ET1032 Informática Industrial

7 de julio de 2017

Soluciones al examen.

NOTA: El nivel de detalle con que se responde a las preguntas en este documento no es el que se requiere ni se espera para la calificación de los ejercicios. Al redactar este documento se pretende dar información más amplia y completa con fines documentales y didácticos.

Pregunta 1. (**1,5 puntos**) – Indica justificadamente el valor de la variable **`res`** después de ejecutar el código que aparece a continuación.

```
char *res, *el = "el"
char *txt = "La frase del ejemplo y el ejercicio";

res = txt;

while (*res != '\0') {
       if (res == el) break;
       res++;
}

res = res - txt;
```

El ejercicio propuesto incide en la diferencia entre las variables tipo puntero, que almacenan una dirección de memoria, y el contenido de la memoria en dichas direcciones, lo que se suele llamar *dato al que apunta el puntero*. Para obtener fácilmente el valor de **`res`** basta con constatar que la comparación **`(res == el)`** está comparando las direcciones almacenadas en las variables, no los datos a los que apuntan. De esta manera, como **`el`** contiene la dirección de memoria en que se encuentra la cadena **`"el"`**, mientras que **`res`** recorre las direcciones en que se almacena la otra cadena, cuyo inicio se encuentra en **`txt`**, la igualdad nunca será verdadera y el bucle terminará cuando **`res`** apunte al carácter nulo final de cadena.

Luego a **`res`**, que contiene la dirección final de la cadena larga, se le resta **`txt`** que contiene la inicial, con lo que finalmente **`res`** guarda la longitud de la cadena, **35** según los datos del problema.

Pregunta 2. **(1,5 puntos)** – Indica justificadamente los cuatro valores, expresados en hexadecimal con todos sus bits, que devolvería la siguiente función cuando recibe como parámetros, respectivamente, los enteros **`0xFFFFFFFF`**, **`0x80000000`**, **`0x638DB060`** y **`0xABCDEF01`**. Se consideran enteros de 32 bits.

```
int problema(int val)
{
   int aux;

   aux = ~val;
   aux++;

   return val + aux;
}
```

La función problema que se presenta implementa calcula en sus dos líneas centrales el **complemento a 2** del parámetro que recibe. Efectivamente, la sentencia **`aux = ~val`** realiza el **complemento a 1** del valor de entrada, cambiando sus unos por ceros y viceversa, y la sentencia siguiente suma **1** a **`aux`**, con lo que se implementa la operación **`complemento a 2`**. En la aritmética que utilizan los ordenadores, aplicando esta operación sobre un entero se obtiene su valor cambiado de signo, de manera que al sumar **`val`** y **`aux`** siempre se obtiene **0**, que es el valor que por tanto devuelve siempre la función, sea cual sea el parámetro de entrada.

Veamos para constatarlo una tabla con los valores significativos que se van sucediendo para obtener este cero a partir de los valores de entrada:

val	aux = ~val	aux++	val + aux (32 bits)
0xFFFFFFFF	0x00000000	0x00000001	0x00000000
0x80000000	0x7FFFFFFF	0x80000000	0x00000000
0x638DB060	0x9C724F9F	0x9C724FA0	0x00000000
0xABCDEF01	0x543210FE	0x543210FF	0x00000000

Pregunta 3. **(2 puntos)** – Programa en lenguaje C una función **`relEnteros`** que, dado un vector de **`n`** enteros, devuelva un puntero a una matriz de **`n`**×**`n`** caracteres, cuyos elementos tengan que ver con la relación entre los enteros, tal y como se indica en la siguiente tabla. Las relaciones están ordenadas y priorizadas, es decir, si se cumple la superior no se verifica la siguiente, y así sucesivamente.

Elemento (i, j)	Relación
`'0'`	Alguno de los enteros **`v[i]`** o **`v[j]`** es **`0`**
`'='`	Ambos enteros **`v[i]`** y **`v[j]`** son iguales
`'m'`	El entero **`v[i]`** es múltiplo del entero **`v[j]`**
`'d'`	El entero **`v[i]`** es divisor del entero **`v[j]`**
`'-'`	No se da ninguna de las relaciones anteriores

El prototipo de la función pedida sería **`char *relEnteros(int *v, int n)`**. En su programación se deben utilizar las simetrías propias de las relaciones, de forma que no se realicen cálculos innecesarios. Por ejemplo, si el entero **`v[i]`** es múltiplo de **`v[j]`**, **`v[j]`** será divisor de **`v[i]`**.

La función, tras reservar memoria para la matriz de salida simplemente debe ir comparando cada elemento del vector con los siguientes, en busca de las relaciones de la lista. Esto se consigue con dos bucles anidados, de manera que el interior comienza por el elemento siguiente al externo. Dentro de este bucle interno se verifican las condiciones en el orden de la tabla, mediante **`if - else`** concatenados, y se actualiza el valor de la fila y la columna correspondiente. Dos casos merecen ser comentados: si el elemento inicial es **0** directamente se completarán su fila y columna con ceros; por otra parte, todos los elementos de la diagonal comparten fila y columna y, o bien son **0** o son iguales, por lo que se tratan antes del bucle interno. Las verificaciones de múltiplo o divisor se harán de manera sencilla con el operador módulo **`%`**. Con esto, veamos a continuación una posible implementación de esta función.

```
// Constantes para las marcas de salida

#define CERO            '0'
#define IGUAL           '='
#define MULTIPLO        'm'
#define DIVISOR         'd'
#define NADA            '-'

char *relEnteros(int *v, int n)
{
        int i, j;
        char *res;

        // Verificación del tamaño y de la memoria
        if (n < 1) return NULL;

        if ((res = malloc(n * n)) == NULL)
                return NULL;
        // El bucle externo. Se deja el último elemento para el final
        // pues no requerirá comparar con otros.
        for (i = 0; i < n - 1; i++) {
                // Si el elemento es 0 no compara y llena fila y columna
                // con ceros. Antes del for se pone el valor en la diagonal
                if (v[i] == 0) {
                        res[i * n + i] = CERO;
                        for (j = i + 1; j < n; j++) {
                                res[i * n + j] = CERO;
                                res[j * n + i] = CERO;
                        }
                }
                // Si no es 0 se pone igual en la diagonal y se va comparando con
                // todos los demás elementos siquientes
                else {
                        res[i * n + i] = IGUAL;
                        for (j = i + 1; j < n; j++) {
                                // Si el otro es 0 se pone en fila y columna
                                if (v[j] == 0) {
                                        res[i * n + j] = CERO;
                                        res[j * n + i] = CERO;
                                }
                                // Si es igual, se pone igual
                                else if (v[i] == v[j])
                                        {
                                        res[i * n + j] = IGUAL;
                                        res[j * n + i] = IGUAL;
                                }
                                // Si es múltiplo se pone en fila, y divisor
                                // en la columna
                                else if (v[i] % v[j] == 0) {
                                        res[i * n + j] = MULTIPLO;
                                        res[j * n + i] = DIVISOR;
                                }
                                // Si es divisor se pone al revés que antes
                                else if (v[j] % v[i] == 0) {
                                        res[i * n + j] = DIVISOR;
                                        res[j * n + i] = MULTIPLO;
                                }
                                // Y si no, se pone nada en fila y columna
                                else {
                                        res[i * n + j] = NADA;
                                        res[j * n + i] = NADA;
                                }
                        }
                }
        }
        // El último elemento será 0 o igual
        if (v[i] == 0)
                        res[i * n + i] = CERO;
        else
                        res[i * n + i] = IGUAL;

        return res;
```

Pregunta 4. (**2 puntos**) – Programa en lenguaje C una función **`bitTrans`** que reciba como entrada un vector de 32 enteros sin signo y devuelva un puntero a otro vector de 32 enteros sin signo, de manera que el primer elemento de este nuevo vector esté formado por los bits **0** –bits de menor peso- de todos los elementos del vector de entrada, siendo el peso de cada bit el índice del entero en el vector original. El segundo elemento estaría formado por los bits **1**, el tercero por los bits **2** y así sucesivamente. Si las ***b*** de la expresión que aparece abajo son bits; los superíndices indican el índice del elemento del vector de entrada de donde sale tal bit; y los subíndices el número de bit –comenzando por 0 a la derecha- que se obtiene de tal elemento, el elemento i^{ésimo} del vector de salida se expresaría en binario como:

$$b_i^{31} b_i^{30} b_i^{29} b_i^{28} \ldots b_i^3 b_i^2 b_i^1 b_i^0$$

El prototipo de la función pedida sería **`unsigned int *bitTrans(unsigned int *v)`**. Se considera que la arquitectura trabaja con enteros de 32 bits.

La función consistirá en dos bucles anidados. El externo fijará el bit a extraer y el índice del resultado, mientras que el interno obtendrá dicho bit para cada elemento de la entrada y lo desplazará según el índice de este para añadirlo con el operador **`or`** al resultado. Veamos una posible implementación de la función:

```
unsigned int *bitTrans(unsigned int *v)
{
        int i, j;
        unsigned int *res, val, bit;

        // Se reserva memoria
        if ((res = malloc(32 * sizeof(int))) == NULL)
                return NULL;
        // Bucle para cada bit. Inicia val a 0
        for (i = 0; i < 32; i++) {
                val = 0;
                // Bucle para cada elemento de entrada:
                // lee el bit i-ésimo del elemento j de entrada
                // y lo desplaza j posiciones para ponerlo en
                // val
                for (j = 0; j < 32; j++) {
                        bit = (v[j] >> i) & 1;
                        val = val | (bit << j);
                }
                // Se actualiza el elemento del vector de salida
                res[i] = val;
        }
        return res;
}
```

Pregunta 5. **(3 puntos)** – El conversor analógico digital **ADC**, como la mayor parte de los dispositivos del microcontrolador **PIC18F4685**, puede gestionarse mediante consulta de estado o mediante interrupciones. Sin tener en cuenta la configuración del dispositivo, pero sí los bits que permiten configurar el modo de gestionarlo –por interrupciones o consulta de estado-, indica qué bits de control y estado será necesario considerar, tanto para la configuración del modo como durante la ejecución del programa, según la gestión quiera hacerse de una u otra forma. **(0,6 puntos)**

Escribe dos fragmentos de código que realicen 8 conversiones de un mismo canal –se supone ya configurado- y las almacenen en un vector de enteros. Uno de ellos lo hará mediante consulta de estado **(1 punto)** y el otro mediante interrupciones. **(1,4 puntos)**

A continuación aparece el único registro de control y estado del conversor AD que es necesario utilizar para realizar este ejercicio. Los 10 bits del resultado, que se considera justificado a la derecha, se guardan en **`ADRESH`** y **`ADRESL`**.

Registro ADCON0							
-	-	CHS3	CHS2	CHS1	CHS0	GO/DONE	ADON
bit 7							bit 0

Para tratar el conversor analógico digital del **PIC18F4685** mediante interrupciones es obligado habilitarlas poniendo a **1** el bit **`PIE1bits.ADIE`**. Además se podría asignar prioridad a la interrupción mediante el bit **`IPR1bits.ADIP`** si **`RCONbits.IPEN`** está a **1**. Tanto en este caso como en el de gestión por consulta de estado, el **ADC** debe haber sido activado escribiendo un **1** en **`ADCON0bits.ADON`**. Los anteriores son bits de control. A la hora de determinar, dentro de la rutina de tratamiento de la interrupción, si la causa ha sido el conversor, hay que consultar el bit de estado **`PIR1bits.ADIF`** y ponerlo a **0** al terminar el tratamiento.

Si se va a tratar mediante consulta de estado basta con gestionar el bit de control y estado **`ADCON0bits.GO`** -o bit **`GO/DONE`** según aparece en la documentación-. Este bit debe ponerse a **1** para comenzar una conversión y se pone a **0** automáticamente cuando ésta ha terminado. Aunque el bit **`PIR1bits.ADIF`** también se pone a **1**, lo más eficaz es ignorarlo y usar solo **`GO/DONE`** en la consulta de estado. Por supuesto, en el caso de gestión mediante interrupciones también se debe poner a **1** este bit para comenzar una nueva conversión, sea en la rutina de tratamiento, sea en el programa principal.

Veamos a continuación un sencillo ejemplo de conversión mediante consulta de estado, en que se considera que toda la configuración del **ADC** ya se ha efectuado, y se toman 8 muestras del canal actual.

```
// Variables a usar
int conv[8], i;

// Conversión y almacenamiento de 8 muestras

for (i = 0; i < 8; i++) {                  // 8 conversiones
  ADCON0bits.GO = 1;                       // Empezamos a convertir
  while (ADCON0bits.GO == 1);              // Y esperamos que termine
  conv[i] = (ADRESH << 8) + ADRESL;        // Guardamos el valor de 10 bits
}

// Terminado
```

En el caso de usar interrupciones hay que tomar más decisiones. En el ejemplo que aparece a continuación la rutina de tratamiento solo genera el entero tras la conversión e indica que esta ya se ha llevado a cabo. El programa principal se encarga de guardar el valor y comenzar una nueva conversión. Veámoslo a continuación.

```
// Variables globales para la sincronización
volatile int adval;
volatile byte dato;

// Programa principal
main()
{
   int conv[8], idx;

   // Iniciamos las variables a 0
   dato = 0;
   idx = 0;
   // Configuración de ADC (solo lo relativo al ejercicio)
   PIR1bits.ADIF = 0;         // Limpiamos el flag
   PIE1bits.ADIE = 1;         // y habilitamos la interrupcion
   ADCON0bits.ADON = 1;       // Lo activamos
   ADCON0bits.GO = 1;         // E iniciamos la primera conversión

   while(1) {
      while (dato == 0);      // Esperamos que termine la conversion
      conv[idx++] = adval;    // Guardamos el valor generado en la ISR
      if (idx == 8) break;    // Si hemos tomado 8 muestras salimos
      ADCON0bits.GO = 1;      // Empezamos una nueva conversion
      dato = 0;               // Marcamos que no hay nuevo dato.
   }
   // Posiblemente el programa haga algo más
   // …
}

// Rutina de tratamiento
void highISR()
{
  // Gestión del ADC. Componemos el valor y lo ponemos en
  // la variable adval. Luego ponemos dato a 1.
  if (PIR1bits.ADIF == 1) {   // Verificamos el flag
      adval = (ADRESH << 8) + ADRESL;
      PIR1bits.ADIF = 0;      // Se pone el flag a 0
      dato = 1;
  }
}
```