ET1032 Informática Industrial

8 de julio de 2016

La duración del examen es de 3h
Está permitido el uso de libros o apuntes y de calculadora

Pregunta 1. **(1,5 puntos)** – Indicad justificadamente los valores que se imprimirán en la función **`main`** al ejecutar el siguiente código. Suponed que los enteros de la arquitectura son de 32 bits.

```
unsigned int ej1(unsigned int data)
{
        int i, b = 2;

        for (i = 0; i < 16; i++) {
          data = data & (0xFFFFFFFF - b);
          b = b << 2;
        }
        return data;
}

main()
{
        unsigned int val[] = {0xFFFFFFFF, 0xAAAAAAAA, 0x55555555, 0x12345678};
        int i;

        for (i = 0; i < 4; i++)
          printf("Para x = 0x%08X ej1(x) devuelve 0x%08X\n", val[i], ej1(val[i]));
}
```

Para entender cómo se transforman los datos, analizaremos en primer lugar la función **`ej1`**. Su operación de transformación sobre el dato de entrada se lleva a cabo dentro del bucle **`for`**, en la sentencia

`data = data & (0xFFFFFFFF - b);`

Aquí se hace un **`and`** lógico con un valor que, en binario, es todo unos (**`0xFFFFFFFF`**) excepto los unos de la variable **`b`**, dado que se resta. Es decir, que en cada iteración se mantienen todos los unos de **`data`** menos los que están presentes en **`b`**. Como inicialmente **`b`** vale 2 –en binario **`10`**- y en cada iteración se desplaza dos veces a la izquierda, (**`b = b << 2;`**) **`data`** pierde en cada iteración, por la operación **`and`**, un **`1`** en posición impar –contando desde la derecha y empezando en 0-. En resumen, es como si se hiciera un **`and`** de **`data`** con el valor binario **`010101…010101`**. Por lo tanto, el resultado de la ejecución es

```
Para x = 0xFFFFFFFF ej1(x) devuelve 0x55555555
Para x = 0xAAAAAAAA ej1(x) devuelve 0x00000000
Para x = 0x55555555 ej1(x) devuelve 0x55555555
Para x = 0x12345678 ej1(x) devuelve 0x10145450
```

Pregunta 2. **(1,5 puntos)** – La ejecución del fragmento de código que aparece a continuación

```
struct ej2 {
        char sn[80], *pchar;
        int c1, c2, c3;
};

main()
{
        struct ej2 v1, v2, *p1, *p2;

        p2 = &v1 + 10;
        v2.pchar = p2;
        p1 = &v1;
        v1.pchar = p1;
        v1.pchar = v1.pchar + 10;

        printf("La estructura ocupa %d bytes ", sizeof(v1));
        printf("y la diferencia es %d.\n", v2.pchar - v1.pchar);
}
```

muestra por pantalla **`La estructura ocupa 96 bytes y la diferencia es ????.`** Indicad el valor que debería aparecer en lugar de los signos de interrogación, justificando la respuesta.

En este ejercicio se pone de manifiesto cómo la aritmética de punteros se adapta al tamaño de los datos, de ahí que tenga sentido especificar el tipo de datos a los que apunta cada puntero aunque los punteros en sí almacenan siempre direcciones de memoria y son, por tanto, del mismo tipo en cierto modo.

Aunque tanto **`p2`** como **`p1`** parten del mismo valor, **`&v1`**, es decir la dirección de **`v1`**, **`p2`** recibe este valor incrementado en 10. Como tanto **`p2`** como **`&v1`** son punteros a datos tipo **`struct ej2`**, el 10 se trata como 10 elementos de tamaño 96 –**`sizeof(v1)`**- así que **`p2`** almacena en realidad **`&v1+960`**. Este valor se copia posteriormente en **`v2.pchar`**. Por otra parte, **`p1`** se copia primero en **`v1.pchar`** y luego esta variable se incrementa en 10. Como **`v1.pchar`** es un puntero a **`char`**, el tamaño de los datos es 1, por lo que el valor final de **`v1.pchar`** será **`&v1+10`**. Así la diferencia entre ambos punteros es 950 y el programa muestra

`La estructura ocupa 96 bytes y la diferencia es 950.`

Pregunta 3. **(1,5 puntos)** – Indicad si las siguientes sentencias, numeradas con comentarios de 1 a 4, son correctas o no según el uso de los punteros. Justificad adecuadamente las respuestas.

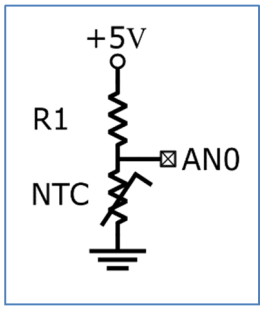
```
char *p, v[100];
int i;

p = "Hola";                          // 1
for (i = 0; i < 4; i++)
   printf("%d: %c\n", i, p[i]);      // 2
strcpy(v, p);                        // 3
strcpy(p, v);                        // 4
```

Nota: la función **`strcpy`** copia la cadena de texto a la que apunta su segundo parámetro a partir de la dirección de memoria a la que apunta el primero.

La variable **`p`** es tipo puntero a **`char`**, por lo que almacena una dirección de memoria a partir de la cual encontraremos caracteres. Por otra parte, **`v`** es un puntero constante, una dirección de memoria fija a partir de la cual hemos reservado 100 bytes para almacenar caracteres. La asignación 1 es correcta. Se guarda en **`p`** la dirección de memoria a partir de la cual el sistema ha almacenado la cadena **`"Hola"`**. En la llamada 2 se leen e imprimen uno por uno estos caracteres, también correctamente. Y en 3 se copian desde la dirección almacenada en **`p`** al espacio guardado a partir de la dirección **`v`**, también correctamente. El error se puede tener en 4. Ahora se quieren copiar datos desde la dirección **`v`** a la almacenada en **`p`**, que recordemos apunta a la zona de memoria en que el sistema ha puesto la cadena **`"Hola"`**. En la mayor parte de sistemas esta dirección corresponde a datos de solo lectura, por lo que no podemos modificarlos y la llamada 4 provocará un error de memoria.

Pregunta 4. **(3 puntos)** – Una resistencia de coeficiente de temperatura negativo NTC es un transductor que permite medir temperatura reflejando cambios de ésta en su resistividad, con una función de transferencia exponencial. Supongamos el circuito de la figura, en el que la resistencia NTC tiene una función de transferencia



$$R(t) = K_1 \cdot e^{\frac{K_2}{t}}$$

con K_1 = 57Ω y K_2 = 2000ºK –por supuesto, t se expresa también en grados Kelvin-. Se tiene además que el valor de R1 es de 47KΩ. Ambas resistencias están conectadas, como se ve en la figura, a la entrada analógica **`AN0`** de un microcontrolador **PIC18F4685**. Se pide **(1,2 puntos)** calcular las tensiones de referencia positiva $\mathbf{V_{ref+}}$ y negativa $\mathbf{V_{ref-}}$ adecuadas para medir con la mayor precisión posible temperaturas entre 0 y 100ºC. Una vez calculadas estas referencias, indicad **(1,8 puntos)** qué temperatura se estaría midiendo si del conversor AD se lee el valor de 10 bits **`0x0E7`**.

Como ayuda, la tabla inferior muestra valores de resistencia para ciertas temperaturas significativas, expresadas en grados centígrados.

Temperatura (ºC)	0	25	100
Resistencia (Ω)	86600	46839	12149

Dado que la tabla nos da los valores de resistencia para las dos temperaturas extremas que queremos medir, basta con ver qué tensiones habrá en **`AN0`** en ambos casos para tener directamente los valores de las referencias de tensión. Aplicando de forma sencilla la ley de Ohm al divisor de tensión y despreciando la corriente de entrada en **`AN0`**, se tiene

$$V_{AN0} = 5V \cdot \frac{R_{NTC}}{R_{NTC} + R_1}$$

Así pues, cuando la resistencia es de 86600Ω, correspondiente a 0ºC la tensión es de 3,24V y para 100ºC, con una resistencia de 12149Ω, de 1,03V. Con estos valores en las referencias se tiene que las medidas entre 0 y 100ºC se darían en todo el rango de resolución del conversor, pero cualquier temperatura igual o inferior a 0 grados daría una medida mínima de **`0x000`** y cualquier otra igual o superior a 100 grados daría el valor máximo, que para 10 bits es de **`0x3FF`**. Por esto y por dar unos valores más regulares a las referencias de tensión a utilizar, vamos a seleccionar los valores de 1V para $\mathbf{V_{ref-}}$ y 3,3V para $\mathbf{V_{ref+}}$.

Con estas referencias el valor de 10 bits leído en el conversor se convierte en tensión mediante una relación lineal que tiene como extremos los puntos (**`0x000`**, 1V) y (**`0x3FF`**, 3,3V). De esta manera, para obtener la temperatura correspondiente a un cierto valor leído en el conversor hay que pasar primero a voltaje según esta relación lineal, luego con la expresión anterior del divisor calcular la resistencia R_{NTC} que genera dicho voltaje y, finalmente, con la función de transferencia del transductor obtener la temperatura. Vemos ahora estos pasos y sus valores intermedios.

A partir del símbolo S leído **`0x0E7`** (231 en decimal) tenemos el voltaje según

$$V = V_{ref-} + S \cdot \frac{(V_{ref+} - V_{ref-})}{2^{10} - 1} = 1V + 231 \cdot \frac{3{,}3V - 1V}{1023} = 1{,}52V$$

Con este voltaje y despejando en la ecuación del divisor de tensión

$$R_{NTC} = R_1 \cdot \frac{V_{AN0}}{5V - V_{AN0}} = 47K\Omega \cdot \frac{1{,}52V}{5V - 1{,}52V} = 20528{,}74\Omega$$

Y por último, despejando la ecuación de transferencia de la resistencia NTC

$$t = \frac{K_2}{\ln {R_{NTC}}/{K_1}} = \frac{2000ºK}{\ln \frac{20528{,}74\Omega}{57\Omega}} = 339{,}76ºK = 66{,}76ºC$$

Pregunta 5. (**2,5 puntos**) – Un determinado dispositivo de comunicaciones almacena en memoria los datos que recibe con el siguiente formato:

- 2 bytes que indican el número de bytes de datos de usuario, de manera que el primero es el de mayor peso y el segundo el de menor. Este valor nunca puede ser 0.
- Tantos bytes de datos como indique el valor anterior.
- 2 bytes de verificación, que son la suma módulo 2^{16} de todos los bytes de datos –no se incluyen los que indican el tamaño ni estos dos-. El primer byte es el de mayor peso y el segundo el de menor.

Un ejemplo de bloque de datos válido sería:

Nº Bytes		**Datos de usuario**										**Suma**	
0	10	23	117	208	44	32	166	47	11	240	6	3	126

Como se ve, el número de datos es 10 (0 · 256 + 10), a continuación vienen los datos, cuya suma es 894, y por último en la suma se tiene este valor de la forma 3 · 256 + 126.

Realizad la función **`int verfica(unsigned char *data)`** que reciba como parámetro la dirección del bloque de datos recibido y devuelva el número de bytes de los datos de usuario o **`-1`** en caso de error al verificar su suma.

El código de la función simplemente calculará el número de datos de usuario con los dos primeros bytes, luego los irá sumando uno a uno y finalmente comparará el valor obtenido –previamente limitado a 16 bits- con el contenido en los dos últimos bytes del bloque total de datos. Veamos a continuación una forma de hacerlo.

```
int verifica(unsigned char *data)
{
  int tam, i, verif, sum = 0;

  tam = (data[0] << 8) + data[1];    // Calculamos el numero de bytes de usuario
  if (tam == 0)                      // No puede ser 0, devolvemos error
    return -1;

  for (i = 2; i < tam + 2; i++)      // Sumamos todos los bytes en un entero
    sum = sum + data[i];

  sum = sum & 0xFFFF;                // Y lo limitamos a 16 bits
  verif = (data[i] << 8) + data[i + 1]; // Leemos el valor de la suma recibido

  if (sum != verif) return -1;       // Si es distinto al calculado damos error
  return tam;                        // Si coinciden, devolvemos el numero de
}                                    // bytes de usuario
```