ET1032 Informática Industrial

1 de junio 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 punto) – Indica justificadamente cuál es la mayor precisión que se puede alcanzar midiendo tiempos con el temporizador **`TMR0`** de un microcontrolador **PIC18F4685** funcionando con un reloj de 4 MHz, sin usar otra señal de reloj externa. Indica así mismo el mayor lapso de tiempo que se podría medir sin realizar extensión del temporizador por software. La configuración del temporizador no tiene por qué ser la misma en ambos casos.

Con un reloj de 4 MHz la base de tiempos para el temporizador **`TMR0`**, si no se utiliza una señal externa conectado al pin correspondiente, es el reloj de instrucciones, cuya frecuencia sería una cuarta parte del reloj del sistema, es decir de 1 MHz. Esta frecuencia se puede a su vez dividir, mediante el *prescaler* de **`TMR0`**, entre las distintas potencias de 2 hasta 256 como máximo. De esta forma, sin aplicar *prescaler* el tiempo de ciclo del temporizador, **mínimo**, sería de **1 µs**, que nos da una medida de la precisión máxima con que se pueden medir tiempos.

En el otro extremo, configurando el *prescaler* para dividir la frecuencia base entre 256 se tiene un tiempo de ciclo, máximo, de 256 µs, lo que nos da un tiempo de desbordamiento del temporizador de 2^n x 256 µs, es decir, dado que **`n`** son los 16 bits de **`TMR0`**, 65536 x 256 µs = 16,777216 s.

En realidad el **tiempo máximo medible** sería de un ciclo menos, **16,776960 s**, dado que cuando el temporizador se desborda tras cambiar sus 16 bits se llega a un valor igual al inicial, con lo que es imposible distinguir sin utilizar variables gestionadas por programa, si se ha desbordado o no ha transcurrido ni un ciclo.

Pregunta 2. (1 punto) – Indica justificadamente si el valor de la variable **`res`** es el mismo después de ejecutar el bloque de código 1 que después del bloque 2. Se supone que los enteros de la arquitectura son de 32 bits.

```
int res, *pint;

// Bloque 1
res = pint;
res = res + 8;

// Bloque 2
res = pint + 8;
```

Es evidente que el valor de **`res`** es distinto después de cada una de las dos ejecuciones, siendo tras el primer bloque el valor (la dirección de memoria) almacenado en **`pint`** más **8** y tras el segundo el mismo valor almacenado en **`pint`** más **32**.

Veamos por qué es así: en el bloque 1 se asigna a la variable **`res`** el contenido de la variable **`pint`**, realizándose una conversión de tipos de puntero a entero (que posiblemente genere un *warning* al compilar el código). Tras esta asignación se suma a **`res`**, una variable de tipo entero, el valor constante **8**, con lo que el resultado para **`res`** al final del bloque 1 es el valor de **`pint`** + **8**.

En el segundo bloque se asigna a **`res`** el resultado de sumar **8** a **`pint`**. Como su tipo es puntero a entero, el compilador usa este tipo en la suma, y multiplica la constante **8** por el tamaño de los enteros, que es cuatro. Al asignar a **`res`** se realiza un cambio de tipos (y se genera previsiblemente un *warning*) pero el valor a almacenar en **`res`** es ya **`pint`** + **32**.

Pregunta 3. **(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`**, **`0xA5A55A5A`**, **`0x03C00960`** y **`0x12345678`**.

```
unsigned char problema(unsigned int val)
{
   unsigned char r1 = 0, r2 = 0;

   while (val) {
        if (val & 1) r1 = r1 + 1;
        if (val & 2) r2 = r2 + 1;
        val = val >> 2;
   }

   return (r2 << 4) + r1;
}
```

Como se puede ver por los tipos de datos sin signo y por las operaciones lógicas y de desplazamiento, la función **`problema()`** realiza ciertos cálculos bit a bit a partir de la variable de entrada **`val`**. En las dos sentencias **`if`** se incrementan sendos contadores, **`r1`** y **`r2`**, según el resultado de la operación *and* con **1** (**01** en binario) y **2** (**10** en binario). Posteriormente **`val`** se desplaza dos bits hacia la derecha en cada iteración, saliendo del bucle **`while`** cuando su valor es 0, es decir, no quedan bits a **1**.

De esta manera es fácil observar cómo de cada dos bits de **`val`**, se incrementa **`r1`** si el de menos peso vale **1** y **`r2`** si el de mayor peso vale **1**. Si empezamos la cuenta de bits por **0** (el de la derecha) podremos concluir que **`r1`** cuenta el número de unos en bits pares y **`r2`** el de unos en bits impares de **`val`**. Finalmente ambos contadores se combinan en un entero, desplazando **`r2`** a la izquierda **4** posiciones (es decir, multiplicándolo por **16**) y sumando **`r1`**. De esta manera se deja **`r1`** en el medio byte más bajo y **`r2`** en el alto. Como los enteros son de 32 bits, el valor máximo que puede tomar cualquiera de las variables es 16, así que solo se pierde información al combinarlos si hay 16 unos en posición par o impar, como en el primer caso.

Una vez analizada la función se puede calcular fácilmente la tabla de resultados que aparece a continuación, para los valores pedidos:

val	Valor devuelto
`0xFFFFFFFF`	**`0x10`**
`0xA5A55A5A`	**`0x88`**
`0x03C00960`	**`0x44`**
`0x12345678`	**`0x58`**

Pregunta 4. **(1,5 puntos)** – Indica justificadamente todos los valores posibles, en hexadecimal, de los enteros **`v[2]`**, **`v[5]`** y **`v[7]`** después de la ejecución del código que aparece a continuación. Supón que inicialmente todas las posiciones del vector tienen el valor 0 y que los enteros son de 32 bits.

```
int v[10], i;
char *pc;

pc = (char *)v;
for (i = 0; i < 10; i++) {
   pc[0] = i;
   pc[3] = 10 - i;
   pc = pc + 4;
}
```

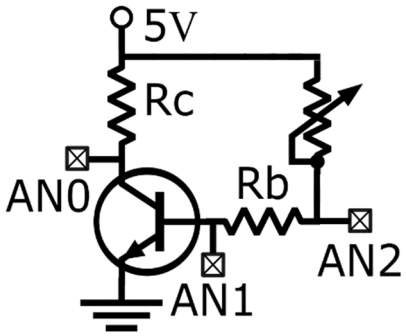
El fragmento de código que se muestra modifica de forma selectiva ciertos bytes de los **10** enteros de un vector. Inicialmente se crea un puntero a **`char`**, **`pc`**, para que apunte a la dirección de inicio del vector. Dentro del bucle **`for`** se modifican dos bytes, el más alto y el más bajo en memoria dentro de cada entero, y posteriormente se suma **`4`** al puntero **`pc`** para pasar al siguiente entero –recordemos que son de 32 bits-.

Para saber el valor resultante de los enteros hay que suponer, como se indica, que inicialmente todos valen **0**. El programa pone el valor de **`i`** en el byte más bajo en memoria, y el de **`10 - i`** en el más alto, para cada entero. El valor final de los enteros dependerá de la organización de la memoria. Si es *little endian* el valor más bajo en memoria será el de menor peso, mientras que el más alto será el de mayor peso. En caso de tratarse de una organización *big endian* ocurrirá al revés. La tabla siguiente muestra los valores de todos los enteros tras la ejecución del código en ambos casos, resaltando los que se piden en el enunciado.

i	Little endian	Big endian
0	0x0A000000	0x0000000A
1	0x09000001	0x01000009
2	**0x08000002**	**0x02000008**
3	0x07000003	0x03000007
4	0x06000004	0x04000006
5	**0x05000005**	**0x05000005**
6	0x04000006	0x06000004
7	**0x03000007**	**0x07000003**
8	0x02000008	0x08000002
9	0x01000009	0x09000001

Pregunta 5. **(2,5 puntos)** – Se desea utilizar un microcontrolador **PIC18F4685** para medir la ganancia de corriente **h_{FE}** de un transistor bipolar **NPN** según el circuito de la figura. Además del conversor analógico/digital utilizando los canales que se indican, se dispone de un pulsador que se oprime cada vez que se desea calcular la ganancia, conectado a uno de los pines digitales.

Se supone que el sistema ya está configurado adecuadamente, con el conversor activo, los canales analógicos configurados como tales, los tiempos de conversión adecuados –el tiempo de adquisición no es manual- y el resultado justificado a la derecha. Realiza el fragmento de código –un bucle infinito- que se encarga de esperar que se pulse y luego realizar las conversiones y los cálculos adecuados para obtener la ganancia como un valor entero. Se debe tratar adecuadamente el pulsador, esperando que se libere antes de detectar otra pulsación y realizar un nuevo ciclo de medida y cálculo. Los valores de las resistencias **Rc** y **Rb** son constantes definidas en el programa.



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 se guardan en **`ADRESH`** y **`ADRESL`**.

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

En primer lugar vamos a estudiar cómo realizar el cálculo de la ganancia pedida, h_{FE}. Por definición la ganancia de un transistor bipolar es la relación entre la corriente colector-emisor y la corriente base-emisor. Cuando el transistor está en la zona de conducción esta ganancia es constante y es un parámetro propio del transistor. Cuando está saturado esta relación no se cumple ya que la corriente de colector es la corriente máxima que puede circular a través de la unión colector-emisor, según la recta de carga –al final, según la ley de Ohm-.

Según esto y viendo el montaje de la figura, es fácil calcular las corrientes necesarias usando los valores medidos

$$I_c = \frac{5V - V_{AN0}}{R_c} \quad \text{e} \quad I_b = \frac{V_{AN2} - V_{AN1}}{R_b}$$

y de esta forma

$$h_{FE} = \frac{I_c}{I_b} = \frac{R_b \cdot (5V - V_{AN0})}{R_c \cdot (V_{AN2} - V_{AN1})} = \frac{R_b \cdot (1023 - AN0)}{R_c \cdot (AN2 - AN1)}$$

Se puede observar, en la última expresión, que no es necesario pasar los valores de conversión a voltaje –y se puede poner el valor máximo de conversión en lugar de 5V- pues solo importa la proporción entre ambas diferencias. Del mismo modo se aprecia también que las resistencias pueden estar expresadas en cualquier múltiplo o divisor del ohmio –aunque en realidad esto no es significativo pensando solo en la resolución del ejercicio, pues son constantes dadas-. Por último, y dado que la ganancia se expresa siempre como un valor entero, según dice el enunciado, podemos usar enteros también para el numerador y el denominador y suponer que se ha elegido la unidad adecuada para que las resistencias estén expresadas como enteros con la precisión suficiente.

Una vez establecido el método de cálculo de h_{FE} basta con considerar el resto del programa –o del fragmento de él- que se pide. Dentro de un bucle infinito se esperará que se oprima el pulsador, se realizarán las tres conversiones y se calculará la ganancia según se ha explicado. Posteriormente se esperará que se libere el pulsador y terminará el bucle, con lo que se volverá a su inicio, a esperar la pulsación. Adicionalmente, aunque no se pide

en el enunciado, se pondrá una variable a 1 o a 0 según el transistor esté o no saturado. Para ello simplemente se comparará el valor convertido de **AN0**, que corresponde a V_{CE}, con una cierta constante que representa, convenientemente ajustada, un umbral máximo para V_{CEsat}. A continuación aparece el código descrito.

```
#define RBASE          xxxxx
#define RCOL           xxxxx
#define V5VOLTS        1023
#define VCESAT         41                          // Unos 0,2V

main()
{
   int conv[3], hfe, num, den, i;
   byte sat;

   // Configuración del sistema
   // …

   while(1) {
      // El pulsador está en RB0, activo a nivel bajo
      while (PORTBbits.RB0 == 1);                  // Esperamos que se pulse

      for (i = 0; i < 3; i++) {                    // Para los 3 canales
          ADCON0bits.CHS = i;                      // Configuramos canal
          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
      }
      if (conv[0] < VCESAT) sat = 1;               // Se indica saturación en
      else sat = 0;                                // su caso
      num = RBASE * (V5VOLTS - conv[0]);           // Se realizan los cálculos
      den = RCOL * (conv[2] - conv[1]);            // indicados arriba y se
      hfe = num / den;                             // obtiene la ganancia

      while (PORTBbits.RB0 == 0);                  // Esperamos que se libere

   }
}
```

Pregunta 6. **(2,5 puntos)** – Programa en lenguaje C una función **`subMatriz`**, cuyo prototipo aparece en el cuadro inferior, que devuelva una submatriz de otra dada. Para ello se pasa a la función, además de la matriz original **`m`**, la fila y columna iniciales de la submatriz **`fIni`** y **`cIni`**, y su número de filas y columnas, **`fil`** y **`col`** respectivamente. Se considera en todos los casos que los índices de fila y columna comienzan en 0.

En caso de que la fila o columna de inicio excedan las dimensiones de la matriz original, se señalará un error. En caso de que las dimensiones de la submatriz, siendo la fila y columna iniciales correctas, excedan las de la matriz dada, se devolverá una submatriz de las dimensiones solicitadas, con ceros en aquellos elementos que superen el número de filas o columnas de la original. El error indicado o cualquier otro se señalará devolviendo una matriz con valores negativos en su número de filas y columnas.

A continuación aparece la estructura de datos para las matrices y el prototipo de la función pedida.

```
struct mat {
   double *data; // Puntero a los datos
   int f, c;     // Número de filas y columnas, respectivamente
};

struct mat subMatriz(struct mat m, int fIni, int cIni, int fil, int col);
```

La función propuesta puede hacerse, como es habitual, de varias formas. En este caso se ha seguido una que, si bien no es la más eficaz, da lugar a código regular y compacto. El algoritmo, basado en dos bucles **`for`** anidados, para filas y columnas, como muchas funciones matriciales, divide tanto el bucle externo –filas- como el interno –columnas- en dos partes. En la primera copia elementos de la matriz original y en la segunda escribe ceros si es necesario. Como cada bucle verifica primero la condición de salida, evita que se añadan columnas a cero en las filas si no es necesario, a costa de tener una comparación que sería inútil si no hay que añadir ceros, por cada fila. En el caso del bucle de filas, el segundo **`for`** añade filas completas de ceros. De no ser necesario, solo tendríamos una comparación adicional, por lo que no reduce significativamente la eficiencia.

Antes de los bucles que se han explicado, la función verifica errores y asigna el límite de fila y columna para los elementos que salen de la matriz original. Las constantes **`ERR_DIM`** y **`ERR_MEM`** estarían definidas con valores negativos para poder señalar de forma diferenciada estos dos errores, respetando la especificación de la función. Veamos a continuación el código comentado.

```
struct mat subMatriz(struct mat m, int fIni, int cIni, int fil, int col)
{
	int resi, resj, mi, mj, ffin, cfin;
	struct mat res;

	// Verificamos los índices de inicio
	if ((fIni >= m.f) || (cIni >= m.c)) {
		res.data = NULL;
		res.f = ERR_DIM;
		res.c = ERR_DIM;
		return res;
	}
	// Reservamos memoria para los datos
	if ((res.data = malloc(fil * col * sizeof(double))) == NULL) {
		res.f = ERR_MEM;
		res.c = ERR_MEM;
		return res;
	}
	// Si no hay errores asignamos las dimensiones
	res.f = fil;
	res.c = col;
	// ffin y cfin son los límites para sacar elementos de la matriz
	// original. Si hay que añadir ceros, serán la dimensión correspondiente.
	// Si no, la suma de la coordenada de origen y la dimension destino
	if (fIni + fil > m.f) ffin = m.f;
	else ffin = fIni + fil;
	if (cIni + col > m.c) cfin = m.c;
	else cfin = cIni + col;
	// Primera parte del bucle de filas. resi es el indice a la matriz
	// resultado res; mi es el índice a la matriz origen m. La condición de
	// salida es que se alcanza el valor de final calculado antes.
	for (resi = 0, mi = fIni; mi < ffin; resi++, mi++) {
		// Primera parte del bucle de columnas, análoga a la anterior
		for (resj = 0, mj = cIni; mj < cfin; resj++, mj++)
			res.data[resi * res.c + resj] = m.data[mi * m.c + mj];
		// Si resj no ha llegado a col se añaden ceros. En otro caso no se entra
		for (; resj < col; resj++)
			res.data[resi * res.c + resj] = 0.0;
	}
	// Como antes, si resi no ha llegado a fil se añaden filas de ceros.
	// En otro caso no se entra.
	for (; resi < fil; resi++)
		for (resj = 0; resj < res.c; resj++)
			res.data[resi * res.c + resj] = 0.0;

	// Se termina devolviendo el resultado
	return res;
}
```