Programación de Microcontroladores AVR con AVR-Libc

Tabla de contenidos

1. Ejemplo y Análisis

Para la programación en C de dispositivos AVR, los usuarios disponemos de la librería Avrlibc y la herramienta compilación Avr-GCC.

El manual en línea de Avrlibc brinda algunos programas de ejemplo sobre el uso de la misma. A continuación se tiene uno de éstos códigos, que posteriormente se desglosa y explica.

```
//**************************************
//**************************************
//  AVRLIB-DEMO
//  For avrlib and avrmini development board.
//
//  File:     button.c
//  Author:   Michael Gurevich
//  Date:     Sept. 22, 2002
//  Modified: June 9, 2004

#include <avr/io.h>
#include <avr/signal.h>

#include "global.h"
#include "timer.h"

#define DEBOUNCE_THRESHOLD 1000
void checkButton(void);
int main(void)
{
        // set led pins (low 4 bits) as outputs,
        // set button pins (high 4 bits) as inputs
        outb(DDRD, 0x0F);

        // Turn off LEDs by setting the low 4 bits
        outb(PORTD, 0x0F);

        // loop forever
        while(1) {
          // call the checkButton function
          checkButton();
        }

        return 0;
}

void checkButton(void) {
  static u16 buttonDownCounter;
       if (! bit_is_set(PIND,4)) {
          if (buttonDownCounter++ == DEBOUNCE_THRESHOLD) {
            cbi(PORTD,0);
          }
        }
        else { //button is not pressed
          buttonDownCounter = 0;
          sbi(PORTD,0);
        }
}
```

La primera parte usualmente son comentarios que describen el código y para qué fue escrito. También se incluye información sobre el autor y las fechas de creación y última modificación del código.

```
//**************************************
//**************************************
//  AVRLIB-DEMO
//  For avrlib and avrmini development board.
//
//  File:     button.c
//  Author:   Michael Gurevich
//  Date:     Sept. 22, 2002
//  Modified: June 9, 2004
```

Luego siguen los `#include`. Éstos se utilizan para indicarle al compilador dónde ubicar ciertos códigos que dan funcionalidades que usamos en nuestro código, pero que se encuentran en otros archivos. Estos archivos usualmente tienen una extensión `.h`, conocidos como headers, y contienen macros y prototipos de funciones. Cuando compilamos el código, el compilador literalmente agrega el código contenido en los headers que se hayan incluido.

```
#include <avr/io.h>
#include <avr/signal.h>

#include "global.h"
#include "timer.h"
```

A continuación se encuentra un `#define` ó definiciones. Éstos son simplemente macros. Antes de que el código sea compilado, el argumento de la definición se sustituye cada vez que aparezca el nombre de la definición en el código. Usualmente se utilizan para darle un nombre significativo a valores que son constantes a lo largo de la ejecución del programa.

```
#define DEBOUNCE_THRESHOLD 1000
```

Luego se tiene un prototipo de función. Dentro de ésta, se definen el nombre de la función así como los tipos de datos de que dicha función tomará como argumentos y entregará como salida. Pueden estar incluidas en los headers o en el mismo código antes de la definición de la función.

```
void checkButton(void);
```

Luego aparece la función `main`. Esta es la parte principal del código. Sólo puede existir una sola función `main` en el código final compilado. Las instrucciones contenidas dentro de esta función son ejecutadas secuencialmente, es decir, una línea a la vez.

```
int main(void)
{
        // set led pins (low 4 bits) as outputs,
        // set button pins (high 4 bits) as inputs
        outb(DDRD, 0x0F);

        // Turn off LEDs by setting the low 4 bits
        outb(PORTD, 0x0F);

        // loop forever
        while(1) {
          // call the checkButton function
          checkButton();
        }

        return 0;
}
```

Finalmente, se tienen las definiciones de las funciones. Son partes de código que son llamados desde la función `main` o de otras funciones, y que poseen funcionalidades específicas que puedan ser utilizadas repetidamente.

```
void checkButton(void) {

  static u16 buttonDownCounter;
       if (! bit_is_set(PIND,4)) {
          if (buttonDownCounter++ == DEBOUNCE_THRESHOLD) {
```

```
            cbi(PORTD,0);
          }
        }
        else { //button is not pressed
          buttonDownCounter = 0;
          sbi(PORTD,0);
        }
}
```

2. Sintaxis del lenguaje C

Para aquellos lectores que posean poca o ninguna experiencia en programación, se presenta un pequeño análisis de la sintaxis del lenguaje C. Este análisis va a estar conformado por las partes más básicas del lenguaje, que le permitan a los usuarios crear códigos simples para el control de microcontroladores AVR.

2.1 Comentarios

Los comentarios son cadenas de caracteres que son ignorados por el compilador. Como su nombre lo indica, son comentarios que están ahí para ayudar al programador o a quien lea el código a entender el funcionamiento de las partes que conforma el código. Existen dos estilos para los comentarios:

```
// Con doble slash. Esto es un comentario en estilo C++

/* Este es un comentario en estilo C */
/* Se inician con asterisco-slash y se terminan con slash-asterisco
*/
```

2.2 Variables

Definir una variable significa apartar una porción de la memoria RAM dentro del microcontrolador AVR que estamos programando y asignarle un nombre. Luego de definida, se le puede asignar un valor, que posteriormente podemos llamar, utilizando el nombre que le hayamos asignado a la variable. Una definición

de variable se realiza de la siguiente forma:

```
u08 myvar;   // entero de 8 bits sin signo llamado myvar
```

Donde `u08` es el tipo de datos que almacena y `myvar` es el nombre de la variable. Como se puede notar, la expresión finaliza con punto y coma. Esto es así con las declaraciones en lenguaje C.

Una variable debe ser declarada antes de ser utilizada, y antes de cualquier instrucción ejecutable de una función en donde sea definida.

Los tipos de variables enteros son:

```
u08 a;  // entero de 8 bits sin signo (0 a 255) ó 0 a MAX_U08

s08 b;  // entero de 8 bits con signo (-128 a 127)
// o MIN_S08 a MAX_S08

u16 c;  // entero de 16 bits sin signo (0 a 65535)
// o 0 a MAX_U16

s16 d;  // entero de 16 bits con signo (-32768 a 32767)
// o MIN_S16 a MAX_S16

u32 e;  // entero de 32 bits sin signo (0 a 4294967295)
// o 0 a MAX_U32


s32 f;  // entero de 32 bits con signo (-2147483648 a          //
2147483647) o MIN_S32 a MAX_S32
```

Los nombres `MAX_U08`, etcétera, son definiciones que están contenidas en la librería Avrlibc y que equivalen a los valores mayores y menores de cada tipo de entero.

Para asignar un valor a una variable se utiliza el signo de igualdad.

```
u08 foo, bar;   // define 2 enteros de 8 bits sin signos

foo = 12;       // asigna el valor 12 a la variable foo

bar = foo * 10; // asigna el producto 10 por el valor de la
//variable foo a la variable bar
```

2.3 Arreglos

Los arreglos son porciones de memoria de igual tamaño y adyacentes que permiten indexar valores de una forma conveniente.

```
u08 myarray[5];  // define un arreglo de 5 enteros de 8 bits sin
//signo

u08 myarray2[3] = {10, 12, 18};  // define un arreglo de 3 //enteros
de 8 bits sin signo, e inicializa el arreglo asignando //los valores
10, 12 y 18
```

Los arreglos se pueden acceder para asignarle valores y ser utilizados con expresiones con corchetes, donde se indica el índice del elemento del arreglo que queremos asignar o llamar. El índice del primer elemento de un arreglo es siempre 0.

```
myarray2[0] = 11;   // asigna el valor 11 al primer elemento del
//arreglo myarray2, el cual tiene indice 0

myarray2[1] = myarray[0]; // asigna el valor del primer elemento //de
myarray al segundo elemento de myarray2
```

2.4 Condicionales

La declaración condicional básica es `if.. else`.

```
if (a == 2) {          // si a es igual a 2
     foo++;            // incrementa la variable foo
     bar = foo + 10;   // asigna el valor de foo + 10 a bar
}
else {                 // sino (a no es igual a 2)
     foo--;            // disminuye la variable foo
     bar = foo - 10;   // asigna el valor de foo - 10 a bar
}
```

Se puede notar los operadores de incremento (`++`) y disminución (`--`) los cuales incrementan y disminuyen, respectivamente, en una unidad el valor de la variable.

En este condicional, luego de la palabra reservada `if` sigue una expresión entre paréntesis que evalúa un número. Si se cumple esta evaluación, se ejecutan las instrucciones entre las llaves inmediatamente después de la expresión. Si no se cumple, se ejecutan las instrucciones entre llaves luego de la palabra reservada `else`. No necesariamente luego de un `if` debe haber un `else`, ya que si sólo se quieren ejecutar ciertas instrucciones dado un sólo caso, no es necesario colocar más casos.

La expresión condicional usualmente evalúa mediante el uso de operadores de relación. Estos operadores devuelven el valor 1 si la relación se cumple y a 0 si no se cumple. Los operadores son los siguientes:

- Igualdad ==
- Desigualdad !=
- Menor que <
- Mayor que >
- Menor o igual que <=
- Mayor o igual que >=

Es importante recalcar que existe una diferencia entre = y ==. El primero es un operador de asignación, mientras que el segundo es un operador de relación. En otras palabras, uno asigna el valor, el otro compara los valores.

2.5 Lazos

Existen dos formas estructures de lazos en C: los lazos `while` y lazos `for`.

Los lazos `while` son bloques de código que son ejecutados repetidamente hasta que cierta condición se cumpla. Su estructura es la siguiente:

```
while(expresion){
     instrucciones
}
```

Como en un condicional, la expresión entre paréntesis es evaluada primero. Mientras dicha expresión no se cumpla, se ejecutan las instrucciones entre las llaves, y al finalizar esta instrucciones se vuelve a evaluar la expresión. Cuando la expresión se cumpla, sale del lazo y continúa la ejecución secuencial del resto del código. La expresión realiza una evaluación relacional dependiente de algún valor o variable que se actualiza cada vez que se ejecuta el lazo.

```
u08 button;
while (button != 1) {         //mientras el valor de la variable
//button no sea igual a 1

   button = checkButton(3); // llamada a la funcion checkButton con
argumento 3, y asigna su valor a la variable button
}
```

Este código se ejecuta hasta que la función de como respuesta 1. Este tipo de estructura es común para leer el valor de una entrada.

Una excepción para este tipo de estructuras es la siguiente:

```
while (1) {  // lazo sin fin
     ...
     ...
}
```

Este lazo sin fin se encuentra en prácticamente todos los proyectos AVR, ya que siempre se tiene alguna funcionalidad que se desea repetir continuamente.

Los lazos `for` son utilizados cuando se necesita repetir una acción un número específico de veces. Su estructura es la siguiente:

```
for (inicializacion; condicion; actualizacion) {
     instrucciones
}
```

La inicialización, condición y actualización son expresiones.

Normalmente, la inicialización y la actualización son asignaciones o llamadas a funciones, y la condición es una expresión relacional.

Un lazo `for` típico es el siguiente:

```
for (i=0; i<4; i++) {   // para i desde 0 a 3
     checkButtons(i);   // llamada a la funcion checkButtons con //
argumento i
}
```

En este código la inicialización asigna a la variable `i`, el contador del lazo, el valor 0. Esta instrucción es ejecutada una sola vez, antes de entrar en el lazo. La condición es una expresión relacional, mientras se cumpla se ejecutan las instrucciones dentro del lazo. Al finalizar el lazo, se ejecuta la actualización antes que la condición sea evaluada nuevamente, y se repite todo este proceso de evaluación de la condición, ejecución de las instrucciones y ejecución de la actualización hasta que la condición no se cumpla y sale del lazo.

Note que:

```
for (inicializacion; condicion; actualizacion) {
    instrucciones
}
```

es equivalente a:

```
inicializacion;
while(condicion){
    instrucciones
    actualizacion
}
```

2.6 Funciones

Una función posee la siguiente estructura:

```
tipo_del_dato_entregada Nombre_Funcion(tipo_arg1 nombre_arg1,
tipo_arg2 nombre_arg2, ... tipo_argN nombre_argN)
{
    declaraciones
    instrucciones
}
```

La definición de una función especifica lo que hace dicha función; esta definición se puede realizar en algunas partes del código. Puede ser definida luego de la función `main`, siempre que el prototipo de la función haya sido declarada con anterioridad. El prototipo contiene el nombre, el tipo de dato que entrega dicha función y los argumentos que recibe, pero no el cuerpo de la función. Aquí, las declaraciones se utilizan para los prototipos de función que declaren que la función existe.

Las definiciones se utilizan donde el cuerpo de la función es realmente especificado. Normalmente, un fichero .c viene acompañado de un fichero .h que contiene todos los prototipos de funciones. Con la librería AVRLibc, usualmente se

utilizan cierto número de ficheros .c (por ejemplo `timer.c` o `a2d.c`) que contienen definiciones de funciones. Estos ficheros .c son compilados conjuntamente con el código escrito, y son especificados en el `makefile`. Pero, para poder usar funciones en estos ficheros .c, en primer lugar se deben declarar sus prototipos. Esto se hace incluyendo los ficheros .h correspondientes (en los ejemplos anteriores serían los ficheros `timer.h` y `a2d.h`). Las funciones también pueden ser declaradas antes de la función `main`, pero esto se considera como mal estilo de programación.

Luego que una función es declarada, puede ser utilizada, o "llamada", siempre que la definición para la misma exista. Si la función no entrega un valor, el tipo del dato entregado será `void`.

`Void` también se utiliza si no existen argumentos para esa función. El programa de ejemplo al inicio de esta sección contiene una función de este tipo:

```
void checkButton(void){
     ...
}
```

El código dentro de la función se ejecuta cuando se "llama" a esta función desde otra función:

```
checkButton();
```

Si la función entrega un valor, la palabra reservada `return` debe estar presente en la función, seguida del valor que será entregado o devuelto. Una vez que esta palabra reservada se ejecuta, la función sale de su ciclo y evalúa al valor devuelto. Si la función tiene argumentos, usualmente se les envía a la función como valores entre paréntesis luego del nombre de la función en la llamada a la misma. Dentro de la función, los argumentos se comportan como variables que toman el valor que se les ha asignado. El siguiente es un ejemplo de esto :

```
int main(void){
     u08 bar;
     bar = funcionejemplo(10);
     ...
}
u08 funcionejemplo(u08 foo){
     return (foo * 10);
}
```

En este ejemplo, la variable `bar` termina almacenando el valor 100.

Note también que la función `main` normalmente tiene `int` como tipo de valor entregado. Esto es debido a una convención para lo que se quiere lograr al programar microcontroladores. Un valor devuelto de 0 indica que el programa a finalizado correctamente, y cualquier otros valores especifican diferentes errores. También cabe destacar que la función `main` no requiere un prototipo de función.

3. Comandos específicos de la librería AVRLibc

3.1 Registros AVR

Toda la información en el microcontrolador, desde la memoria de programa, la información del timer, hasta el estado de los pines de entrada y salida, se almacena en registros. Los registros son como los espacios de un librero, representando la memoria del procesador interno del microcontrolador. En un procesador de 8 bits, cada espacio puede almacenar hasta 8 libros, donde cada libro representa un bit en número binario, es decir, 0 o 1. Cada uno de estos bits posee una dirección en la memoria, para que el microcontrolador pueda ubicarlos con precisión.

Tomemos como ejemplo el ATMega 16. Los 32 pines de entrada/salida del ATMega 16 se dividen en 4 puertos (A,B,C y D). Cada puerto tiene asociado tres registros. Por ejemplo, para el puerto D, estos registros serían, en lenguaje C, PORTD, PIND y DDRD. Para el puerto B serían PORTB, PINB y DDRB, y así sucesivamente con los demás puertos. En lenguaje C, PORTD es en realidad una

macro, que se refiere a un número, siendo éste la dirección del registro en el microcontrolador. Esto se debe a que es mucho más sencillo recordar PORTD que un número hexadecimal arbitrario cada vez que se desea acceder a dicho registro.

3.1.1 Los registros DDR

El registro DDRD asigna la dirección del puerto D. Cada bit del registro DDRD asigna el pin correspondiente del puerto D como entrada o como salida, donde 0 asigna una entrada para el pin correspondiente, y 1 asigna una salida. Por ejemplo, para asignar el primer pin del puerto D como salida, se puede utilizar la función `sbi(reg,bit)`, que asigna un bit (lo coloca en alto o en 1 binario) en un registro:

```
sbi(DDRD, 0);   //Esta dos instrucciones son equivalentes
sbi(DDRD, PD0); //pin 0 del puerto D como salida
```

Para colocar el segundo pin como entrada, se tiene la función `cbi(reg,bit)` que limpia un bit (lo coloca en bajo o en 0 binario) en un registro:

```
cbi(DDRD, 1);   //Estas dos instrucciones son equivalentes
cbi(DDRD, PD1); //pin 1 del puerto D como entrada
```

Note que los índices de los bits empiezan en 0. Tenemos 8 bits por registros, enumerados del 0 al 7. Esto puede causar confusión al principio, porque cuando se dice “primer pin” de un puerto, nos referimos al bit 0 o P0 en lenguaje C.

También se pueden asignar valores a todos los bits de un registro en una sola instrucción. Para esto se utiliza la función `outb(reg,byte)`. Esta función escribe un byte (8 bits) en un registro específico. Por ejemplo, si se quiere asignar los pines 1 al 4 del puerto B como salidas y los pines 5 al 8 como entradas, se utiliza:

```
outb(DDRB, 0x0F); //Asigna los 4 primeros pines a 1
                  //y los 4 últimos a 0
```

3.1.2 Los registros PORT

Los registros PORT funcionan de manera diferente de acuerdo a si un pin esté definido como entrada o como salida. El caso más sencillo es cuando el pin está como salida. En ese caso, el registro PORTC, por ejemplo, controla el valor en los pines físicos del puerto C. Por ejemplo, se puede asignar todos los pines del puerto C como salidas y luego colocar 4 de estos en alto (1 binario), y los otros 4 en bajo (0 binario):

```
outb(DDRC, 0xFF);  //Todos los pines como salida
outb(PORTC, 0xF0); //Primeros 4 en alto y los otros en bajo
```

Cuando un pin está definido como entrada, el registro PORT no contiene valores lógicos aplicados a dicho pin. Para esto se utiliza el registro PIN. Si un pin es una entrada, y se le asigna un 1 con el registro PORT, se coloca una resistencia pull-up en dicho pin. Esto puede resultar útil para una variedad de circuitos.

```
outb(DDRC, 0x00); //Todos los pines del puerto C como entradas
outb(PORTC, 0xFF); //Resistencias pull-up en todos los pines
```

3.1.3 Los registros PIN

Cuando un pin está definido como entrada, el registro PIN contiene el valor aplicado a dicho pin. Los pines tienen un umbral eléctrico de alrededor de 2.5 voltios. Si un voltaje mayor a este nivel es aplicado a dicho pin, el bit correspondiente en el registro PIN será 1. Debajo de este nivel, el bit será 0.

Para leer los valores de un puerto de entrada, tenemos la función `inb(reg)`. Esta función devuelve un número de 8 bits que contiene el valor de los 8 bits del registro especificado.

```
outb(DDRD, 0x00);   // Puerto D como entrada
outb(PORTD, 0xFF);  // Resistencias pull-ups en puerto D
foo = inb(PIND);    // Lee el valor del registro PIND
                    // y lo almacena en la variable foo
```

También existe una función para revisar el estado de un bit específico, sin tener que leer todo los bits del registro PIN. La función `bit_is_set(reg,bit)` devuelve un 1 si el bit especificado está en 1, y devuelve 0 si dicho bit está en 0.

```
u08 bar;  // declaramos una variable de 8 bits

outb(DDRD, 0x00);    // Puerto D como entrada
outb(PORTD, 0xFF);   // Resistencias pull-ups en puerto D

bar = bit_is_set(PIND,1);
// bar contiene el valor del pin 1 del puerto D
```

3.2 Otros registros

Existen muchos otros registros. Se puede acceder a ellos a través de las funciones anteriormente descritas. Muchos dispositivos internos del microcontrolador, como los timers y los convertidos A/D, poseen registros asociados en la librería AVRLibc que facilita su programación. También existen algunos pocos registros de 16 bits, por lo que para éstos, se utilizan las funciones `outw(reg,word)` y `inw(reg)` para leer/escribir sobre éstos.

AVRLib es esencialmente una colección de funciones y macros que hacen más intuitivo y fácil el acceso a las funcionalidades de los microcontroladores AVR. Estas funciones tienen a tener nombres intuitivos, y siguen una convención para sus nombres de `nombreAcciónaRealizar()`, donde `nombre` es el nombre descriptivo, empezando con el nombre en minúscula de los ficheros .c y .h donde se contiene dicha función (por ejemplo, timer para las funciones del temporizados o a2d para las funciones del convertidor analógico-digital). La otra porción normalmente se refiere a lo que hace la función. La mayoría de estas funciones deben ser inicializadas con anterioridad a través de una función para esto, como por ejemplo `timerInit()`, `uartInit()` y `midiInit()`.