ESCUELA UNIVERSITARIA POLITÉCNICA DE LA ALMUNIA
UNIVERSIDAD DE ZARAGOZA

Programación orientada a Microcontroladores (MCUs)

C/C++ Embebido

AUTOR NAOUFAL EL RHAZZALI

Contenido

Secciones de código

Figuras

Tablas

Prefacio

Este repositorio tiene el ánimo de servir como referencia a todos aquellos que quieren introducirse en los lenguajes C/C++ y el mundo de los sistemas Embebidos. Así se proporciona principalmente un libro escrito por Naoufal El Rhazzali, donde se recoge toda la teoría necesaria para introducirse en el mundo antes citado. Además, se irán agregando diferentes materiales, como software, hardware o bibliografía interesante.

NOTA: Este libro está en fase de redacción, por lo que puede incluir errores.

Introducción

Cuando se trata de la programación de un Microcontrolador (MCU), se ha de tener una perspectiva global de dicha tarea. Al final para que un MCU puda desempeñar una tarea determinada, debe tener las instrucciones de dicha tarea en algún lugar llamado memoria. No obstante, dichas instrucciones habrán llegado a dicha memoria de alguna forma. Por lo tanto, la tarea de programar consiste en generar una serie de instrucciones y "quemarlos" en la memoria del MCU para que éste haga dicha tarea.

Para entender mejor la idea planteada en el párrafo anterior, conviene separarla en dos partes: PROGRAMACIÓN y MCU.

Programación

Programar combina dos elementos principales:

- Variables. La información que se quiere procesar, esto es, la materia prima.
- Lenguaje. Las herramientas que permiten manipular las variables, creando algoritmos y controlando el flujo de ejecución de tal forma que se puedan implementar tareas concretas y acciones de control determinadas.

Con ánimo de entender mejor la idea planteada arriba, repárese en que, en una planta industrial, donde se está desarrollándose una actividad de manufactura, rara vez la materia prima es procesada una vez llega a la planta. Normalmente, ésta se deposita en almacenes, en espera de que sea solicitada por producción. Entonces, las variables así se tratan. Primero son declaradas (almacenadas en memoria), para posteriormente manipularlas.

Con ánimo de facilitar el entendimiento, se va a centrarse en los lenguajes C/C++, que son ampliamente utilizados en la programación de microcontroladores (MCUs).

Con la idea principal clara: programar son variables y control del flujo de ejecución; a continuación, se va a explorar C/C++, para ver cómo implementan dichos lenguajes dicha idea.

C/C++

El lenguaje C, y su evolución a C++, es un lenguaje que se compone de una serie de palabras reservadas. Las palabras reservadas son el lenguaje en sí, las cuales entiende el compilador y es capaz de traducir a código máquina. Para entender esto, imagínese que en “Google Traductor” pone traducir del “Español” al “Inglés”, y pone en el cuadro de texto “jdhsjdh”. “Google Traductor” no será capaz de traducir eso, porque no existe ninguna palabra en español que sea “jdhsjdh”. Lo mismo pasaría en C/C++, si escribimos alguna palabra que no pertenezca a las palabras reservadas del lenguaje, el proceso de compilación (traducción) caería en error y no se completaría satisfactoriamente.

Entendido lo anterior, el siguiente paso es conocer dichas palabras reservadas. No obstante, conviene categorizar dichas palabras. Así, existe un grupo de palabras orientados a un uso, y otro grupo a otro. Según dicha clasificación, se va a sumergirse en el lenguaje.

Memoria

Cualquier programa informático manipula información. Dicha información ha de ser almacenada en algún lugar para que no se pierda el hilo de modificaciones y se pueda

leer/escribir cuando sea necesario. Por ello, cuando se quiere trabajar con alguna información, primero se debe reservar una estantería en memoria para dicha información. Generalmente la unidad mínima de información es el bit, pero las memorias se organizan en bytes. Esto último, es lo que se entiende por longitud de estantería. Así, una estantería de longitud un byte, pude albergar 8 bits, otra estantería de 2 bytes, puede albergar 16 bits. Esto último desencadena el concepto de arquitectura de MCU. A pesar de que arquitectura puede también usarse para referirse a otros conceptos en un MCU, cuando se habla de una arquitectura de 8 bits, se refiere a que la longitud de estantería es de 8 bits. Así, existen arquitecturas de 8bits, 16bits, 32bits y 64bits.

Entendido lo anterior, la pregunta que se debe plantearse es: ¿cómo se reserva un espacio en memoria a través de C/C++? Para ello, C/C++ ofrece las siguientes palabras reservadas, mostradas en la **Sección de código 1**.

```
/*  VARIABLES TYPES IN C/C++    */

                char  var;
               short  var;
           short int  var;
                 int  var;
                long  var;
           long long  var;
       long long int  var;
       unsigned long  var;
    unsigned long int  var;
unsigned long long int  var;
               float  var;

/*  VARIABLES TYPES IN C/C++    */
```

Sección de código 1 – Tipos de variables en C/C++

Cada una de dichas palabras reservadas permiten reservar una o más estanterías de memoria. Una vez reservadas las estanterías correspondientes, a lo largo del ámbito de declaración de dicha variable, la dicha será identificada mediante su nombre “var”. Esto último, es de suma importancia de cara a entender el lenguaje y su relación con el MCU.

El tamaño en bytes que reserva cada palabra reservada depende del compilador. Por ello, con el MCU Atmega328 del Arduino UNO, se ha realizado un pequeño código para comprobar el tamaño de cada variable según la declaración hecha.

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  static bool printOnce = true;

  if(printOnce){

    Serial.print("char size:  ");
    Serial.print(sizeof(char));
    Serial.println(" bytes.");
```

```
    Serial.print("short size:  ");
    Serial.print(sizeof(short));
    Serial.println(" bytes.");

    Serial.print("short int size:  ");
    Serial.print(sizeof(short int));
    Serial.println(" bytes.");

    Serial.print("int size:  ");
    Serial.print(sizeof(int));
    Serial.println(" bytes.");

    Serial.print("long size:  ");
    Serial.print(sizeof(long));
    Serial.println(" bytes.");

    Serial.print("long long size:  ");
    Serial.print(sizeof(long long));
    Serial.println(" bytes.");

    Serial.print("long long int size:  ");
    Serial.print(sizeof(long long int ));
    Serial.println(" bytes.");

    Serial.print("unsigned long size:  ");
    Serial.print(sizeof(unsigned long));
    Serial.println(" bytes.");

    Serial.print("unsigned long int size:  ");
    Serial.print(sizeof(unsigned long int));
    Serial.println(" bytes.");

    Serial.print("unsigned long long int size:  ");
    Serial.print(sizeof(unsigned long long int));
    Serial.println(" bytes.");

    Serial.print("float size:  ");
    Serial.print(sizeof(float));
    Serial.println(" bytes.");

    printOnce = false;
  }
}
```

Sección de código 2 – Conocer el tamaño de variables en Arduino

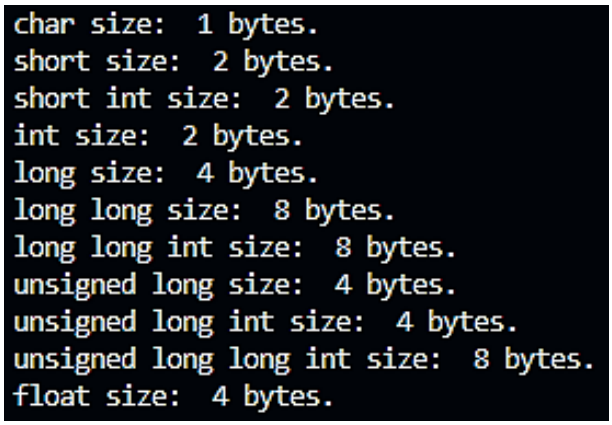

Figura 1 – Salida de la Sección de código 2

Tras ejecutar el código de **Sección de código 1**, se puede visualizar en pantalla la salida mostrada en la **Figura 1**. Se puede ver que un "char" ocupa un 1 byte, mientras que "int" 2 bytes, y float 4 bytes.

Entendido lo anterior, queda claro que, si la información que se quiera almacenar tiene un rango que va de 0 a 255, entonces una variable de tipo "char" sería suficiente. Ahora si se sabe que la variable tiene valores mayores que 255, entonces habría que pensar en una variable tipo "short", "int" o incluso "long long".

Otra cuestión importante a tratar es el signo. Cuando se declaran variables enteras, el compilador entiende que dichas variables tienen signo, por lo que reserva el último bit (o MSB del inglés most significant bit) para resolver el signo, quedando reducido el rango de valores que puede tomar la variable. Así, cuando se sabe que la variable con la que se va a trabajar no tiene signo, es mejor añadir el modificador "unsigned" para poder aprovechar todos los bits, y disponer de un rango mayor.

Por otra parte, todos los tipos de variables mostrados en la Sección de código 1, se pueden clasificar en tres grupos bien diferenciados:

- Naturales.
- Enteros.
- Flotantes (o de coma flotante).

Así, según la categorización señalada, los tipos de variables quedarían agrupados como muestra la **Tabla 1**.

Naturales	Enteros		Flotantes
	Signed	Unsigned	
char	*short* *short int* *int* *long* *long long* *long long int*	*unsigned short* *unsigned short int* *unsigned int* *unsigned long* *unsigned long long* *unsigned long long int*	*float*

Tabla 1 – Categorización de los tipos de variables en C/C++

Las variables char, no tienen signo y toman valores del 0 al 255. Generalmente, se suelen emplear para codificación de caracteres. No obstante, no dejan de ser más que números, así si se asigna a una variable "char" el valor de 65, las funciones de impresiones por pantalla de las librerías estándar de C o de las librerías de Arduino

mostrarían una “A”. Esto último, es porque 65 en codificación ASCII es una “A”. No obstante, a efectos de cálculo, es un número sin más.

Las variables enteras, sí que se entienden como números. En caso de un “int” el valor que puede tomar la variable es de -32.767 hasta +32.767. Ahora si se trata de un “unsigned int”, en ese caso la variable puede tomar valores desde 0 hasta los 65.535. Más tarde, se profundizará en la codificación de cada tipo de variables, tanto en binario como en hexadecimal.

finalmente, las variables flotantes son valores decimales. Éstos tienen una codificación un tanto especial, por ello, queda por desarrollar este asunto en futuras ediciones de este libro.

Control de flujo

If

…

If-else

…

For

…

While

…

Do-while

…

Switch-case

…

Código interesante - skip

¿Par o impar?

Se cumple que cualquier número impar, al convertirlo en binario, su LSB (del inglés les significant bit) será siempre “1”. Así, si se aprovecha dicha información se puede escribir el siguiente código.

```
int num = 0;
    if ( num & 1 ){
        /* do something when the number is odd  */
    }
    else {
        /* do something when the number is even  */
    }
```

Sección de código 3 – Comprobar impar o par con el operador &

El programa de la **Sección de código 3** se basa en el operador a nivel bits: &. Este operador realiza una operación “and” bit a bit a las variables de entrada.

	Binario	Decimal
“num = 18”	0001 0010	18
1	0000 0001	1
&	-	-

“(num & 1)”	0000 0000	0

Tabla 2 – Resultado de la operación & en caso par

	Binario	Decimal
“num = 21”	0001 0101	21
1	0000 0001	1
&	-	-
“(num & 1)”	0000 0001	1

Tabla 3 – Resultado de la operación & en caso impar

En la **Tabla 2** y la **Tabla 3** se puede observar que el resultado de la operación es siempre “1” en caso impar, y siempre “0” en caso par.

No obstante, también existe otra forma de resolver este mimso problema. Ello es mediante el operador resto: %. Ahora el programa tomaría la forma que se muestra en la **Sección de código 4**.

```
int num = 0;
    if ( num % 2 ){
        /* do something when the number is odd  */
    }
    else {
        /* do something when the number is even  */
    }
```

Sección de código 4 – Comprobar impar o par con el operador %

Generalmente, el primer programa (**Sección de código 3**) es más eficiente que el segundo (**Sección de código 4**). No obstante, el primero no tiene garantías para trabajar con números negativos. Al final del día, la mayoría de los compiladores actuales optimizan tanto el primer programa como el segundo al mismo ensamblador. Por esto último, convendría trabajar con el segundo programa, de cara a una mayor claridad de código.

MCU

Un microcontrolador es una solución digital combinacional/secuencial, que permite computación. Esto quiere decir que permite realizar una acción de control, a partir del procesamiento de información representada en la unidad de medida del bit. Para entender mejor un MCU, se ha de tener conocimientos básicos en electrónica analógica y digital.

CPU

Instrucción - skip

Cada instrucción toma tres etapas para ser ejecutada en la CPU.

Fetch → decode → execute

Lo anterior explica que primero es “fetch”, lo cual carga la instrucción a la memoria. Lo segundo es “decode” lo cual decodifica la instrucción. Finalmente, “execute” lo cual ejecuta la instrucción. Cada una de dichas etapas cuesta un ciclo de reloj para la CPU.

Una nota importante a señalar, cuando se ejecutan instrucción de “branching”, esto es, saltos, el proceso de ejecución de una instrucción se corrompe, y la CPU debe reiniciar

el proceso. Esto último provoca retardos en vano. Por lo tanto, en aplicaciones de alta criticidad de tiempo se debe evitar el uso de declaración de control de flujo tipo: “while”, “for”, “if”, “if-else”, etc. No obstante, en la mayoría de aplicaciones dichos retardos no tienen mayor importancia.

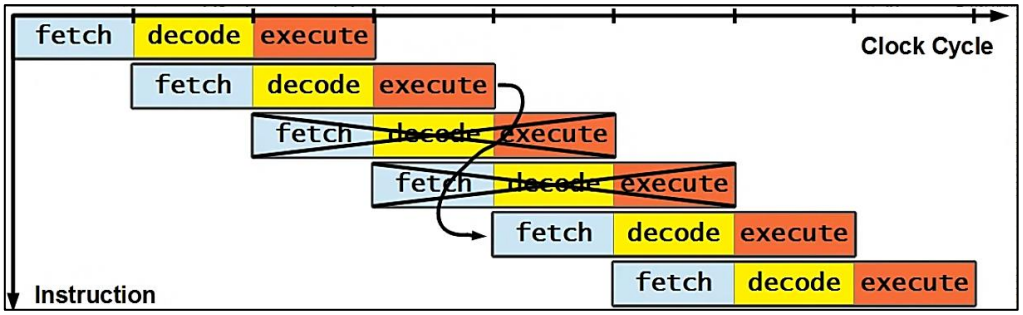

Figura 2 – flujo de ejecución de una instrucción en la CPU [1]

PC – Program Counter - skip

El contador de programa o Program Counter, o en adelante PC, es un registro encargado de apuntar a la “estantería de la flash” que contiene la instrucción a ejecutar a continuación. Cada instrucción al ser ejecutada por la CPU incrementa el PC como efecto secundario. Por ello, el incremento del PC está embebido en todas las instrucciones que ejecuta la CPU.

Bibliografía

[1] Quantum Leaps, LLC, «How to change the flow of control through your code,» 2013.