeStringToLCD(p);
        writeIntegerToLCD(contador);
        SemaphoreGive(mySemaphore);
    }
    contador++;
    Sleep(1000);
  }
}
```

Para el caso de la segunda parte, usando la entrada analógica para medir los potenciómetros, y actualizar el comportamiento del servo según la respuesta obtenida de esos potenciómetros, conseguimos nuestro cometido, sin olvidarnos nunca de seguir usando la función blinktask.

```
/*
make.c

make.c is the main project file.  The Run( ) task gets called on bootup, so
stick any initialization stuff in there.  In Heavy, by default we set the USB,
OSC, and Network systems active, but you don't need to if you aren't using
them.  Furthermore, only register the OSC subsystems you need - by default, we
register all of them.
*/

// include all the libraries we're using
#include "appled.h"
#include "dipswitch.h"
#include "servo.h"
#include "digitalout.h"
#include "digitalin.h"
#include "motor.h"
#include "pwmout.h"
```

```
#include "stepper.h"
#include "xbee.h"
#include "webserver.h"
#include "lcd_4bit.h"
#include "AT91SAM7X256.h"
void BlinkTask( void* parameters );
void pruebaservo(void* parametro);

void Run( ) // this task gets called as soon as we boot up.
{
  TaskCreate( BlinkTask, "Blink", 400, 0, 1 );
  TaskCreate(pruebaservo,"Pruservo",1000,0,5);

  // Do this right quick after booting up - otherwise we won't be recognised
  Usb_SetActive( 1 );

  // Fire up the OSC system and register the subsystems you want to use
  Osc_SetActive( true, true, true, true );
  // make sure OSC_SUBSYSTEM_COUNT (osc.h) is large enough to accomodate them
  // all
  Osc_RegisterSubsystem( AppLedOsc_GetName(), AppLedOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( DipSwitchOsc_GetName(), DipSwitchOsc_ReceiveMessage,
          DipSwitchOsc_Async );
  Osc_RegisterSubsystem( ServoOsc_GetName(), ServoOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( AnalogInOsc_GetName(), AnalogInOsc_ReceiveMessage,
          AnalogInOsc_Async );
  Osc_RegisterSubsystem( DigitalOutOsc_GetName(), DigitalOutOsc_ReceiveMessage,
          NULL );
  Osc_RegisterSubsystem( DigitalInOsc_GetName(), DigitalInOsc_ReceiveMessage,
          NULL );
  Osc_RegisterSubsystem( MotorOsc_GetName(), MotorOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( PwmOutOsc_GetName(), PwmOutOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( LedOsc_GetName(), LedOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( DebugOsc_GetName(), DebugOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( SystemOsc_GetName(), SystemOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( SerialOsc_GetName(), SerialOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( IoOsc_GetName(), IoOsc_ReceiveMessage, NULL );
  Osc_RegisterSubsystem( StepperOsc_GetName(), StepperOsc_ReceiveMessage,
          NULL );
  Osc_RegisterSubsystem( XBeeOsc_GetName(), XBeeOsc_ReceiveMessage,
          XBeeOsc_Async );
  Osc_RegisterSubsystem( XBeeConfigOsc_GetName(), XBeeConfigOsc_ReceiveMessage,
          NULL );

  // Starts the network up.  Will not return until a network is found...
  //Network_SetActive( true ); //no red
}

// A very simple task...a good starting point for programming experiments.
// If you do anything more exciting than blink the LED in this task, however,
// you may need to increase the stack allocated to it above.
```

```
void BlinkTask( void* p )
{
 (void)p;
  Led_SetState( 1 );
  Sleep( 1000 );

  while ( true )
  {
    Led_SetState( 0 );
    Sleep( 900 );
    Led_SetState( 1 );
    Sleep( 5 );
  }
}


//Tarea simple para escribir Hola mundo en el LCD, se podría haber hecho en la
//tarea anterior antes del while
void pruebaservo( void* p )
{
  (void)p;
  int i;
  AppLed_SetActive(0, 1);
  Sleep( 1000 );
  Servo_SetSpeed(0,10);

  while ( true )
  {
    i=AnalogIn_GetValue(0);
    AppLed_SetState( 0, 1 );
    Servo_SetPosition(0,i);
    Sleep( 500);
  }
}
```

7.4. Conclusiones

Como conclusión la práctica 7 fue muy interesante ya que nos adentraba en el mundo de los sistemas empotrados en tiempo real gestionados mediante sistemas operativos destinado para ello, lo cual no habíamos visto hasta la fecha en la titulación, y que nos deja un gran sabor de boca y con ganas de profundizar más en la cuestión en la continuación de esta asignatura. En la práctica se consiguió acabar en su totalidad y nos permitió conocer como mediante la gestión de tareas, podemos gestionar una serie de tratamiento de funciones o procedimientos con una serie de prioridades asignadas que nosotros creamos convenientes para nuestros proyectos.

8. Práctica 8

8.1. Objetivos

El objetivo de esta práctica es servir como presentación del funcionamiento de los dispositivos USB de clase audio y la generación de sonido mediante timers PWM. Se usará la placa "makingthings" para realizar un dispositivo capaz de generar comandos MIDI que que se reproduzcan en el PC y otro que reciba comandos MIDI y genere el sonido mediante un altavoz haciendo uso de señales PWM.

8.2. Introducción

Una de las clases de dispositivo más interesantes del estándar USB es la clase audio. El estándar define 3 subclases:

- **AudioControl:** Permite comunicarse con diversos controles de audio.
- **AudioStreaming:** Para la transferencia de sonido en sí.
- **MIDIStreaming:** Sirve para transferir comandos MIDI.

También proporciona un valor para identificar subclases no definidas en el estándar.

El estándar MIDI (Musical Instrument Digital Interface) es un protocolo que permite a distintos tipos de dispositivos electrónicos musicales comunicarse entre sí usando una serie de comandos que definen diversos aspectos del sonido.

Durante la práctica el PC y su "piano virtual" conectarán con la placa de Makingthings mediante USB, esto implica que el PC mandará comandos Midi para controlar la placa y configurarla de forma adecuada, esta información mandada desde el PC si no se extrae del buffer USB de la placa hará que su firmware USB acabe mandando constantemente NACK al PC, si es ta situación se prolonga durante un tiempo el PC dará por no válido nuestro sistema. Para evitar este problema, tanto para durante los dos apartados de la práctica, hay que leer mediante USB_Read lo que el PC mande, no es necesario interpretar todo lo que manda el PC, sólo tenemos que vaciar el buffer para que no se llegue a la situación descrita, en caso contrario el sistema se podría bloquear. Por ejemplo, se trata de que si llega un comando desde el PC de cambio de programa/instrumento, nuestra placa lo saca del buffer (lectura USB) y no hace nada, el PC dará por válido que se ha cambiado de instrumento, aunque en el apartado dos nuestro sistema siga sonando de la misma forma y no con el instrumento nuevo que pide el PC.

8.3. Desarrollo de la práctica

Se realizó un programa para el microcontrolador de la placa que generaba comandos MIDI con instrumentos distintos en dos situaciones, al apretar un pulsador y al detectar ciertos movimiento con el acelerómetro.

```
/*
make.c
```

```
/*
make.c is the main project file.  The Run( ) task gets called on bootup, so
stick any initialization stuff in there.  In Heavy, by default we set the USB,
OSC, and Network systems active, but you don't need to if you aren't using
them.  Furthermore, only register the OSC subsystems you need - by default, we
register all of them.
*/

#include "config.h"

// include all the libraries we're using
#include "appled.h"
#include "dipswitch.h"
#include "servo.h"
#include "digitalout.h"
#include "digitalin.h"
#include "motor.h"
#include "pwmout.h"
#include "stepper.h"
#include "xbee.h"
#include "webserver.h"

void BlinkTask( void* parameters );
void pruebamidi(void* parametro);
void Run( ) // this task gets called as soon as we boot up.
{
    TaskCreate( BlinkTask, "Blink", 400, 0, 1 );
    TaskCreate(pruebamidi, "Prumidi", 1000, 0, 5);
    // Do this right quick after booting up - otherwise we won't be recognised
    Usb_SetActive( 1 );
}

// A very simple task...a good starting point for programming experiments.
// If you do anything more exciting than blink the LED in this task, however,
// you may need to increase the stack allocated to it above.
void BlinkTask( void* p )
{
    (void)p;
    Led_SetState( 1 );
    Sleep( 1000 );

    while ( true )
    {
        Led_SetState( 0 );
        Sleep( 900 );
        Led_SetState( 1 );
        Sleep( 5 );
    }
}

// Tarea encargada de gestionar nuestro dispositivo MIDI.
```

```
void pruebamidi( void* p )
{
    (void)p;
    char buffermidisalida[8] = {0x0C, 0xC0, 0, 0, 0x09, 0x90, 0x46, 0x70};
    char buffermidisalidaoff[8] = {0x0C, 0xC0, 0, 0, 0x08, 0x80, 0x46, 0x70};
    char bufferacelsalidamaracas[8] = {0x0C, 0xC9, 0, 0, 0x09, 0x99, 70, 0x70};
    char buffermidientrada[8];
    bool keyboard_active = false;
    Led_SetState(1);
    Sleep(6000);

    while (true) {
        if (DigitalIn_GetValue(2)) {
            Usb_Write(buffermidisalida, 8);
            keyboard_active = true;
        } else if (keyboard_active) {
            Usb_Write(buffermidisalidaoff, 8);
        }

        if (AnalogIn_GetValue(4) - 512 > 10)
            Usb_Write(bufferacelsalidamaracas, 8);

        Usb_Read(buffermidientrada, 8);

        Sleep(20);
    }
}
```

8.4. Conclusiones

Finalmente por motivos de tiempo, el segundo apartado de la práctica, consistente en reproducir sonidos modulados mediante PWM en un pequeño altavoz conectado a la placa, no se llego a realizar ya que ni siquiera llegó a ser explicada. Por lo demás consideramos muy interesante la sesión, aprendimos mucho sobre los dispositivos USB de clase audio y sus distintas aplicaciones, así como sobre el estándar MIDI.