Programación en lenguaje Ce con el entorno Uvi51win

(18/01/09)

1. Introducción

El entorno Uvi51 incorpora en su versión windows framework la posibilidad de programar el microcontrolador utilizando el lenguaje Ce. Ya se han adaptado los programas Uvi51 (asistente), Uvisim (simulador) y Uvialt (depurador). Por lo tanto, a fecha de hoy es posible escribir, compilar, simular, enviar a la placa y depurar programas escritos en lenguaje Ce.

El lenguaje Ce aporta muchas ventajas en comparación con el lenguaje ensamblador, en particular una mayor claridad del código y un menor número de líneas, lo que supone una reducción del tiempo dedicado a generar el código, y una depuración y mantenimiento más fáciles. Sin embargo, en el caso particular de la programación de un microcontrolador carente de sistema operartivo, el lenguaje Ce apenas contribuye a una mayor portabilidad. Entre los inconvenientes de la utilización del lengueje Ce, cabe citar un peor rendimiento tanto en el tamaño del código máquina generado como en la velocidad de ejecución. Para reducir estos inconvenientes, es conveniente seguir las recomendaciones que se proponen en los apartados siguientes.

2. ¿Cómo obtener un buen rendimiento programando en Ce?

Para evitar que el lenguaje Ce consuma mucha memoria de programa y, sobre todo, para evitar que el código se ejecute demasiado lento se deben seguir las siguientes recomendaciones.

TIPOS DE VRIABLES:

1) No utilizar tipos de datos complejos. No se deberían utilizar nunca estructuras (struct), y se deberían evitar los punteros. El compilador utilizado optimiza mejor el manejo de arrays que la aritmética de punteros (punteros genéricos).
2) Evitar utilizar arrays de más de una dimensión.
3) Evitar utilizar tipos de coma flotante (float).
4) Utilizar el tipo bit para booleanos. El tipo bit no existe en Ce normativo, pero sí existe en este compilador.
5) Utilizar tipos sin signo (unsigned) cuando no sea necesario un tipo con signo (signed). Utilizar el tipo más pequeño que sea adecuado para su aplicación: por ejemplo usar un char si no es necesario un int, o bien utilizar un int si no es necesario un long.

ESTRUCTURAS DE PROGRAMA:

1) Evitar utilizar la estructura switch-case.
2) Evitar funciones de interrupción largas o complicadas (ver apdo. 9).

3. ¿Cómo facilitar la tarea de depuración programando en Ce?

Para facilitar la depuración, se aconseja seguir las siguientes recomendaciones.

1) Evitar las líneas de código que contengan gran cantidad de instrucciones. Por ejemplo, en lugar de la línea:
   if(x<5) { a*= 2; b/=5; c= a+b;}
   sería preferible separar de esta forma:
   ```
   if(x<5){
        a*= 2;
        b/=5;
        c= a+b;
   }
   ```
   de manera que al depurar se pueda avanzar instrucción a instrucción, ya que el depurador ejecuta línea a línea.

2) Evitar las esperas muertas. Por ejemplo, la línea:
   while(1);
   puede crear problemas al depurador si se ejecuta paso a paso.

3) Utilizar nombres de variables significativos pero no demasiado largos. Los nombres largos no caben en la tabla de variables del depurador, y se hace preciso mover el puntero del ratón sobre la tabla para poder reconocerlos.

4. ¿Qué se necesita para programar el micro en Ce?

En primer lugar es necesario tener un conocimiento al menos básico del lenguaje Ce y del micrcontrolador. El compilador utilizado está diseñado específicamente para el microcontrolador 8051, y se puede adaptar sin dificultad para el micro 80C537.

Cuando se utiliza el lenguaje ensamblador, normalmente creamos un archivo CNX para definir el sistema. En el caso de programar en Ce, se deben añadir otros archivos, por lo que es preferible crear una carpeta para cada "proyecto", y poner todos los archivos en ella. Un proyecto típico utiliza los siguientes archivos, de los cuales sólo los indicados en negrilla son generados por el usuario. Supongamos que el nombre del proyecto es miprog; en la carpeta .....\ce\simul\miprog\ estarían los archivos si el programa está adaptado para simular (origen 0, etc). Si está preparado para depurar, el path de la carpeta sería .....\ce\depur\miprog\.

miprog.cnx	Archivo de conexionado (el de Uvi51 de siempre). La diferencia está en que la línea principal es en este caso miprog.c en vez de miprog.src. De esta forma, Uvi51 sabe que el fuente se ha escrito en Ce.
miprog.c	Archivo del programa fuente en Ce.
miprog.gen	Archivo de opciones de enlace. Este archivo indica al enlazador (linker) que el código debe empezar en la dirección 0x8000 (para depurar) ó 0x0000 (para simular), y otros aspectos importantes de asignación de variables. El archivo gen se facilita listo para usar, y se recomienda no modificarlo.
miprog.cmp	Archivo de directorio de compilación. Se genera automáticamente al

	ordenar la compilación.
miprog.xcl	Archivo de directorio de enlace. Se genera automáticamente al ordenar la compilación.
lcdnew.r03	Archivo objeto que se enlaza al programa del usuario para acceder al LCD usando el puerto P4 (el equivalente de rutinas.src que se usa con el ensamblador)
lcdnewp0.r03	Idem para utilizar el puerto P0 (adecuado para el simulador)
brkp.r03	Archivo objeto que se enlaza al programa del usuario para poder crear puntos de ruptura empleando la función brkp(). Sólo se emplea con el depurador

El asistente de Uvi51 adapta automáticamente los botones al tipo de lenguaje, por lo que en el caso de un programa en Ce, el botón de ensamblar "–A→" se convierte en un botón de compilar (en realidad compilar y enlazar) "–C→". Los errores de compilación se pueden revisar editando el archivo LST. Aunque se pueden utilizar diferentes editores para generar el archivo C y revisar el LST, se recomienda emplear el editor Notepad o el Crimson. En el caso de usar el Notepad es preferible no utilizar la tecla TAB. En el caso del Crimson es conveniente asignar al TAB un número pequeño de espacios (por ejemplo 2).

Si se generan errores del linker, se puede consultar el archivo MAP.

Cuando se llama al depurador Uvialt pulsando el botón "Debugger", el depurador utiliza el primer programa (micro) definido en el archivo CNX. Se recuerda que el simulador puede operar con 4 micros a la vez, mientras que el depurador sólo utiliza uno. El depurador se adapta al tipo de código fuente. Esto supone cambios internos notables, y cambios menores en la apariencia de la interfaz de usuario.

Los cambios de la interfaz de usuario, cuando se emplea el lenguaje Ce son:

1) La ventana de código muestra el código fuente en Ce.
2) La tabla de símbolos muestra solamente las variables que son "visibles" en cada momento (globales + locales de la función actual). Las locales se muestran sobre fondo amarillo (ver apdo. 11). La información sobre las variables incluye el tipo (int, etc.), pero sólo en la ventana de símbolos principal (no en la de "Selected symbols").
3) Cuando se ordena ejecutar un paso (Step), se ejecuta una línea de código de Ce, que normalmente incluirá muchas instrucciones máquina. Mientras no se alcanza una nueva línea de Ce, aparece un mensaje "waiting". Si tarda más de 15 segundos, aparace un mensaje indicativo y se detiene. Ésto puede suceder cuando se llama a una función de biblioteca (library) o un objeto enlazado con código basado en bucles (retardos, aritmética, etc).

5. ¿Cómo se utiliza la pantalla LCD si se programa en Ce?

Los programas en Ce se enlazan automáticamente con dos módulos objeto. Uno de ellos (lcdnew.r03) proporciona las funciones básicas de manejo de la pantalla LCD, conectada **necesariamente** al puerto P4 (P0 si se trabaja con el simulador). Nota: en la tabla que sigue, uchar indica unsigned char, y uint indica unsigned int.

bit lcdsetup()	Inicializa la pantalla. Devuelve '0' si ok, '1' si falla
void lcdclear()	Borra la pantalla y pone el cursor al principio de la 1ª línea
void lcdhome()	Pone el cursor en la 1ª línea
void lcdsetcursor(uchar posicion)	Pone el cursor en la posición indicada
void lcdputchar(uchar ch)	Muestra el carácter ch (ascii) en la posición actual del cursor
void lcdputstring(uchar *str)	Muestra la cadena de caracteres str a partir de la posición actual del cursor
void lcdputint(uint n)	Muestra el texto del valor decimal sin signo n

Es conveniente esperar un cierto tiempo (2 ms) desde la ejecución de lcdclear() y la escritura en el lcd.

6. Limitaciones

El compilador utilizado no es capaz de compilar programas fuente en Ce de más de unas 550 líneas. Este valor es indicativo, ya que depende de otros factores como el grado de complejidad de cada línea de código. Como ejemplo, todos los proyectos de la asignatura LSP se pueden desarrollar con menos de 500 líneas.

El enlazador no impone restricciones, aunque el archivo GEN utilizado asigna las direcciones 0x8000 a 0x9FFF para el código (8 kbytes) y las direcciones 0xA000 a 0xEFFF para variables en memoria externa. **Nota: El depurador Uvialt solamente actualiza en pantalla aquellas variables de memoria externa que se encuentren en el margen 0xA000 a 0xA0FF** .

7. Asignación de memoria a variables

El enlazador asigna las variables a la memoria siguiendo las directrices indicadas en el archivo GEN (que se recomienda no modificar). Por defecto, asigna las variables de bit a la zona 0x20-0x2F de memoria interna, y las restantes variables desde 0x30 hasta 0x7F de memoria interna. Para constantes se debe utilizar la palabra const como en el ejemplo:

const int x= 5000;

En este caso la "variable" x es en realidad una constante. En algunos casos es preferible utilizar una macro:

#define x 5000

Cuando no hay suficiente memoria interna disponible en el área de direccionamiento directo (DATA, hasta 0x7F) es posible recurrir a la memoria indirecta (IDATA, de 0x80 hasta 0xFF) o a la memoria externa (XDATA). Para ello se utiliza por ejemplo:
idata unsigned int z;
o bien:
xdata signed long w;

En estos casos, el acceso es algo más lento que cuando se emplea la zona de direccionamiento directo, aunque puede ser muy adecuado para ubicar arrays.

Se recuerda que es el linker el que asigna la dirección de las variables. Si no hay espacio para todas, el linker muestra mensajes de error en pantalla y/o en el archivo MAP.

8. Acceso a direcciones fijas de memoria

En algunas situaciones es necesario (o conveniente) acceder a determinadas direcciones físicas de la memoria interna, externa o SFRs. Para ello existen varias opciones.

En el caso de querer acceder a un SFR, se puede utilizar directamente la sintaxis:

```
P4= 0xFF;
```

por ejemplo, que pone todo el puerto P4 a 'unos'. O bien:

```
TMOD= 0x11;     /* configura los temporizadores T0 y T1 a modo 1 */
P3.4= 0;        /* pone el bit 4 del puerto 3 a '0' */
```

Estas asignaciones se pueden realizar siempre que se haya incluido el archivo io517.h utilizando:

```
#include "io517.h"
```

Si se quiere utilizar una zona de XRAM bajo cuenta y riesgo del usuario (debería comprobar que no choca con asignaciones del linker) se puede emplear un puntero (¿qué no se puede lograr con un puntero, para bien o **para mal**?). El puntero se puede definir utilizando un cast, de la forma siguiente:

```
unsigned char *p= (unsigned char *)  0x01D000;
```

con lo que p queda apuntando a la dirección 0xD000 de xram (el 01 indica XRAM, 00 indicaría RAM y 02 indicaría ROM). Entonces:

```
*p= 25;
```

carga el valor decimal 25 en la dirección D000 de la memoria externa. Para leer el valor escrito, se puede hacer:

```
unsigned char x;
x= *p;      /* el contenido de la dirección D000 de xram se copia en x */
```

9. ¿Cómo se definen las funciones de interrupción (ISRs)?

El equivalente a las ISR utilizadas en el lenguaje ensamblador son las funciones de interrupción. Las funciones de interrupción se declaran en el archivo io517.h.

Para definir una función de interrupción (FI), se emplea una sintaxis especial. Por ejemplo, para la FI de desbordamiento del temporizador T0:

```
interrupt  void  T0_int(void)
{
        x++;
}
```

En este ejemplo, la variable global x se incrementa cada vez que entra la interrupción. **MUY IMPORTANTE:** El linker gestiona la asignación de variables usando una técnica de reutilización que permite a varias variables locales compartir la misma dirección de memoria siempre que no se sobreescriban. Para ello estudia el árbol de funciones partiendo de main(). Sin embargo, este proceso no funciona correctamente con las FI, ya que pueden entrar en cualquier momento. Por lo tanto, si es necesario emplear variables en una FI es preciso declararlas globales, o bien **static** dentro de la FI. Al declarar una variable static, el linker la trata como global y no hay conflictos. Recomendación: Si se quiere que la variable sea visible en todo momento, entonces se debe declarar global. Si se quiere que sólo sea visible cuando estamos ejecutando "dentro de la FI", entonces se debe declarar local sin olvidar el **static**. Es importante que este principio se aplique no sólo a las FI, sino también a cualquier función que se llame desde una FI.

Además, si una función se llama desde una FI, no debe llamarse desde otra FI ni desde el árbol de funciones de main(), ya que las funciones de este compilador no son reentrantes. Recomendación: Evitar que las FIs llamen a otras funciones. Hacer FIs lo más sencillas posible (cortas, sin aritmética complicada, etc). No utilizar nunca **recursión** (funciones que se llaman a sí mismas) con este compilador.

Las declaraciones de las FIs disponibles para este micro (io517.h) son:

```
interrupt [0x03] void EX0_int (void);     /* External Interrupt 0 */
interrupt [0x0B] void T0_int (void);      /* Timer 0 Overflow */
interrupt [0x13] void EX1_int (void);     /* External Interrupt 1 */
interrupt [0x1B] void T1_int (void);      /* Timer 1 Overflow */
interrupt [0x23] void SCON0_int (void);   /* Serial Port 0 */
interrupt [0x2B] void T2_int (void);      /* Timer 2 Overflow */
interrupt [0x2B] void T2EX_int (void);    /* Negative Transition on T2EX */
interrupt [0x43] void ADC_int (void);     /* ADC Interrupt */
interrupt [0x4B] void EX2_int (void);     /* External Interrupt 2 */
interrupt [0x53] void EX3_int (void);     /* External Interrupt 3 */
interrupt [0x5B] void EX4_int (void);     /* External Interrupt 4 */
interrupt [0x63] void EX5_int (void);     /* External Interrupt 5 */
interrupt [0x6B] void EX6_int (void);     /* External Interrupt 6 */
interrupt [0x83] void SCON1_int (void);   /* Serial Port 1 */
interrupt [0x9B] void CT_int (void);      /* Compare Timer Overflow */
```

Al definir las FIs en el código no se indica el [0xnn] que va después de "interrupt".

10. Programa de ejemplo

El siguiente código es un ejemplo que muestra la estructura típica de un programa escrito en Ce para este micro. Atención, este compilador no permite emplear dos barras para incluir un comentario, sino que es preciso emplear /* comentario */. Los números de línea se incluyen solamente para facilitar la explicación del funcionamiento del programa, pero obviamente no se escriben en el programa real.

```
1     /* Programa de prueba */
2
3     #include "io51.h"  /* sería "io517.h" para el 80C537 */
4
5     /* variables globales */
6     bit isok;        /* variable booleana */
7     char beta;       /* char es unsigned char por defecto */
8     signed char diferencia;
9     int vel;         /* int es signed int por defecto */
10    unsigned int coches;
11
12    /* prototipos de funciones */
13    int calculo(int n1, int n2);
14
15    /* función main. "StartUp" se ejeucuta antes que main ! */
16    main()
17    {
18      int retardo;         /* variable local de main */
19
20      TMOD= 0x01;          /* prepara el timer 0 en modo 1 */
21      TH0= TL0= 0;
22
23      IE= 0x82;            /* habilita interr. de T0 */
24      TR0= 1;
25
26      retardo= 5;
27      while(1){            /* bucle principal */
28        retardo= calculo(4,5);
29        retardo--;
30      }
31    }
32
33    int calculo(int n1, int n2)
34    {
35      int z;
36      n1= n1-2;
37      z= n1;
38      z= z*n2;
39      return z;
40    }
41
42    interrupt void T0_int()
43    {
44      coches++;      /* incrementa una variable global */
45    }
46
```

11. Visibilidad de los parámetros de funciones

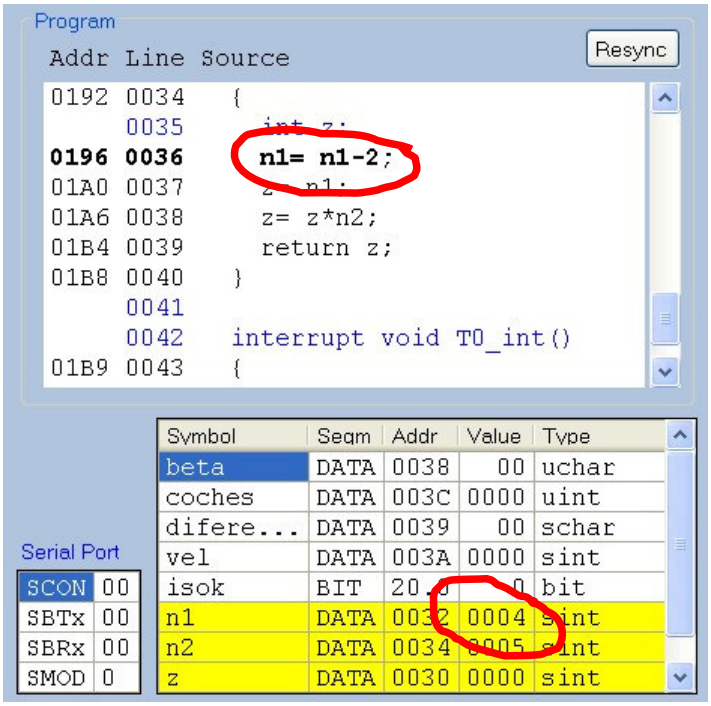
Los parámetros x e y que se pasan a la función calculo() del siguiente ejemplo...

```
int calculo(int x, int y)
{
  .....
}
```

...se comportan de forma parecida a variables locales de la función calculo. El proceso de llamada a la función comienza con el paso de los valores de los parámetros (4 y 5 en el siguiente ejemplo) almacenando sus valores en los registros R7, R6, etc. (depende de la disponibilidad que encuentre el compilador).

```
retardo= calculo(4,5);
```

A continuación se produce una llamada a la subrutina (función). El propio código de la subrutina se encarga de copiar los valores de los parámetros (que todavía están en R7, etc.) a las variables "parámetro" correspondientes de la función (n1 y n2). Esta copia solamente se realiza si existe alguna instrucción que asigne un valor al parámetro. En caso contrario el compilador optimiza el código evitando la copia. Aunque el funcionamiento es correcto, el valor del parámetro mostrado en la tabla de símbolos puede ser incorrecto (normalmente 0) en la tabla de símbolos.



En la figura se muestra el caso en que existe asignación, por lo que el valor de n1 aparece correctamente en la tabla de símbolos (0004).

Si se elimina la instrucción
`n1= n1-2;`
entonces el valor mostrado sería incorrecto (0).

Si dentro de una función es imprescindible monitorizar el valor de un parámetro, y el simulador no lo muestra correctamente, basta forzar una asignación "absurda" como:

```
n1= n1;
```