VISUALIZACIÓN DE NÚMEROS REALES EN LOS MICROCONTROLADORES DE 8 BITS

MARIANO BARRÓN RUIZ
Departamento de Ingeniería de Sistemas y Automática.
Universidad del País Vasco. España.

La optimización del tamaño del código y de la velocidad de ejecución de los programas son dos objetivos que siempre están presentes en la mente del diseñador de sistemas basados en microcontrolador (µC). Las aplicaciones que requieren el procesado de números en coma flotante exigen normalmente un µC dotado de más memoria de código, de más potencia de proceso y en consecuencia de mayor costo de adquisición. En este tipo de aplicaciones uno de los aspectos críticos lo constituye la función encargada de mostrar los resultados de tipo float en los dispositivos de visualización. La solución habitual pasa por el uso de la función sprintf que casi siempre resulta muy lenta y de un tamaño excesivo. En este documento se presenta una solución a este problema y se comparan sus resultados con los de la función sprintf de dos compiladores C comerciales, demostrándose la superioridad de la nueva función. Por otro lado, el listado de la función de salida tiene un elevado valor didáctico ya que su compresión no es posible sin un profundo conocimiento del estándar IEEE-754.

1. Introducción

Son numerosas las aplicaciones basadas en microcontrolador que requieren cálculos con números reales o números que tienen parte entera y parte fraccionaria; en estos casos, un compilador de lenguaje C es una herramienta imprescindible ya que viene acompañado de bibliotecas de funciones matemáticas que operan con tipos de datos float. En cualquier caso siempre resultará necesario adaptar las funciones de entrada y salida de números reales al hardware que utilice el diseño basado en microcontrolador. Para la entrada de números reales desde teclado puede adaptarse la función atof (ASCII to float), que es una función de biblioteca presente en casi todos los compiladores y cuyo código fuente puede encontrarse en la bibliografía [1]. La función atof no requiere demasiado código ni tiene grandes exigencias de velocidad cuando la entrada de números se realiza desde un teclado, ya que la introducción de los datos es mucho más lenta que el procesado de los mismos. Sin embargo la función opuesta ftoa (float to ASCII) no suele estar incluida en las bibliotecas de los compiladores ni su código fuente aparece en la literatura debido a su mayor complejidad; por ello, para mostrar números reales los diseñadores utilizan la función sprintf la cual, exige mucha memoria de código y casi siempre resulta muy lenta. En los siguientes apartados se explica el estándar IEEE 754 utilizado por la mayoría de los compiladores C para representar números reales en coma flotante, se aporta una solución a la función ftoa y se compara sus resultados con los de la función sprintf para dos compiladores comerciales de lenguaje C para microcontroladores de 8 bits.

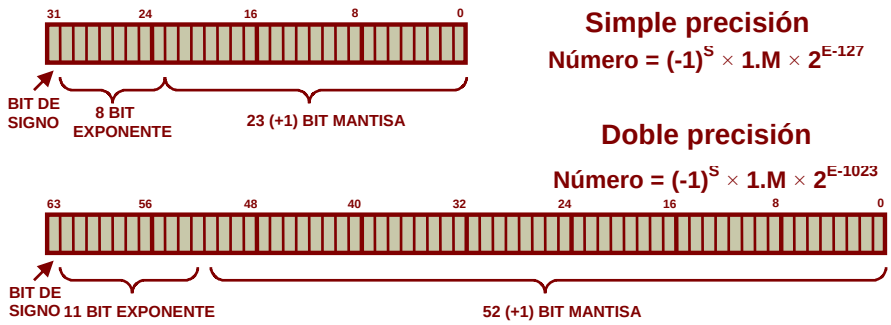

Figura 1. Formatos IEEE 754 de simple y doble precisión

2. Estándar IEEE 754 para representación de números reales

El estándar IEEE 754 determina el formato más utilizado en la actualidad para el tratamiento de números reales en ordenadores y microcontroladores. Básicamente, este formato representa los números reales en notación científica, lo que le permite escribir números muy grandes y muy pequeños con el menor número posible de dígitos, por ejemplo el número 0.0000009876 se representa como 9.876×10^{-7}.

Los números en coma flotante IEEE 754 tienen cuatro componentes básicos: el signo S, la mantisa M, la base que siempre es 2 y no se representa y el exponente E. El primer bit es el bit de signo S que vale 0 para los números positivos y 1 para los números negativos. El estándar contempla varios formatos para representar números con distintos grados de precisión, sin embargo, los microcontroladores de 8 bits usan casi exclusivamente el formato de simple precisión. La figura 1 muestra los formatos IEEE 754 de simple precisión (32 bits) y de doble precisión (64 bits).

La representación IEEE 754 supone que el número siempre está normalizado, lo que significa que el valor de la mantisa se encuentra comprendido entre 1 y 2. Como el bit más significativo de la mantisa siempre vale 1, no necesita ser escrito de forma explícita, de esta forma se consigue representar una mantisa de 24 (o 53) bits utilizando sólo 23 (o 52) bits. Una mantisa de 24 bits proporciona 7 dígitos decimales de precisión ($2^{24} \approx 1.68\times10^{7}$) y una mantisa de 53 bits supone 15 dígitos decimales de precisión ($2^{53} \approx 9\times10^{15}$). El único caso en que la mantisa no empieza por 1 es con el número 0; éste es un caso especial que se representa con todos los bits de la mantisa y del exponente a cero. Nótese que existen dos representaciones para el cero (+0 y -0).

El exponente representa cantidades positivas y negativas pero no utiliza el habitual complemento a 2, por el contrario, el exponente se desplaza en un valor fijo para que los números negativos se representen por números binarios inferiores a los números positivos, facilitando de esta forma la comparación de los exponentes. El desplazamiento, véase la figura 1, es 1023 para el formato de doble precisión y 127 para el formato de simple precisión. En este último formato un exponente de 33 se representa como 160, que corresponde a la suma de 33 y del desplazamiento 127. En el formato de simple precisión los exponentes -127 (todo ceros) y +128 (todo unos) se reservan para los siguientes casos:

1. Exponente +128 (0xFF) y Mantisa = 0 se utiliza para representar los valores $+\infty$ y $-\infty$.
2. Exponente +128 (0xFF) y Mantisa $\neq$ 0 se utiliza para representar resultados fuera de rango y distintos de 0, $+\infty$ y $-\infty$. Por ejemplo: $0*\infty$, $\sqrt{}$(Negativo), 0/0, ∞/∞, $\infty-\infty$, etc. A estos resultados se les denomina de forma abreviada NaN (Not a Number).
3. Exponente -127 (0x00) y Mantisa = 0 se utiliza para representar los valores +0 y -0.
4. Exponente -127 (0x00) y Mantisa $\neq$ 0 se utiliza para representar números desnormalizados.

Un número desnormalizado es aquel cuya mantisa no empieza por 1, y que por ello representa la cantidad $(-1)^S \times 0.M\times 2^{-126}$. La desnormalización permiten representar cantidades más próximas al cero, véase figura 2, pero no suelen ser soportada por los compiladores C de los microcontroladores de 8 bits.

Es muy fácil convertir números reales a formato IEEE 754 y viceversa. El número decimal -21,25 se convierte en primer lugar a su equivalente en base dos: -10101,01. El bit de signo debe ser 1 por tratarse de un número negativo. A continuación debe desplazarse la coma para conseguir que la mantisa se encuentre comprendida entre 1 y 2, lo cual nos conduce al valor $1{,}010101 \times 2^4$. Si se utiliza el formato de simple precisión se obtiene el campo del exponente añadiendo al verdadero exponente, 4, el desplazamiento, 127, obteniéndose el valor 131 = 10000011b. Si se unen todos los campos en una palabra de 32 bits se obtiene [1][1000001 1][0101010 00000000 00000000] = 0xC1AA0000, obsérvese que se ha omitido el bit más significativo (1) de la mantisa.

El proceso de conversión inversa consiste en: extraer el bit de signo, extraer el exponente y restarle el desplazamiento (127), extraer la mantisa y precederla de un 1 seguido de la coma decimal, desplazar la coma el número de posiciones indicadas por el exponente y realizar la conversión de base dos a base diez.

3. Rango de representación de números en el formato IEEE 754

Es posible hablar de dos rangos de representación de los números en coma flotante: el rango correspondiente a los números normalizados exclusivamente y el rango que también incluye a los números desnormalizados [2]. Considerando el formato de simple precisión existen cuatro intervalos de la recta real, véase la figura 2, que el formato IEEE 754 no puede representar:

1. Números negativos menores que $-1.1111111\ 11111111\ 11111111 \times 2^{127} = -(2-2^{-23}) \times 2^{127} \approx -2^{128}$
2. Números negativos mayores que $-0.0000000\ 00000000\ 00000001 \times 2^{-126} = -2^{-149}$
3. Números positivos menores que $+0.0000000\ 00000000\ 00000001 \times 2^{-126} = +2^{-149}$
4. Números positivos mayores que $+1.1111111\ 11111111\ 11111111 \times 2^{127} = (2-2^{-23}) \times 2^{127} \approx +2^{128}$

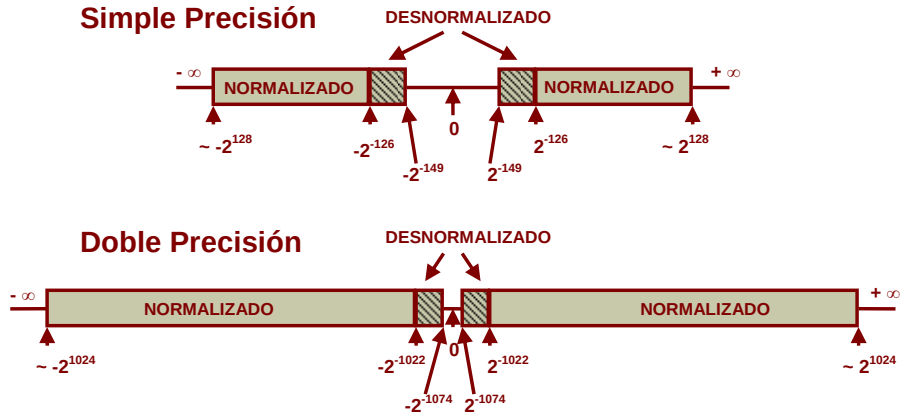

Figura 2. Rango de representación de los Formatos IEEE 754 de simple y doble precisión

La figura 2 muestra que el rango de representación correspondiente al formato de doble precisión es notablemente más grande que el del formato de simple precisión. Los formatos desnormalizados permiten en ambos casos una mayor aproximación al valor cero.

4. Función ftoa

La función ftoa (float to ASCII) descrita en este apartado convierte un número normalizado en el formato IEEE 754 de simple precisión en una cadena ASCII terminada en cero y la almacena en un array. El prototipo de la función es:

unsigned char ftoa(float valor, unsigned char arr[14]);

La función retorna el valor: 1 si **valor** es +INF; 2 si es –INF; 3 si es NaN, o 0 en cualquier otro caso. La función escribe como máximo 7 dígitos decimales para la mantisa, 2 dígitos decimales para el exponente y 13 caracteres para toda la cadena, como sucede por ejemplo en "-1.234567e-18". Para almacenar el resultado se utiliza un array de 14 bytes que permita alojar los 13 caracteres de salida y el carácter terminador. La cadena resultante almacenada en arr[14] tiene el siguiente formato:

[-] dígito [,] [dígito/s]	si	$1{,}0 \leq \lvert \text{valor} \rvert \leq 9.999.999$
[-] dígito [,] [dígito/s] [-] e [dígito/s]	si	$\lvert \text{valor} \rvert < 1{,}0$ o si $\lvert \text{valor} \rvert > 9.999.999$
INF	si	valor $\geq 2^{128}$
-INF	si	valor $\leq -2^{128}$
NaN	si	valor = NaN

Como puede deducirse de lo anterior, la cadena de salida carece de exponente decimal si el valor absoluto del resultado está comprendido entre 1,0 y 9.999.999; los resultados cuyo valor absoluto sean mayores que esta última cantidad o menores que 1 se representan con mantisa y exponente. Se ha elegido este formato de salida para conservar la máxima precisión de la mantisa de 7 dígitos decimales. A continuación se muestra el listado de la función ftoa para un microcontrolador 8051:

```
#define U8        unsigned char
#define ZERO      0x00000000
#define NAN       0xFFFFFFFF
#define plusINF   0x7F800000
#define minusINF  0xFF800000
#define NDIG      7               // Maximo número de dígitos decimales de la mantisa
#define BO        0               // Byte ordering: 0 = Big endian, 3 = Little endian
#pragma FLOATFUZZY (0)

// mystrcpy copia una cadena residente en la memoria de programa en la memoria de datos
extern void mystrcpy(U8 idata *dst, U8 code *src)

union ful {                       // Unión para facilitar el manejo separado de la
   float          fl;             // mantisa y del exponente
   unsigned long  ul;
   unsigned char  uc[4]; };

U8 ftoa(float fnum, U8 idata *str)// La función retorna 0 si fnum no es +INF, ni –INF, ni NaN
{
bit bpoint=0;                     // bpoint = 1 después de escribir la coma decimal
U8 i, d;
char dexpon= 0;                   // Exponente decimal
union ful un;
un.fl = fnum;
if (un.ul == NAN)    {mystrcpy(str, "NaN");   return 3;}
i=1;                              // La función retorna 1 si +INF o i++ (2) si -INF
if (un.uc[BO] & 0x80){un.uc[BO] ^=0x80; i++; *str++='-';} // Si fnum<0 entonces fnum=-fnum
if (un.ul == ZERO)   {mystrcpy(str, "0");     return 0;}
if (un.ul == plusINF){mystrcpy(str, "INF");   return i;}  // i=1 si +INF, 2 si -INF

if (un.fl<1.0) {
   while (un.fl<1e-10) {un.fl *= 1e10;  dexpon-=10;} // Lograr 1.0 <= un.fl < 10.0 sin
   while (un.fl<1e-3)  {un.fl *= 1e3;    dexpon-=3;} // usar divisiones y minimizando
   while (un.fl<1.0)   {un.fl *= 10.0;    --dexpon;} // el número de multiplicaciones
}
else {
   while (un.fl>=1e10) {un.fl *= 1e-10; dexpon+=10;}
   while (un.fl>=1e3)  {un.fl *= 1e-3;   dexpon+=3;}
   while (un.fl>=10.0) {un.fl *= 0.1;    ++dexpon;}
}

// A partir de este punto trabajamos en coma fija, reservando el byte 1 para
// la parte entera y los bytes 2, 3 y 4 para la parte fraccionaria
un.ul <<= 1;                      // Mover el exponente al byte más significativo y restar
d = un.uc[BO] - 127;              // el desplazamiento. Como (1.0<un.fl<10.0), d puede valer:
                                  // 0,1,2 o 3. (1 = 1.0·2**0, y  9 = 1.001·2**3)
un.uc[BO] = 1;                    // Borrar el exponente, re-insertar el bit oculto, o parte
un.ul <<= d;                      // entera, y ajustar el exponente binario a cero.

*str++ = '0' + un.uc[BO];         // Escribir el primer dígito decimal y
if((dexpon < 1)||(dexpon>6))      // si dexp < 1, o dexp>6
   {*str++=','; bpoint = 1;}      // escribir la coma decimal tras el primer dígito

for(d=NDIG-1; d!=0 ;d--)          // Escribir los siguientes 6 (NDIG-1) dígitos de la mantisa
   {
   un.uc[BO] = 0;                 // Parte entera = 0.
   un.ul += (un.ul<<2);           // Para multiplicar un.ul por 10 multiplicar primero
   un.ul <<= 1;                   // por 5 y luego multiplicar por 2
   *str++ = '0' + un.uc[BO];      // Escribir el siguiente dígito decimal
   if (bpoint == 0)               // Si todavía no se ha escrito la coma decimal
      {
      if(--dexpon == 0)           // Escribir la coma decimal cuando dexpon == 0
         {*str++=','; bpoint = 1;}
      }
   }
while( str[-1] == '0')  str--;    // Borrar los ceros finales después de la coma decimal
if(str[-1] == ',')      str--;    // Si el ultimo carácter es la coma decimal, borrarla

if(dexpon != 0) {
   *str++='e';                    // Escribir el exponente con un máximo de 2 dígitos decimales
   if (dexpon<0) {*str++='-'; dexpon=-dexpon;};
   d = (U8)dexpon;
   if (d>9) *str++='0'+d/10;
   *str++='0'+d%10;
   }
*str=0;                           // Terminar la cadena de salida, retornar 0 y salir
return 0;
} // Fin de ftoa
```

La función ftoa compara el valor de entrada fnum con NaN (Not a Number), si la comparación resulta cierta escribe la cadena "NaN" en el array de salida y termina retornando el valor 3. Si la comparación con NaN resulta falsa, la función comprueba el signo de fnum y si se trata de un número negativo escribe un signo menos en el array de salida y cambia el signo de fnum para continuar el tratamiento utilizando siempre valores positivos. Inmediatamente después compara fnum con cero y con mas infinito, si la primera comparación es cierta la función termina retornando el valor 0, si la segunda comparación es cierta la función termina retornando el valor 1 o 2 según se trate de +INF o de –INF. Si todas las comparaciones anteriores resultan falsas el valor de entrada fnum se divide o multiplica por potencias de 10 hasta conseguir que $1{,}0 \leq \text{fnum} < 10{,}0$. Cada vez que fnum se multiplica o divide por una potencia de 10 se ajusta el exponente decimal (variable dexpon) para retener el valor correcto de fnum. Debido a que la división en coma flotante es mucho más lenta que la multiplicación todas las divisiones por potencias de diez (10^n) se sustituyen por multiplicaciones por su valor inverso 10^{-n}. Tanto si el valor inicial fnum es menor que 1,0 como si es mayor que 10,0 se han dispuesto tres bucles while con objeto de minimizar el número de multiplicaciones. Al finalizar este proceso se obtiene un valor flotante (fnum) en el rango 1,0 a 9,999999 y un exponente decimal (dexpon) tales que el valor $\text{fnum} \cdot 10^{\text{dexpond}}$ es idéntico al valor inicial que se pretende convertir a cadena ASCII.

A partir de aquí, el resto del proceso se realiza trabajando en coma fija para obtener un código más rápido. El número en coma fija también ocupa 32 bits (4 bytes); el primer byte contiene la parte entera y los otros 3 bytes contienen la parte fraccionaria. Empieza ahora el envío de dígitos decimales al array de salida; en primer lugar se toma la parte entera del número, se convierte a un carácter ASCII y se almacena en el array de salida; seguidamente se le resta al número inicial la parte entera para quedarnos con la parte fraccionaria; ésta se multiplica por 10 para obtener el primer dígito después de la coma, así volvemos a tener una parte entera y una parte fraccionaria y repitiendo el proceso 6 veces tendremos los siete dígitos de la mantisa guardados en el array de salida. La función siempre escribe una mantisa de 7 dígitos y una coma decimal; si la mantisa termina en uno o varios ceros situados a la derecha de la coma se eliminan y si tras esta operación a la coma no le sigue ningún dígito, también se elimina. Después de escribir la mantisa y si el exponente decimal es distinto de cero, se añade a la cadena de salida el carácter 'e' (exponencial), seguido del signo del exponente (solo si es negativo) y de uno o dos dígitos decimales correspondientes al exponente decimal. Una vez terminado este proceso la función ftoa retorna devolviendo el valor 0 para indicar que el valor escrito es distinto de +INF, -INF y de NaN.

El compilador Keil C51 [3] dispone de una directiva FLOATFUZZY (n) que controla el número de bits que se ignoran cuando se realiza una comparación en coma flotante. Si no se especifica ningún valor el compilador ignora los 3 bits menos significativos de la mantisa. Para que la función ftoa descrita opere correctamente es muy importante forzar al compilador a que realice las comparaciones usando todos los bits, lo cual se consigue por medio de la sentencia "#pragma FLOATFUZZY (0)".

5. Ordenamiento de bytes

Un gran número de microcontroladores disponen de una memoria de datos organizada en unidades de 8 bits también llamadas bytes. Muchos elementos (direcciones, flotas, arrays, strings, etc.) poseen un tamaño superior al byte y deben ser almacenados en un conjunto consecutivo de bytes. Cuando se utilizan estos elementos multi-byte surge el problema de la determinación del orden de almacenamiento. Hay dos métodos de almacenamiento: little-endian y big-endian [4]. En el método little-endian el byte menos significativo se almacena en la dirección de memoria más baja; en el método big-endian es el byte más significativo el que se almacena en la dirección de memoria más baja, véase tabla 1.

Ordenamiento	Dirección + 0	Dirección + 1	Dirección + 2	Dirección + 3
Little-endian	MMMM MMMM	MMMM MMMM	EMMM MMMM	SEEE EEEE
Big-endian	SEEE EEEE	EMMM MMMM	MMMM MMMM	MMMM MMMM

Tabla 1. Métodos de ordenamiento de floats y datos multi-byte.

Nótese que los métodos de almacenamiento little-endian y big-endian sólo reordenan los bytes, nunca los bits o los nibbles que los componen. Uno de los compiladores utilizados en la confección de este artículo, el Keil C51 [5] utiliza la representación de números reales IEEE 754 y almacena los floats siguiendo el ordenamiento big-endian; el segundo compilador utilizado, el Codevision AVR C Compiler [6] utiliza el ordenamiento little-endian. Para que la función ftoa descrita en este trabajo pueda adaptarse a ambos tipos de ordenamiento se define la constante BO (Byte Ordering) con el valor 0 o el valor 3, véase el listado de la función ftoa.

6. Comparación entre las funciones sprintf y ftoa

La tabla 2 recoge los resultados obtenidos con dos compiladores (CodevisionAVR C compiler V1.24.7e y 8051 Keil C compiler V7.50) al utilizar la función de biblioteca sprintf y la función ftoa descrita en este documento. Para cada compilador se ha utilizado un programa que solo se diferencia en el uso de la función sprintf o ftoa. El compilador Codevision utiliza 2.324 bytes más con la función sprintf que con la función ftoa y la ejecución del programa se demora 283.696 ciclos, lo que supone una ejecución más de 6 veces más lenta; además si el resultado de la multiplicación excede el rango de representación, éste compilador no proporciona el resultado +INF, sino $3{,}40282\times10^{+38}$ o $\sim2^{128}$. El compilador Keil C51 utiliza 1.444 bytes más con la función sprintf que con la función ftoa y su ejecución es apreciablemente más lenta.

CodevisionAVR C compiler V1.24.7e		8051 Keil C compiler V7.50	

`void main(void)`
`{`
`U8 i;`
`U8 RESULT[14];`
`float f1=1.0;`
`for(i=0; i<12; i++)`
`  {`
`  f1*=-3141.593;`
`sprintf(RESULT,`
`"%7.6e", f1);`
`  }`
`}`
	`void main(void)`
	`{`
	`U8 i;`
	`U8 RESULT[14];`
	`float f1=1.0;`
	`for(i=0; i<12; i++)`
	`  {`
	`  f1*=-3141.593;`
	`ftoa(f1, RESULT);`
	`  }`
	`}`
		`void main(void)`
		`{`
		`U8 i;`
		`U8 RESULT[14];`
		`float f1=1.0;`
		`for(i=0; i<12; i++)`
		`  {`
		`  f1*=-3141.593;`
		`sprintf(RESULT,`
		`"%7.6e", f1);`
		`  }`
		`}`
			`void main(void)`
			`{`
			`U8 i;`
			`U8 RESULT[14];`
			`float f1=1.0;`
			`for(i=0; i<12; i++)`
			`  {`
			`  f1*=-3141.593;`
			`ftoa(f1, RESULT);`
			`  }`
			`}`

1912 words, (3824 bytes) 333,792 ciclos	750 words, (1500 bytes) 50,096 ciclos	2575 bytes 43,023 ciclos	1131 bytes 27,852 ciclos

Tabla 2. Comparativa de las funciones sprintf y ftoa para dos compiladores comerciales.

7. Conclusiones

La memoria de código es siempre un recurso limitado en los microcontroladores; el uso de la función sprintf para mostrar números reales requiere mucha memoria de código y casi siempre es muy lenta. Si se utiliza la función ftoa descrita en este trabajo se puede conseguir un ahorro en la memoria de código de entre 1.400 y 2.300 bytes. Por otro lado, al utilizar la función ftoa en lugar de la función de biblioteca sprintf puede obtenerse un código hasta un 666% más rápido.

Referencias

[1] Brian W. Kernighan, Dennis M. Ritchie. *El lenguaje de programación C*. 2ª edición. Prentice-Hall, (1991).

[2] Steve Hollasch, *IEEE Standard 754 Floating Point Numbers*, Febrero 2006, http://stevehollasch.com/cgindex/coding/ieeefloat.html,

[3] Keil Elektronik GmbH, *Cx51 Compiler User's Guide* 09.2001.

[4] J. Ganssle, M. Barr, *Embedded Systems Dictionary*, CMP Books, Lawrence, KS, 2003.

[5] Keil Software, Inc, www.keil.com

[6] HP Infotech Codevision AVR C compiler, www.hpinfotech.ro.