ET1032 Informática Industrial

2 de junio de 2016

Soluciones de las preguntas

Pregunta 1. **(1 punto)** – Los microcontroladores de la familia **PIC18** de **Microchip**, que siguen la arquitectura *Harvard*, disponen de una gran cantidad de memoria **FLASH** para los programas y una memoria **RAM** de datos mucho menor. Indicad por qué en este caso es un problema trabajar con grandes tablas de valores constantes y es necesario disponer de algún mecanismo especial que permita acceder a dichas tablas almacenadas en la memoria **FLASH** de código.

La arquitectura *Harvard* mantiene dos memorias separadas, una para datos, en la que el procesador puede leer y escribir normalmente, dirigido por las instrucciones que ejecuta, y otra para instrucciones a la que el procesador solo accede, para leerlas, en la primera fase de su ejecución. Aplicando estrictamente la definición de aquella arquitectura es pues imposible que una instrucción acceda, como operandos fuente, a datos almacenados en la memoria de programa. Sin embargo, en los microcontroladores, usados comúnmente en sistemas empotrados que contienen todos los programas –instrucciones y datos constantes- en su **FLASH**, es necesario poder acceder a tablas, textos y otras constantes almacenados junto con el código. Así pues, si la arquitectura del microcontrolador es *Harvard*, se debe disponer de algún mecanismo en la arquitectura o mediante el diseño del hardware para permitir el acceso a estos datos como tales.

Pregunta 2. **(1 punto)** – Indicad justificadamente el valor de la variable **`res`** después de ejecutar el fragmento de código que aparece a continuación. Se supone que los enteros de la arquitectura son de 32 bits.

```
int ia, ib, res = 0;
unsigned int ua, ub;

ia = 0x50328240;
ua = ia;
ib = ia + ia;
ub = ua + ua;
if (ia > ib) res = res + 1;
res = res << 1;
if (ua > ub) res = res + 1;
```

El valor de la constante **`0x50328240`** en decimal es **`1345487424`**, valor positivo si se considera tanto en binario natural –correspondiente a un entero sin signo- como en complemento a 2 –el formato de representación de los enteros con signo-. Si la sumamos consigo misma, el resultado es **`0xA0650480`**. En este caso, y dado que su bit más significativo es 1, sí tenemos diferencias según el sistema de representación. Si consideramos el valor en binario natural, en decimal sería **`2690974848`**, mientras que en complemento a 2 es **`-1603992448`**, negativo.

Así pues, tratando con enteros con signo **`int`**, tenemos que el valor original es mayor que la suma consigo mismo, cosa que no ocurre si se trata de enteros sin signo **`unsigned int`**. Así pues la condición **`if (ia > ib)`** se cumple y **`res`** pasa a valer 1. Posteriormente se desplaza a la derecha una posición, con lo que vale 2. Como la condición **`if (ua > ub)`** no se cumple, res no se modifica más y el resultado final es **`res == 2`**.

Pregunta 3. **(1,5 puntos)** – Indicad justificadamente el valor de los enteros **`a.cont`** y **`b.cont`** después ejecutar el fragmento de código que aparece a continuación.

```
struct pr3 {
   double dx, dy, t;
   int cont;
}
…
struct pr3 a, b, *p2, *v1[MAX], v2[MAX];
int i;

a.cont = 0;
b.cont = 0;
for (i = 0; i < MAX; i++) {
   v1[i] = &a;
   v2[i] = b;
}
p2 = v2;
for (i = 0; i < MAX; i++) {
   v1[i]->cont = v1[i]->cont + 1;
   p2->cont = p2->cont + 1;
   p2++;
}
```

Este ejercicio pone de manifiesto las diferencias entre realizar copias de los datos o trabajar con distintas referencias –punteros- a los mismos datos, algo que puede causar confusión trabajando con estructuras en C o con clases en algunos lenguajes orientados a objetos. El valor de la variable **`b`** se copia en cada uno de los elementos del vector **`v2`**, pero tanto aquélla como los elementos son variables independientes que ocupan, cada una, su propio espacio de almacenamiento – de tamaño **`sizeof(struct pr3)`**-. En cambio, en todos los elementos del vector de punteros **`v1`** se copia la dirección única de la variable **`a`**, por lo que tenemos muchos punteros que apuntan a un mismo dato. Es fácil ver entonces que cuando se modifica el valor de los campos **`cont`** de los elementos del vector **`v2`**, se modifican distintas variables que a su vez son distintas de **`b`**, que no se ve modificada. Por eso al final **`b.cont`** sigue valiendo 0. Por otra parte, cuando se modifica la variable a la que apunta cada uno de los punteros de **`v1`**, siempre se está modificando **`a`**, que termina incrementándose **`MAX`** veces y por lo tanto **`a.cont`** es igual al final a **`MAX`**. Veámoslo ahora analizando el código con más detalle.

Inicialmente se define la estructura y se declaran las variables de este tipo: **`a`** y **`b`**, el vector de variables **`v2`** y el de punteros **`v1`**, y un puntero auxiliar **`p2`** para recorrer el vector **`v2`**. Tras iniciar los campos **`cont`** de **`a`** y **`b`** a 0, el primer bucle rellena el vector **`v1`** repitiendo en cada elemento la dirección donde reside la variable **`a`**, y el vector **`v2`** poniendo en cada uno de sus elementos una copia –esto es lo importante- de la variable **`b`**. Luego se hace que **`p2`** apunte al inicio de **`v2`** y se entra en el bucle que modifica valores. Para cada elemento de **`v1`** se accede al campo **`cont`** de la variable a la que apunta, que como hemos dicho es siempre **`a`**, y se incrementa, por lo que la misma variable es incrementada **`MAX`** veces. Por otra parte, **`p2`** apunta a un elemento de **`v2`**, que es una variable tipo **`struct p3`**, y cuyo campo **`cont`** se incrementa. Para terminar **`p2`** se incrementa para apuntar al siguiente elemento de **`v2`**. Como los elementos de **`v2`** son valores independientes, y en ningún caso se modifica la variable **`b`**, el valor de **`b.cont`** se mantiene a 0, como se ha dicho antes. Por otra parte, como todos los punteros de **`v1`** hacen referencia a la variable **`a`**, esta se modifica **`MAX`** veces sumándole 1 y **`a.cont`** termina con el valor **`MAX`**.

Pregunta 4. **(1,5 puntos)** – Para la ejecución de una aplicación en el microcontrolador **PIC18F4685**, se ha configurado el registro de control del temporizador cero mediante **`T0CON = 0x83`**, seleccionando un *prescaler* de 16. En un momento de la ejecución se lee del registro **`TMR0`** el valor **`0xC648`** y en otro posterior **`0x016B`**. Indicad el tiempo transcurrido entre ambas lecturas en el caso **a)** de que haya ocurrido un desbordamiento de **`TMR0`** entre ambas lecturas y en el caso **b)** en que hayan ocurrido 4 desbordamientos. Se supone que la frecuencia del microcontrolador es de 40 MHz y, por tanto, la de instrucciones, base de tiempos del temporizador, de 10 MHz.

Registro T0CON							
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

Verificando el valor de **`T0CON`** constatamos que es el esperado: mirando los bits de mayor a menor peso, vemos que está activado y en modo de 16 bits; que el reloj es el reloj interno de ejecución de instrucciones e, ignorando **`T0SE`** cuyo valor es irrelevante, el *prescaler* está asignado al temporizador y configurado con un valor de 3, lo que corresponde a dividir la frecuencia entre 16. Con esta división y dado que el reloj de instrucciones es de 10MHz, el tiempo de ciclo seleccionado es de 1,6 µs –el ciclo del reloj de instrucciones es de 0,1 µs-, así pues, dada cierta diferencia **`x`** entre dos valores de **`TMR0`** en diferentes instantes, el tiempo transcurrido entre ambos será de **`x`** · 1,6 µs –si no ha ocurrido ningún desbordamiento no detectable del registro temporizador entre ellos-.

En el caso que se plantea, entre los valores inicial **`0xC648`** y final **`0x016B`** en que necesariamente tiene que haber ocurrido un desbordamiento de **`TMR0`**, la diferencia en tics –en valor- es de **`0x3B23`**, en decimal **`15139`**, por lo que el tiempo transcurrido es de **`15139`** · 1,6 µs, es decir 24222,4 µs o 24,2224 ms. A la hora de calcular la diferencia se resta directamente. El hecho de que el sustraendo sea menor que el minuendo implicaría que *llevamos uno* en la última cifra, lo que es irrelevante cuando el tamaño de los operandos está limitado a un valor cíclico de 16 bits.

En el caso de que hayan ocurrido 4 desbordamientos, es decir, tres desbordamientos adicionales además del natural que se deriva de que el segundo valor sea menor que el primero, simplemente hay que contar tres veces el tiempo que tarda el contador en desbordarse, es decir en pasar de un cierto valor al mismo, tras recorrer todos los valores posibles entre ellos, equivalente a la duración de **`0x10000`** tics. Esta es **`65536`** · 1,6 µs, o sea 104,8576 ms. Concluyendo, si se ha desbordado tres veces adicionales además de la diferencia calculada antes, el tiempo total es de 3 · 104,8576 ms + 24,2224 ms = 338,7952 ms.

Pregunta 5. **(2 puntos)** – Se desea realizar un sistema con un microcontrolador **PIC18F4685** que, mediante 2 interruptores y 3 bombillas de 12V y 5W, en todo momento mantenga encendidas de 0 a 3 de las luces, según el valor binario configurado mediante los interruptores. Para realizar el circuito se dispone de transistores **NPN** con una ganancia $\mathbf{h_{FE}}$ de 140 y una intensidad de colector máxima de 500 mA, además de los interruptores y cualquier tipo de resistencia. Se pide dar un esquema del circuito a montar, justificando los valores de todas las resistencias utilizadas **(1 punto)**, así como el programa, que incluirá la configuración del microcontrolador de acuerdo con los pines mostrados en el esquema **(1 punto)**. Para el diseño de la electrónica se supone para los transistores que $\mathbf{V_{CE}}$ es despreciable y $\mathbf{V_{BE}}$ es de 650 mV. Para el microcontrolador, alimentado a 5 V, se supondrán los valores eléctricos de la tabla adjunta.

V_{IHmin}	3,6V	V_{OHmin}	4,6V	I_{Imax}	4 µA
V_{ILmax}	1,2V	V_{OLmax}	0,6V	I_{Omax}	35 mA

El diseño de la electrónica es sencillo. Los pulsadores tendrán resistencias de *pull-up* de 56K, que se conectarán al interruptor y al pin de entrada de la forma habitual. El otro extremo del pulsador irá a tierra. De esta manera cuando el pulsador esté oprimido la corriente que lo atraviesa será menor de 100 µA, con un consumo inferior a 0,5 mW.

Para encender las bombillas se usarán los transistores provistos. La intensidad de colector se calcula fácilmente dividiendo la potencia de la bombilla entre la tensión de alimentación, teniendo

$$I_C = \frac{5\ W}{12\ V} = 417\ mA.$$

Ahora utilizamos la ganancia del transistor, 140, para calcular la intensidad de base de saturación mínima, y elegimos una intensidad de base muy superior para garantizar la saturación:

$$I_{BSAT} = \frac{I_C}{h_{FE}} = \frac{417\ mA}{140} = 2{,}98\ mA.\ Elegimos\ I_B = 10 \cdot I_{BSAT} \cong 25\ mA.$$

Para que circule esta intensidad de base como mínimo, calculamos la resistencia de base según

$$R_B = \frac{V_{OHmin} - V_{BE}}{I_B} = \frac{4{,}6\ V - 0{,}650\ V}{25\ mA} = 158\ \Omega.$$

Seleccionamos una resistencia estándar de 150 Ω, y entonces tenemos que en el mejor y peor caso las corrientes que circulan será de

$$I_{Bmin} = \frac{V_{OHmin} - V_{BE}}{150\ \Omega} = 26{,}3\ mA;\ I_{Bmax} = \frac{V_{CC} - V_{BE}}{150\ \Omega} = 29\ mA.$$

Donde podemos comprobar que no se supera la corriente máxima de salida de los pines.

Una vez diseñada la electrónica decidimos que los interruptores se conectarán en los pines **RC0** y **RC1** y las bases de los transistores que activan las bombillas en **RB0 – RB2**, respetando el peso de los bits correspondientes. Con esta configuración podemos proceder ya a hacer el código. De las muchas formas posibles de hacerlos –mediante decisiones condicionales, mediante una sentencia de selección múltiple, etcétera- se ha escogido utilizar una tabla que relacione los valores leídos de los interruptores en el puerto **C** con los valores que activan las bombillas escritos en el puerto **B**, quedando un bucle muy compacto. A continuación se incluye el programa pedido.

```
// Tabla para encender las bombillas.
// Tiene en cuenta que un pulsador no pulsado se lee como 1
unsigned char bombVal[4] = {0x07, 0x03, 0x01, 0x00};

main()
{
   // Solo hay que configurar los 3 pines bajos de PORTB
   // como salidas -con su tris a 0-.
   TRISB = 0xF8;

   // Usamos el valor de los bits de PORTC
   // para escribir en B.
   // Preservamos los bits altos de PORTB aunque en este
   // caso particular no haría falta pues no se usan
   while(1)
      LATB = (LATB & 0xF8) | bombVal[PORTC & 0x03];
}
```

Pregunta 6. (3 puntos) – Cierto sistema envía cada 5 segundos datos de localización que se almacenan en una estructura como la que aparece más abajo, con la latitud y la longitud en grados y la altitud en cm sobre el nivel del mar. Un recorrido determinado se almacena en un vector cuyos elementos, ordenados según el instante de recepción, siguen dicha estructura de datos. Se pide realizar dos funciones para obtener cierta información acerca de un recorrido. Ambas recibirán como parámetros el vector junto con su tamaño y dos instantes temporales en segundos desde el origen de las medidas –no índices del vector- entre los que obtener los datos pedidos –ambos extremos se incluyen-. La primera **(1 punto)** devolverá los desniveles positivo y negativo máximos con respecto a la altura del instante inicial. La segunda **(2 puntos)** devolverá el desnivel acumulado positivo y negativo entre los instantes seleccionados.

NOTA: el desnivel acumulado positivo es la suma de todas las diferencias de altura recorridas en subidas durante el recorrido; el negativo es lo mismo, pero durante las bajadas.

A continuación aparece la estructura de datos y los prototipos de las funciones pedidas. Ambas funciones devuelven el valor positivo y escriben el negativo en la dirección pasada en la variable **neg**.

```
struct data {
   double lat, lon;
   int alt;
};

int maxDesnivel(struct data *recorrido, int tam, int tini, int tfin, int *neg);
int acumulados(struct data *recorrido, int tam, int tini, int tfin, int *neg);
```

```
int maxDesnivel(struct data *recorrido, int tam, int tini, int tfin, int *neg)
{
        int iIni, iFin, i, max, min, h0, des;
        // Primero pasamos de tiempo a índices, dividiendo entre
        // el periodo -5 segundos-.
        // Además ajustamos los índices para que se queden dentro del vector
        iIni = tini / 5;
        if (iIni < 0) iIni = 0;
        else if (iIni > tam - 1) iIni = tam - 1;
        iFin = tfin / 5;
        if (iFin < 0) iFin = 0;
        else if (iFin > tam - 1) iFin = tam - 1;
        // Ahora ajustamos los índices entre ellos, garantizando
        // que el menor sea el primero. Si son iguales, no damos
        // error, sino que devolvemos todo a 0 lo que es coherente
        if (iIni > iFin) {
                i = iFin;
                iFin = iIni;
                iIni = i;
        }
        if (iIni == iFin) {
                *neg = 0;
                return 0;
        }
        // En el algoritmo en sí, tomamos la altura del instante inicial
        // y ponemos los extremos a 0
        h0 = recorrido[iIni].alt;
        max = 0;
        min = 0;
        // Ahora para cada elemento calculamos el desplazamiento y
        // actualizamos el máximo y mínimo según sea necesario
        for (i = iIni + 1; i <= iFin; i++) {
                des = recorrido[i].alt - h0;
                if (des > max) max = des;
                else if (des < min) min = des;
        }
        // Ponemos el mínimo en su lugar y devolvemos el máximo
        *neg = min;
        return max;
}
```

```
int acumulados(struct data *recorrido, int tam, int tini, int tfin, int *neg)
{
        int iIni, iFin, i, aPos, aNeg, hAnt, des;
        // El inicio es como antes.
        iIni = tini / 5;
        if (iIni < 0) iIni = 0;
        else if (iIni > tam - 1) iIni = tam - 1;

        iFin = tfin / 5;
        if (iFin < 0) iFin = 0;
        else if (iFin > tam - 1) iFin = tam - 1;

        if (iIni > iFin) {
                i = iFin;
                iFin = iIni;
                iIni = i;
        }
        if (iIni == iFin) {
                *neg = 0;
                return 0;
        }
        // Ponemos la altura anterior a la inicial y los
        // acumulados a 0
        hAnt = recorrido[iIni].alt;
        aPos = 0;
        aNeg = 0;
        // Para cada elemento, calculamos el desnivel con respecto
        // al anterior, y actualizamos el anterior. Si el desnivel
        // es positivo se acumula a subida, si es negativo o cero
        // a bajada. Sumamos 0 pero ahorramos una comparación, así
        // que es correcto y no es más costoso
        for (i = iIni + 1; i <= iFin; i++) {
                des = recorrido[i].alt - hAnt;
                hAnt = recorrido[i].alt;
                if (des > 0) aPos = aPos + des;
                else aNeg = aNeg + des;
        }
        // Ponemos el negativo en su lugar y devolvemos el positivo
        *neg = aNeg;
        return aPos;
}
```