Aprendiendo a programar un microcontrolador

Ernesto M. Corbellini, Sebastián García Marra y Ariel Burman
Facultad de Ingeniería UBA

29 de marzo de 2013

Índice

1. ¿Qué es un Microcontrolador?

Un microcontrolador (abreviado μC, uC o MCU) es un circuito integrado que tiene la capacidad de interpretar, procesar y ejecutar un conjunto de instrucciones almacenadas en una memoria. Contiene tres bloques funcionales principales: unidad central de proceso (o CPU), memoria y periféricos de entrada y/o salida. El funcionamiento de un μC consiste en que el CPU lee las instrucciones almacenadas en la memoria y las procesa. Estas instrucciones pueden ser por ejemplo sumar dos números y guardar el resultado en una ubicación particular. Algunas de esas instrucciones requieren una interacción con el mundo exterior. Para este fin se encuentran los periféricos que son los dispositivos que le permiten al microcontrolador comunicarse con otros dispositivos o con el usuario. En una computadora (PC), los periféricos más claros son el teclado, el mouse y el monitor.

Es normal preguntarse ¿en qué se diferencia un microcontrolador de una computadora? ¡cualitativamente en nada! Para entender las diferencias vamos a describir un poco más los componentes.

1.1. Memoria

La memoria es uno de los dispositivos más importantes, tanto en el microcontrolador como en la computadora. Es el espacio físico donde se almacenan los datos. Existen varios tipos distintos de memorias, que da lugar a diversas clasificaciones.

La menos ambigua de todas las clasificaciones posibles es la dicotomía entre memorias volátiles y no volátiles. Una memoria no-volátil es aquella que retiene la información aún cuando no está conectada a una fuente de alimentación. Las memorias no-volátiles más comunes son las memorias EEPROM, Flash, o en una computadora, el disco rígido (almacenamiento magnético). Las memorias ROM (Read Only Memory) son memorias de "sólo-lectura". Esto significa que durante el funcionamiento normal del microcontrolador, el contenido de esta memoria no será modificado; la modificación de su contenido se hace en momentos particulares (por ejemplo en su fabricación) y su código permanece inalterado salvo que explícitamente se lo desee modificar, por lo tanto son un tipo de memorias no-volátiles.

En contraposición, las memorias volátiles son aquellas que su contenido sí se pierde al quitar la fuente de alimentación. Un claro ejemplo de la memoria volátil es el espacio donde se encuentran los documentos de texto mientras los editamos. Si por algún motivo se corta el suministro de energía eléctrica perderemos todo el trabajo realizado desde la última vez que guardamos el archivo.

Otra clasificación de las memorias son las memorias de acceso aleatorio o RAM (Random Access Memory) y las memorias de acceso secuencial. Las memorias RAM tienen la característica de poder acceder a cualquier dato almacenado en la memoria, con igual retardo. Tampoco hay diferencia (apreciables) entre los tiempos de lectura y escritura, es decir los tiempos que se requiere para escribir un dato en la memoria o recuperar un dato de ella (a menos que sea una memoria de sólo lectura). La memoria "RAM" de la computadora es llamada así justamente por ser una memoria de acceso aleatorio, además de tener otras propiedades, como ser una memoria volátil.

En cambio, en una memoria de acceso secuencial deberemos acceder a bloques de datos completos sólo para recuperar un dato que se encuentra en la mitad de cierto bloque. Las memorias SD o flash, como por ejemplo las que tiene un *pendrive*, son memorias secuenciales[1]. En nuestro uso diario no lo distinguimos porque el sistema operativo nos simplifica la tareas, pero en el más bajo nivel de programación existen rutinas que acceden en forma secuencial a la información.

¿En un microcontrolador qué memorias tenemos?

El conjunto de instrucciones que queremos que el μC procese se almacena en una memoria no-volátil ¿y

[1] Existen dos tipos de memoria flash: las NAND que son de acceso secuencial, y las NOR que son de acceso aleatorio. Las NOR son utilizadas en aplicaciones donde se requiere alta confiabilidad, como por ejemplo, aplicaciones espaciales, mientras que las NAND, si bien son menos confiables, son mucho más económicas y por lo tanto son las empleadas en dispositivos de almacenamiento masivo.

por qué? Simplemente porque no queremos tener que volver a cargar el programa cada vez que le sacamos la energía al microcontrolador. Este espacio de memoria se lo conoce como "memoria de programa" ya que ese conjunto de instrucciones es el programa que hemos diseñado para que sea ejecutado por el micro.

Como ya vimos, a diferencia de las memorias no-volátiles, las memorias volátiles sí pierden la información cuando se retira la alimentación. Tienen la ventaja de ser mucho más rápidas que las memorias no-volátiles, y por ese motivo son las que utiliza el microcontrolador (y las computadoras) para procesar información. Por ejemplo, cuando el μC está realizando alguna cuenta, requerirá un espacio donde almacenar temporalmente datos, para lo cual utilizará la memoria volátil. Cuando queremos guardar el resultado de esa cuenta para que esté disponible en algún otro momento (aún habiendo quitado temporalmente la alimentación), es necesario trasladarlo a una memoria no-volátil.

1.2. CPU

Como dijimos antes, la CPU o microprocesador es el encargado de interpretar y ejecutar las instrucciones almacenadas en la memoria. Dos características principales son la velocidad y la arquitectura.

La velocidad se refiere a la cantidad de instrucciones por segundo que procesa el microcontrolador. Estrictamente no es así, pero esta idea nos permite comparar a grandes rasgos distintos procesadores. Las CPU de los microcontroladores generalmente trabajan en el rango de 1-100 MHz. En el caso de los microcontroladores más avanzados (utilizados en celulares y tablets) la velocidad está cerca de 1 GHz. En cambio, los microprocesadores de PC actualmente trabajan entre 2 GHz y 4 GHz.[2]

Otra característica importante es el tamaño del bus de datos. Mientras que en las computadores actuales se encuentran procesadores de 32 bits o 64 bits, en el campo de los microcontroladores, las arquitecturas más comunes son de 8 bits y 32 bits. El tamaño del bus de datos indica básicamente cuál es el tamaño de cada dato que puede procesar el microcontrolador en una sola instrucción. Por ejemplo, un microcontrolador de 8 bits necesita sólo una instrucción para sumar dos números de 8 bits (cada dato es de 8 bits), pero si queremos sumar dos números de 32 bits, deberá realizar varias instrucciones que implicarán mayor tiempo de procesamiento y un programa más grande.

Podemos encontrar una analogía en nosotros mismos: cuando queremos sumar números pequeños lo hacemos en forma casi instantánea, como si nos requiriera una sola operación, mientras que si queremos sumar números más grandes, por ejemplo de 4 o 5 dígitos, recurrimos a varias operaciones mentales donde primero sumamos las unidades, luego las decenas, luego las centenas, etc.

Por suerte nosotros no deberemos conocer y resolver los detalles de cómo el microcontrolador realiza estas operaciones. Sólo debemos tener en cuenta que cuando queramos hacer muchas operaciones con números de 32 bits, en muy poco tiempo, nos interesará utilizar microcontroladores de 32 bits.

1.3. Periféricos

Los periféricos son todos aquellos dispositivos que interactúan con el microcontrolador. En una computadora, se considera periférico al teclado, el monitor, o la impresora. En un microcontrolador, los periféricos están relacionados a la interacción con otros dispositivos de hardware.

- I/O (*input/output*): algunos pines de un microcontrolador pueden utilizarse como periféricos de entrada y salida. Configurado como "entrada" permite enviarle información a un microcontrolador, por ejemplo por medio de un pulsador. Un pin configurado como salida se utiliza para accionar algún otro dispositivo, por ejemplo apagar y prender un LED.
- Timer: son módulos que pueden "contar" eventos. Si los eventos suceden cada cierto intervalo de tiempo fijo, nos permiten llevar un registro temporal. Por ejemplo, si queremos enviar un mensaje

[2] M (Mega) = 10^6 ; G (Giga) = 10^9 ; Hz (Hertz) = 1 ciclo/segundo

cada 5 segundos, necesitaremos un *timer* que cuente un timer nos permitirá ejecutar una misma tarea cada un intervalo.

- ADC (Analog to Digital Converter): las variables físicas son variables continuas también llamadas analógicas, mientras que el microcontrolador sólo puede trabajar con números "discretos" basados en un sistema binario. Son discretos porque podemos identificarlos uno a uno. Esto no necesariamente implica que son números enteros. El ADC convierte una señal eléctrica (tensión) en un número binario, discreto, cuyo valor es proporcional a dicha señal. Por ser un número binario, combinación de "1" y "0", se lo considera digital.

- DAC (Digital to Analog Converter): cuando tenemos un número almacenado en la memoria del microcontrolador, que representa en forma proporcional una variable eléctrica, por medio de un DAC podemos convertir ese número (discreto) en un valor de tensión analógico.

- PWM (modulación por ancho de pulso o Pulse Width Modulation): es técnica que permite obtener cualquier valor entre 0 y 1, utilizando sólo dos estados.

Más adelante veremos algunos ejemplos donde utilizaremos estos periféricos.

2. Preparación de nuestra estación de trabajo

El diseño de un programa requiere varias herramientas. Primero debemos crear el código fuente, esto es, el conjunto de funciones y comandos que definen el comportamiento de nuestro programa. En general estas funciones deben ser escritas en un lenguaje determinado. Existen muchos lenguajes, cada uno asociada a diferentes paradigmas de programación. Hoy en día, gran parte del software desarrollado para microcontroladores está escrito en `C` o `C++`. Nosotros vamos a utilizar el lenguaje `C`.

Una vez que tengamos el código fuente, debemos transformar ese código en un conjunto de instrucciones aceptadas y reconocidas por el microcontrolador. Ese conjunto de instrucciones también está expresado en un lenguaje, conocido como lenguaje *assembler* o ensamblador.

La diferencia entre el lenguaje `C` y assembler es que cada microcontrolador tiene su propio conjunto de instrucciones, con nombres distintos y el código assembler generado para un microcontrolador no sirve para otro diferente. En cambio, el código generado en `C` (bajo ciertas condiciones) sí es reutilizable para diferentes microcontroladores. El lenguaje `C` nos permite, hasta cierto punto, abstraernos del microcontrolador que estamos utilizando. Por ese motivo, en este contexto, al lenguaje `C` se lo categoriza como un lenguaje de "alto nivel", mientras que al assembler se lo categoriza de "bajo nivel"[3].

Por lo tanto, necesitamos una herramienta que pueda entender el código fuente escrito por nosotros en `C` y genere el conjunto de instrucciones especiales para nuestro microcontrolador, es decir, el código assembler. Esta herramienta se llama compilador[4]. Por lo tanto, antes de poder realizar nuestro primer programa debemos instalar el compilador en nuestra computadora. Al conjunto de herramientas, dentro de las cuales se encuentra el compilador, que se utiliza para generar el código para el microcontrolador, se lo conoce como *toolchain*.

2.1. Instalando el *toolchain* en Windows

Atmel, la empresa que fabrica el microcontrolador que utilizaremos, proporciona una versión actualizada del toolchain en la siguiente dirección

`http://www.atmel.com/tools/ATMELAVRTOOLCHAINFORWINDOWS.aspx`

[3]En el contexto de la programación para PC el lenguaje C es considerado de bajo nivel, mientras que otros lenguajes como Java o Python, que ofrecen mayor nivel de abstracción, son los que se consideran de alto nivel.

[4]Estrictamente, se necesita de un compilador y un ensamblador para generar el conjunto de instrucciones mencionado

Ahí debemos descargar el archivo indicado en la figura 1

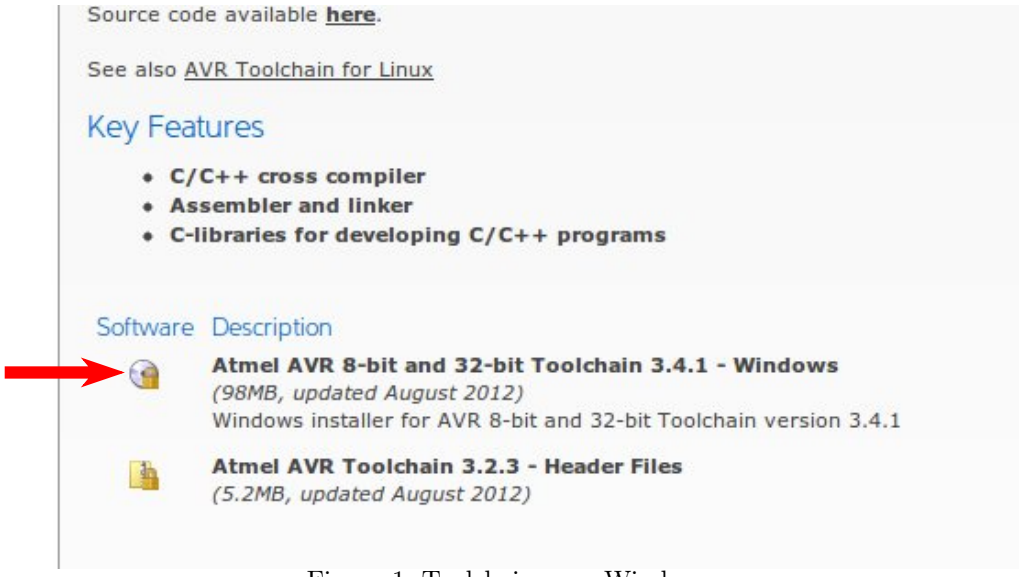

Figura 1: Toolchain para Windows

Una vez descargado, ejecutamos el instalador. Podemos desactivar la instalación de las herramientas para microcontroladores de 32 bits, quitando la tilde en el ítem correspondiente, como se muestra en la figura 2, ya que no vamos a trabajar con esta familia de microcontroladores.

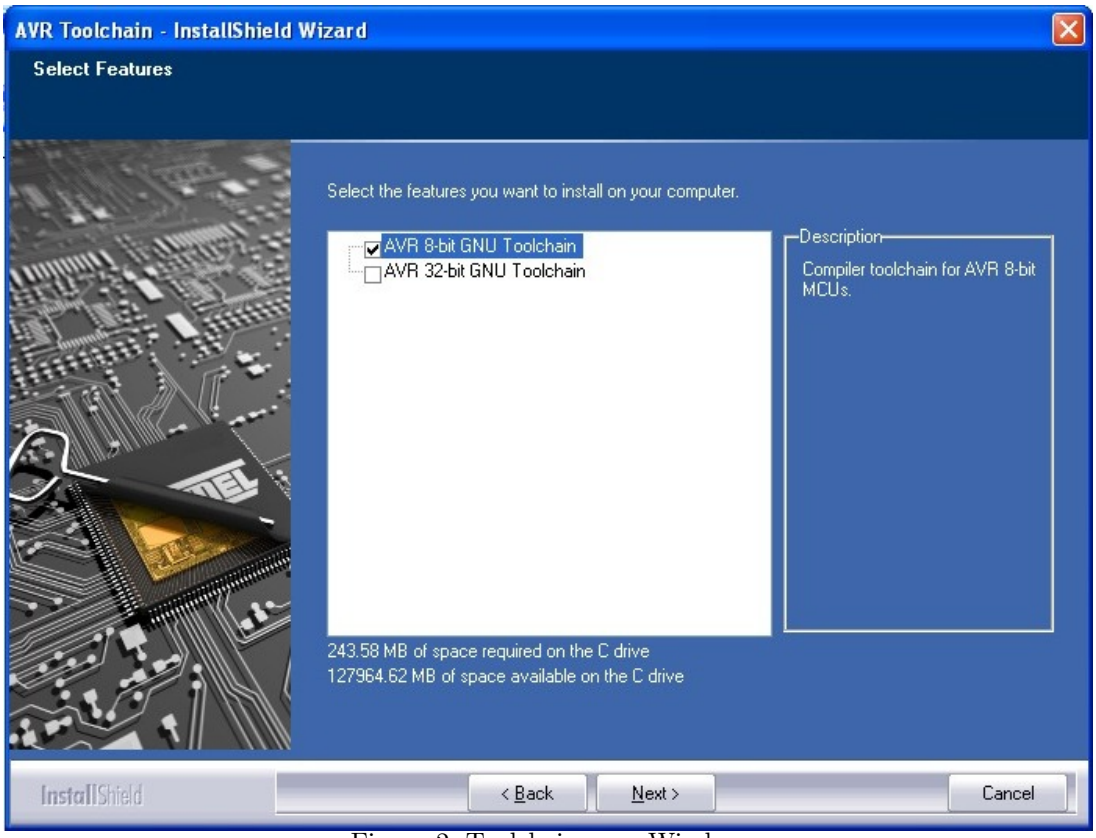

Figura 2: Toolchain para Windows

Una vez que finalice el proceso, ya habremos instalado el toolchain.

2.2. Instalando el toolchain en GNU/Linux

En este caso deberemos instalar los paquetes `gcc-avr avr-libc build-essential avrdude`. En una distribución Ubuntu (que es probablemente la más popular) ejecutamos, desde una terminal la siguiente linea:

```
sudo apt-get install gcc-avr avr-libc build-essential avrdude
```

(Parece mucho más sencilla la instalación en Linux que la instalación en Windows ¿no? :))

2.3. Verificando el correcto funcionamiento del toolchain

Sugerimos crear una carpeta donde guardaremos todos los programas que armaremos. Dentro de esta carpeta, sugerimos crear una carpeta por cada proyecto. En este caso creamos una carpeta para armar un proyecto de prueba.

Para hacer esto vamos a abrir una terminal y nos vamos a posicionar en el directorio donde hayamos guardado el archivo `main.c`. En Windows podemos abrir una terminal con `Super+E` y ejecutamos el comando `cmd` (la tecla `Super` es la tecla con el símbolo de Windows). Luego ejecutamos las siguientes lineas.

```
mkdir proyectos-iie
cd proyectos-iie
mkdir proyecto-de-prueba
```

El comando `mkdir` crea un nuevo directorio con el nombre que se le indique[5]. Por supuesto, las carpetas las podemos armar utilizando el explorador de archivos, pero muchas veces, en diferentes situaciones, no disponemos de un entorno gráfico con lo cual, poder manejarse con fluidez en una terminal, es una habilidad valiosa que en el futuro podremos necesitar.

Para comprobar el funcionamiento del toolchain vamos a "compilar" (generar) un pequeño programa. Los archivos de código fuente son archivos de texto plano (es decir, archivos de texto sin ningún formato). Por lo tanto, los archivos de código fuente pueden crearse y editarse con cualquier editor de texto plano, como el Bloc de notas, Gedit, o Vim, pero no con Microsoft Word u Open Office.

Abrimos entonces un editor de texto plano, copiamos el siguiente código, y guardamos el archivo con el nombre `main.c`, en el directorio que creamos anteriormente.

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

void  main(void){
  DDRC = 0x08;
  PORTC = 0xF7;

  while(1){
  }
}
```

[5]Para utilizar un nombre de directorio con espacios, se debe encerrar el nombre con comillas dobles. Ej: mkdir "nombre con espacio"

El código anterior no realiza ninguna tarea interesante (más allá de prender un LED en la placa de desarrollo), pero nos permitirá determinar si el compilador se instaló correctamente (y si soldamos bien los LEDs en la placa). Además de ser muy poco útil, el código es totalmente críptico. Pero a no preocuparnos, más adelante veremos con detalle cómo armar programas.

Vamos a abrir una terminal (o utilizar la que abrimos anteriormente si todavía no la cerramos) y nos vamos a posicionar en el directorio donde hayamos guardado el archivo `main.c`. Una vez ahí, vamos a ejecutar la siguiente linea:

```
avr-gcc -mmcu=atmega88 main.c -o main.elf
```

El primer comando `avr-gcc` es la invocación al compilador. Los siguientes parámetros son las opciones que queremos que el compilador utilice.

- `-mmcu=atmega88`: le indica al compilador para qué micro debe traducir el código. Si no agregamos esta opción, el compilador nos dará un error indicándonos que no están definidas ciertas opciones.
- `main.c`: es el archivo en código `C` que queremos traducir al lenguaje assembler.
- `-o main.elf`: le indica al compilador que el archivo de salida sea `main.elf`

El archivo generado, `main.elf`, contiene el programa que nosotros queremos que utilice el microcontrolador pero también incluye información que permite la depuración del programa. Para eliminar toda esta información de depuración, que no necesita ser utilizada por el microcontrolador, debemos ejecutar la siguiente linea:

```
avr-objcopy -j .text -j .data -O ihex main.elf main.hex
```

donde `main.hex` es el archivo que debemos cargar en el microcontrolador.

Para evitar tener que realizar todos estos pasos cada vez que queremos compilar nuestro programa, se utiliza un *script*. Un script es una secuencia de comandos agrupados en un solo archivo. El archivo suele llamarse *Makefile* (sin extensión) y se ejecuta a través del comando `make`. Más adelante veremos cómo utilizar el archivo Makefile para diferentes tareas.

2.4. Cargando el programa

Además de que cada microcontrolador tiene su propio conjunto de instrucciones, también tiene su propia secuencia de programación, lo cual hace que para poder cargarle el programa que generamos, debemos usar herramientas adaptadas para este microcontrolador.

Para esto vamos a utilizar dos herramientas. La primera es el programador (figura 3) que es el encargado de comunicarse con la computadora y generar la secuencia de programación particular para el micro que utilizamos. La segunda es el comando `avrdude` que es el programa encargado de enviarle el archivo `main.hex` al programador.

Figura 3: Programador del Club de Robótica[1], derivado del proyecto USBtinyISP[2]

2.4.1. Configuración en Windows

Como cualquier dispositivo nuevo, deberemos instalar los drivers para Windows del programador. Estos los podemos descargar de la página principal del proyecto USBtinyISP[2]

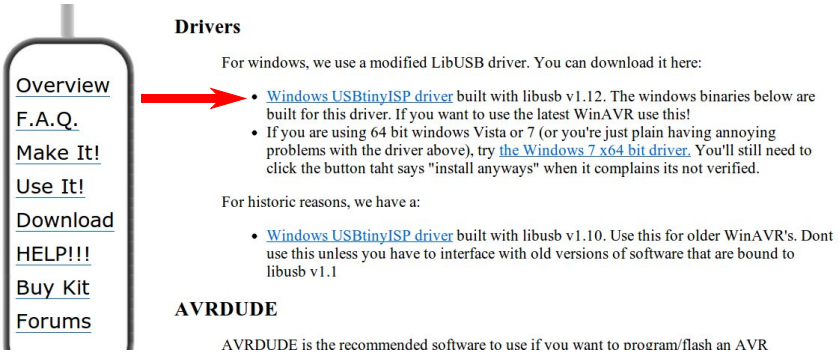

Figura 4: Descarga de drivers para Windows

Una vez descargados los drivers, los descomprimimos en una ubicación conocida, y cuando conectamos el programador nos va a pedir que le indiquemos la ubicación de esos drivers. A continuación observamos los pasos para instalar los drivers del programador en un sistema con Windows XP.

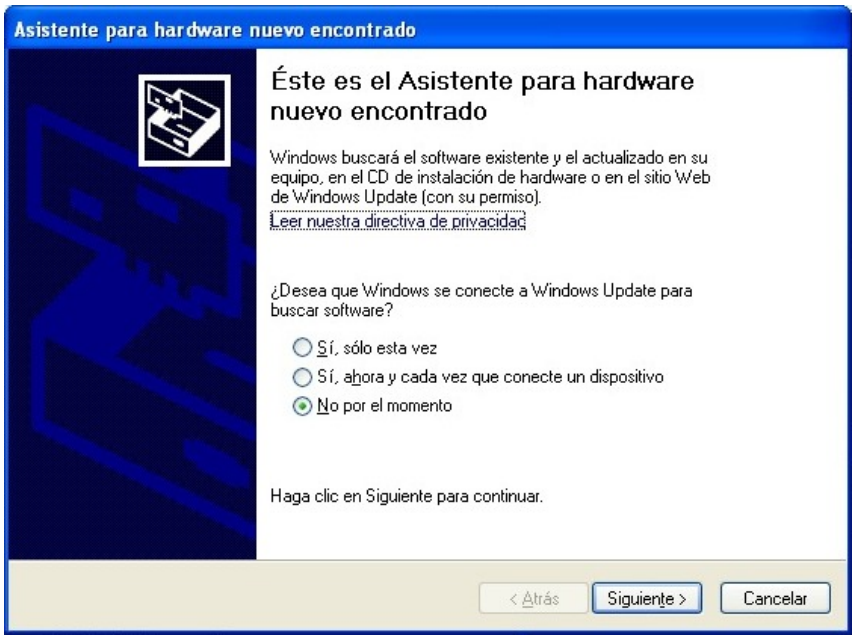

Figura 5: Instalación de drivers para Windows

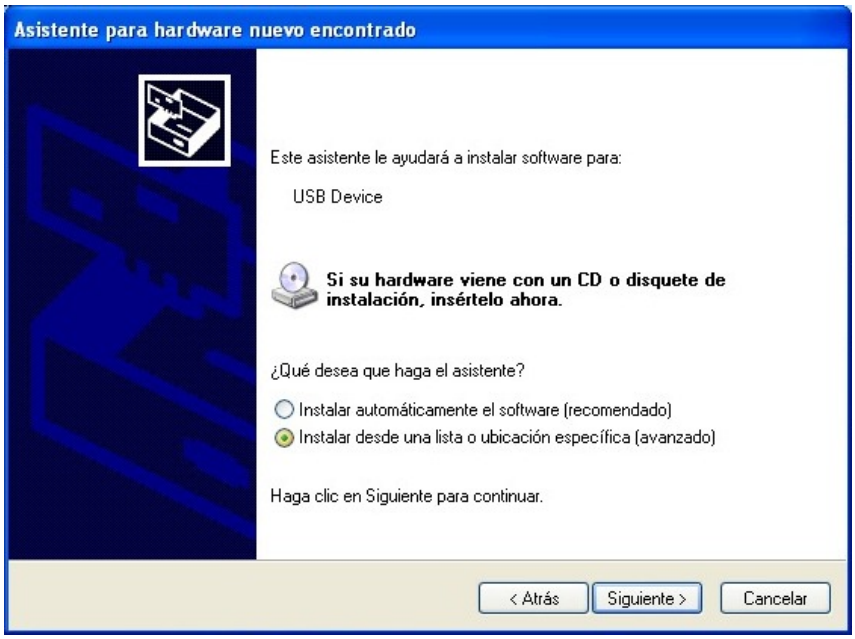

Figura 6: Instalación de drivers para Windows

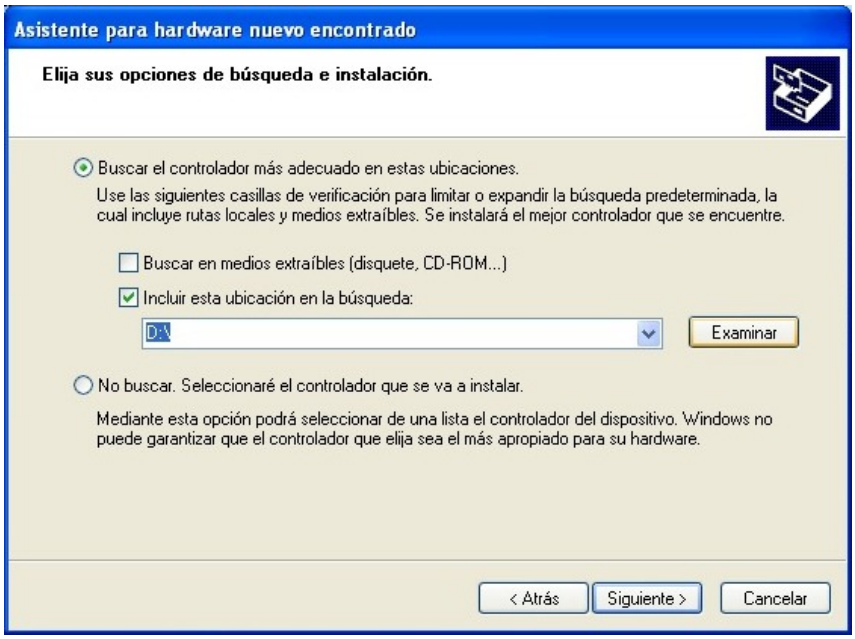

Figura 7: Instalación de drivers para Windows

Figura 8: Instalación de drivers para Windows

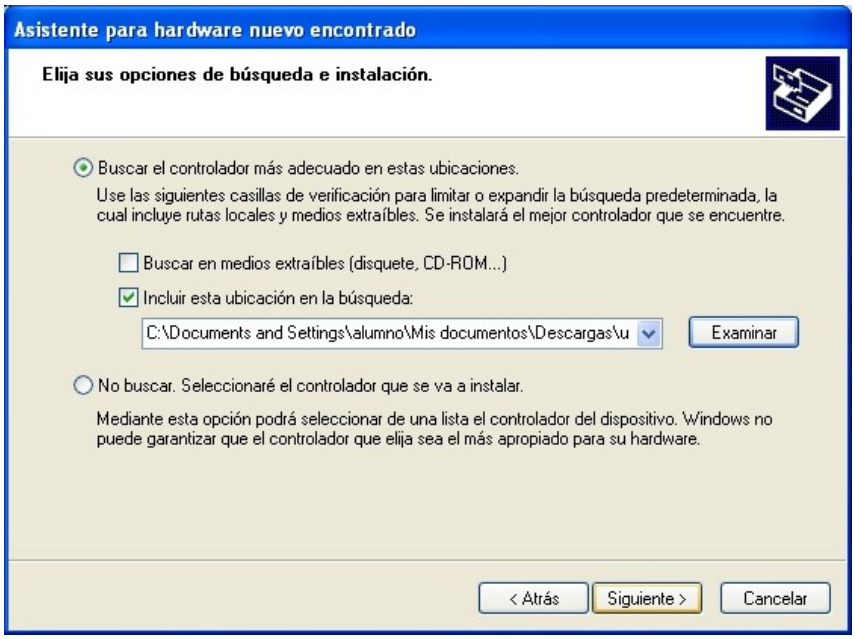

Figura 9: Instalación de drivers para Windows

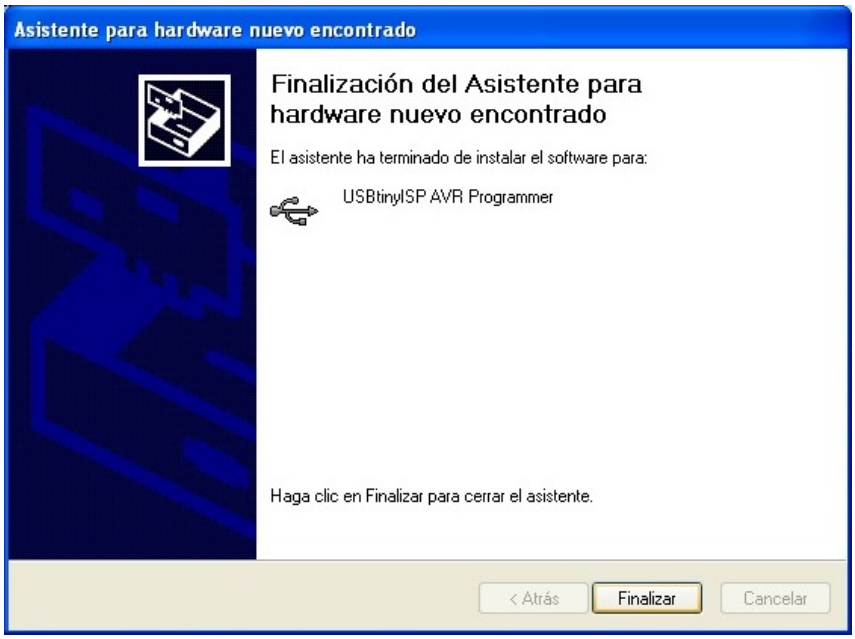

Figura 10: Instalación de drivers para Windows

En Windows Vista y Windows 7 no parece haber mayores complicaciones, solo que puede aparecer un aviso de que los controladores no están firmados y requerirá que autoricemos la instalación superando varios avisos sucesivos (es muy común en Windows que nos aparezcan 3 o 4 ventanas pidiéndonos autorización para hacer una sola tarea).

En Windows 8 parece ser necesario superar aún más barreras

Debemos abrir una terminal y ejecutar las siguientes lineas

```
bcdedit -set loadoptions DISABLE_INTEGRITY_CHECKS
bcdedit -set TESTSIGNING ON
```

luego instalamos el driver y finalmente retornamos el sistema al estado inicial

```
bcdedit -set loadoptions ENABLE_INTEGRITY_CHECKS
bcdedit -set TESTSIGNING OFF
```

2.4.2. Configuración en Linux

En Linux no debemos instalar ningún driver (que fácil que es trabajar en Linux ¿no? :))

Debido a que en un sistema GNU/Linux los dispositivos no pueden interactuar con el sistema a menos que nosotros lo permitamos (es bueno saber que tenemos control sobre lo que sucede en nuestro Sistema Operativo), debemos explícitamente indicar que queremos usar el programador.

Para esto debemos crear el archivo

```
sudo gedit /etc/udev/rules.d/70-usbtiny.rules
```

Con el siguiente contenido

```
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="1781",
ATTRS{idProduct}=="0c9f", GROUP="plugdev", MODE="0660"
```

Luego reiniciamos el servicio `udev` para que cargue las nuevas opciones.

```
sudo service udev restart
```

En todos estos pasos hemos utilizado el comando `sudo`. Esto es porque le indicamos al sistema operativo que queremos realizar una acción para la cual necesitamos permisos especiales.

Nunca debemos usar `sudo` (ni debemos ingresar nuestra contraseña) para hacer cualquier otra cosa que no sea una tarea de configuración y/o administración. No debemos usar `sudo` durante el uso periódico del microcontrolador.

2.5. Comprobando la comunicación

Al conectar el programador lo primero que debemos notar es que se enciende el LED verde, el cual se puede apreciar en la figura 11).

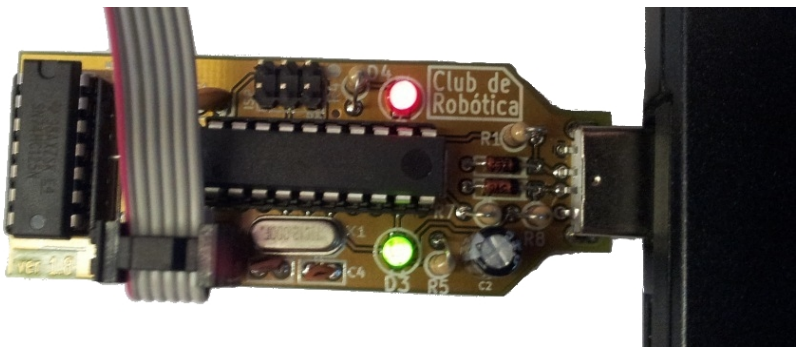

Figura 11: Programador del Club de Robótica, LED de encendido.

El LED rojo, que también se observa en la figura 11) sólo se encenderá mientras el programador se comunique con el microcontrolador (es decir, cuando queramos cargar nuestro programa). Para hacer esto, ejecutaremos el siguiente comando en una terminal.

```
avrdude -c usbtiny -p m88 -U flash:w:main.hex
```

En este caso, estaremos cargando el archivo `main.hex` (que habíamos generado previamente) en el microcontrolador. Sí todo salió bien, obtendremos un mensaje de programación exitosa, además de que debería encenderse el LED1 de la Placa de Desarrollo.

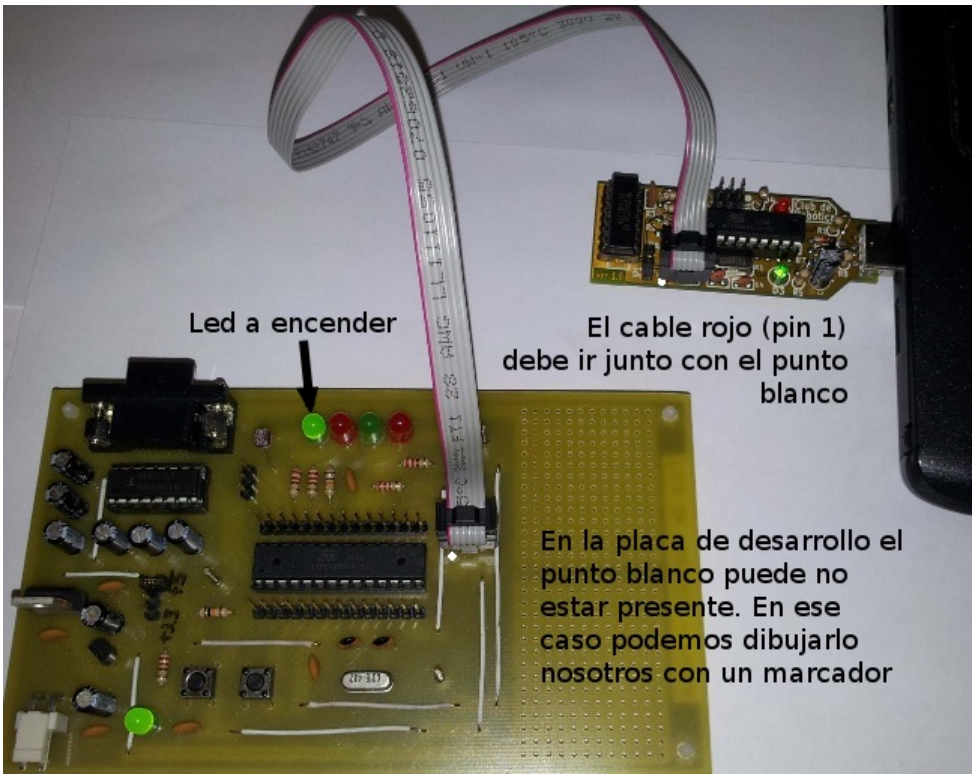

Figura 12: El LED1 de la placa de desarrollo encendido

Más allá de ser muy poco útil, ¡¡¡¡Hemos encendido un LED!!!! Aunque quizás no entendemos nada del código que hicimos ... por ahora.

¿Y qué hacemos si el LED no encendió? ¿Si no pudimos llegar hasta el último paso porque nos trabamos antes? Lo primero que debemos saber es que prácticamente *siempre* los proyectos fallan en el primer intento. ¡A no despesperar! Tendremos que ir revisando paso a paso qué es lo que pudo haber fallado, tratando de identificar primero en qué paso puede estar el problema, y una vez identificado, buscar cómo resolverlo.

2.6. *Troubleshooting*

La sección que nunca debe faltar en un tutorial es el lugar donde dicen qué hacer cuando no anda lo que dicen que había que hacer.

Este tutorial es tan bueno que hasta ahora no se han encontrado fallas ... es lo que dicen todos, pero lo cierto que es que si la sección "Troubleshooting" está vacía es simplemente porque nadie utilizó el tutorial para nada y por eso no se encontraron fallas.

Sistemas en los que se ha reportado una instalación exitosa:

- Windows XP
- Ubuntu 10.04
- Ubuntu 10.10
- Ubuntu 12.04

Si tenés un sistema operativo distinto de estos, y lograste realizar todos los pasos, por favor hacénoslo saber. Por otro lado, si tuviste algún problema, o tuviste que hacer alguna modificación a los pasos indicados en este tutorial, por favor también hacénoslo saber.

3. Comenzando a programar

Una vez que tenemos nuestra estación de trabajo lista para programar, vamos a empezar a armar unos pequeños programas.

3.1. Entorno de desarrollo

Existen entornos de desarrollo que se los conoce como IDE (por su sigla en inglés provenientes de *Integrated Development Environment*) que ayudan a organizar un proyecto.

Nosotros vamos a comenzar editando archivos de texto, pero más adelante podemos intentar trabajar con un IDE. Para aquellos que no conocen ninguna IDE o no tienen preferencia por alguna en particular, sugerimos arrancar con `CodeBlocks`.

3.2. Bibliotecas

Para comenzar a trabajar hay varios enfoques que se pueden utilizar. El más sencillo es utilizar una biblioteca con varias funciones pre-armadas. Hoy en día, la más popular es la biblioteca Arduino[5].

Lo que hay que tener en cuenta es que siempre hay ventajas y desventajas en utilizar una biblioteca. La principal ventaja es que ganamos facilidad de armado de programas, mientras que la principal desventaja es que perdemos la flexibilidad de usar el microcontrolador en su máxima versatilidad.

En particular, hemos armado una biblioteca muy reducida que apunta a poder comenzar a programar en forma simple.

3.3. Encendiendo un LED

Próximamente

4. Referencias

[1] Club de Robótica - Facultad de Ingeniería - U.B.A.

[2] Programador USBtinyISP

[3] Tutorial NewbieHack

[4] Hoja de datos del ATmega88

[5] Arduino