ATMEL tinyAVR

How to program the “tiny” Atmel microcontrollers

Guide on how to use the Atmel microcontrollers of the tinyAVR family with the Arduino IDE 1.6.7

Written by Leonardo Miliani

Version 1.6.7-1

Guide index:

1. **A little introduction to the MCUs**
   1. Attiny25/45/85
   2. Attiny24/44/84
   3. 3. Attiny2313/4313
   4. Notes

2. **How to setup the Arduino IDE 1.6.7**

3. **Wirings**

4. **Programming an ATtiny85**
   1. 1. Example sketch: blinking of 3 LEDs

5. **Serial communication**
   1. 1. Example sketch: data exchange between an Attiny and Arduino through serial

6. **Using the I2C**
   1. 1. Example sketch: read and write from/to an EEPROM through I2C

7. **Using the SPI**

8. **Conclusions and credits**

9. **Licenses and warranties**

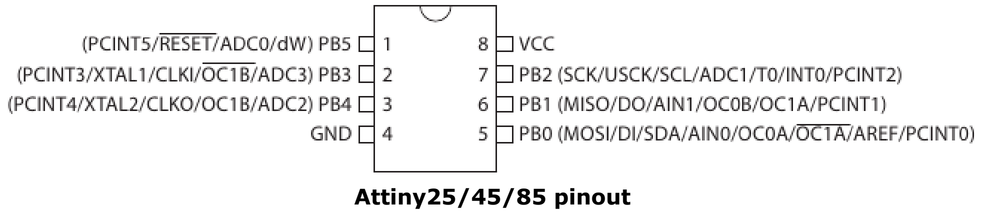
1. A little introduction to the MCUs

Apart the more famous ATmega328P used on the Arduino UNO boards, the ATmega2560 used on the Arduino MEGA2560, the ATmega32U4 of the Arduino Leonardo/Micro/Esplora/Yun and some other chips used in older or less known Arduino boards, Atmel also produces a series of 8-bits microcontrollers with limited dimensions, the **tinyAVR** family ("tiny" because in fact they are tiny...), sold in several packages, included the DIL package, mostly used in the hobbyist market. They are ideal for using in those projects where the number of I/O lines is not fundamental, the space is a primary requirement, low power devices are essential, without giving up the computational power of the megaAVR chips: in fact, the tinyAVR MCUs can work up to 20 MHz (with external crystals). But the chips can run at 1 MHz clock too and, in stand-by, achieve current consumptions of only 200 nA (nanoAmpere).

There are a lot of different Tiny chips but only few of them can be programmed using the Arduino IDE: to achieve this, the IDE needs a particular set of libraries that not only let the user to acces their peripherals with the code but that also gives the IDE the instruments to be able to program them. In the Arduino environment, such set of libraries is called "a core" and, for the Tiny MCUs, the most known is the "Tiny Core". Although it doesn't support a wide variety of different chips, it offers a good compatibility with the IDE, that is you can use the typical Arduino functions like digitalWrite, analogRead, pinMode, etc., with the MCUs the core supports. Currently, the Tiny Core supports the following chips:

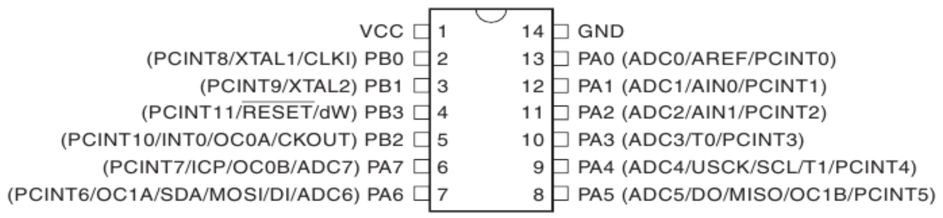
- **Attiny25/45/85**: very small microcontrollers (DIL8 package), with max. 6 I/O lines (5 "normal" pins plus the reset pin that can be activated like another I/O line)
  - Flash: 2/4/8 KBs, respectively
  - SRAM: 128/256/512 bytes, respectively
  - EEPROM: 128/256/512 bytes, respectively
  - Timers: 1x 8-bits timer, 1x 16-bits timer
  - 4-channels, 10-bits ADC
  - Clock: 1/8 MHz with internal oscillator, 16 Mhz with internal PLL, up to 20 Mhz with external crystal
  - Communication: SPI, USI
  - Not supported by HW: USART, I2C

Atmel ATtny85


(PCINT5/RESET/ADC0/dW) PB5 1
(PCINT3/XTAL1/CLKI/OC1B/ADC3) PB3 2
(PCINT4/XTAL2/CLKO/OC1B/ADC2) PB4 3
GND 4
8 VCC
7 PB2 (SCK/USCK/SCL/ADC1/T0/INT0/PCINT2)
6 PB1 (MISO/DO/AIN1/OC0B/OC1A/PCINT1)
5 PB0 (MOSI/DI/SDA/AIN0/OC0A/OC1A/AREF/PCINT0)
Attiny25/45/85 pinout

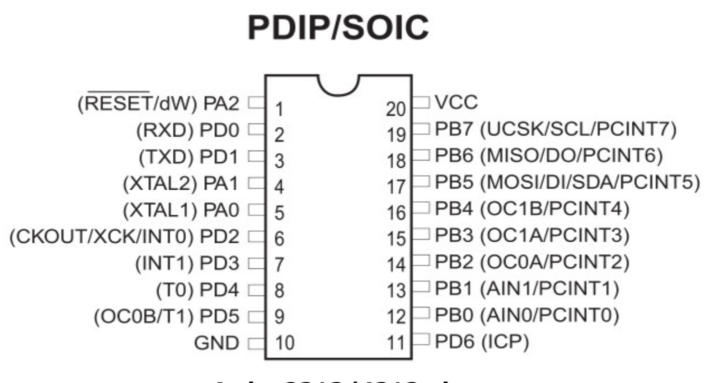
- **Attiny24/44/84**: small/medium-size microcontrollers (DIL14 package), with a max. of 12 I/O lines (11 “normal” pins plus the reset pin)
    - Flash: 2/4/8 Kbs, respectively
    - SRAM: 128/256/512 bytes, respectively
    - EEPROM: 128/256/512 bytes, respectively
    - Timers: 1x 8-bits timer, 1x 16-bits timer
    - 8-channels, 10-bits ADC
    - Clock: 1/8 Mhz with internal oscillator, up to 20 Mhz with external crystal
    - Communication: SPI/USI
    - Not available in HW: USART, I2C

Atmel ATtiny84


VCC 1
(PCINT8/XTAL1/CLKI) PB0 2
(PCINT9/XTAL2) PB1 3
(PCINT11/$\overline{\text{RESET}}$/dW) PB3 4
(PCINT10/INT0/OC0A/CKOUT) PB2 5
(PCINT7/ICP/OC0B/ADC7) PA7 6
(PCINT6/OC1A/SDA/MOSI/DI/ADC6) PA6 7
14 GND
13 PA0 (ADC0/AREF/PCINT0)
12 PA1 (ADC1/AIN0/PCINT1)
11 PA2 (ADC2/AIN1/PCINT2)
10 PA3 (ADC3/T0/PCINT3)
9 PA4 (ADC4/USCK/SCL/T1/PCINT4)
8 PA5 (ADC5/DO/MISO/OC1B/PCINT5)
Attiny24/44/84 pinout

- **Attiny2313/4313**: medium-size microcontrollers (DIL-20 package), with a max. of 18 I/O lines (17 “normal” pins plus 1 reset pin)
    - Flash: 2/4 kB, respectively
    - SRAM: 128/256 bytes, respectively
    - EEPROM: 128/256 bytes, respectively
    - Timers: 1x 8-bits timer,1x 16-bits timer
    - Clock: 1/8 Mhz with internal oscillator, up to 20 Mhz with external crystal
    - Communication: USART, SPI
    - Not available in HW: I2C, ADC

Atmel ATtiny2313



Attiny2313/4313 pinout

Note #1:
the Attiny 25/45/85 microcontrollers are identical each other and differ only for the size of their memories. Same is for the Attiny24/44/84 and for the Attiny2313/4313.

Note #2:
this guide shows some wiring and programming examples with an Attiny85. The same informations reported on this document are adaptable to the other microcontrollers cited above, too: some minor changes have to be done to adapt the example and the code (i.e., the pins used and their declaration in the code).

Note #3:
this guide has been written on a Mac with OS X 10.11, so the file paths, the terminal commands and such things refer to this system, Where possible, I've tried to adapt them for Windows and Linux systems too, like the commands to type on the terminal/command-line. If not present, the modifications should be simple and easily to do by the user for the system in use.

Note #4:
this guide applies to version 1.6.7 of the Arduino IDE and the changes are optimized for this version and won't work with previous releases of the software: if you want to use an older version of this branch, you should use a previous release of this guide, while the 1.0 branch is supported by another guide. Both of them are available for download from my personal website.

2. How to set up the Arduino IDE 1.6.7

As I said, the tinyAVR microcontrollers seen in the previous chapter are not natively supported by the Arduino IDE so we need to make some changes to the IDE so that you can write code and program them through the Arduino environment. To do this, we need the "core" for such chips so that we can use the Arduino functions with ease.

To install the core, we need to do the following steps:

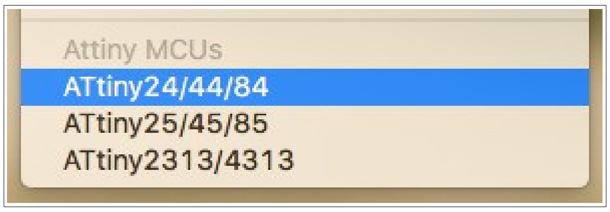
1. close the IDE. It's important that you apply the modifications while the Arduino environment is not running.

2. Now navigate to the folder used to save your sketches: if you don't know where in your computer this folder is, open the IDE then select "*File/Preferences*" from the menu: the first line that you'll see in the window that will appear on the desktop contains the path of the folders of your sketches (remember to close again the IDE before to copy the folder). Check that a folder named `/hardware` is already present: if not, create it.

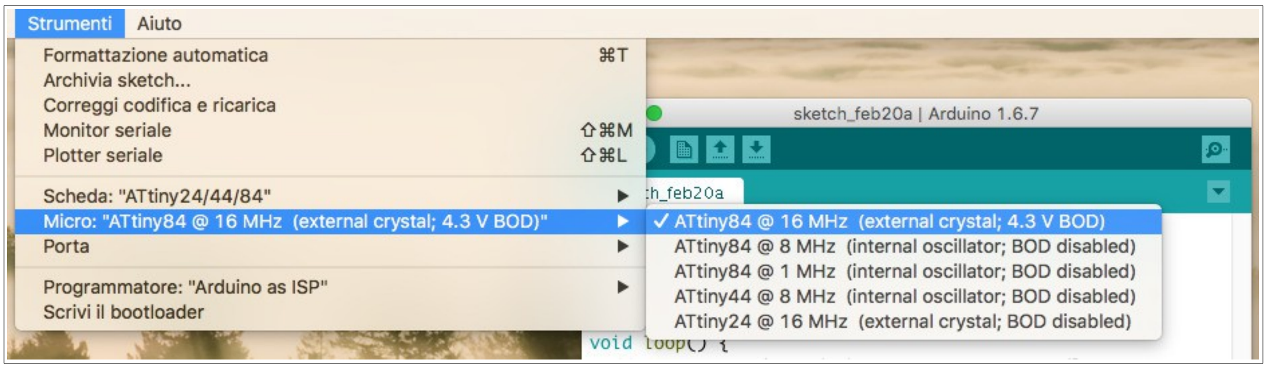
3. Now you need to install the core of this guide. You only need to copy the folder named `/tiny` inside the folder you reached at the previous step.

4. After this, check that inside the folder `/tiny/avr` you have the following files:
   1. `boards.txt`: this contains the definitions for the new microcontrollers. You'll find this file only if you have used the attached core, otherwise you should find a file named `Prospective boards.txt`, that contains all the variants of the chip that are supported by the core. You have to edit it according to your need and save it with the name `boards.txt`. Be careful: the attached file has been edited in a manner that will present you the MCUs grouped by family: in the rest of the guide I'll use this kind of sort. If you create your own file, keep in mind that you'll find some differences in the menus;
   2. `platform.txt`: this contains informations used by the IDE to upload the sketchs and to program the microcontrollers' fuses;
   3. avrdude.conf: this is the file that is used by the program named avrdude that flashes the sketches into the memory of the microcontrollers. The version that you find in this guide fixes a bug of the one bundled with the IDE;
   4. folders `/bootloaders`, `/cores`, `/libraries`: they contain the files to program to support and program the chips;
   5. `README`, `license.txt`: info files.
5. Now copy the folder `/ATtiny` inside your sketchbooks' folder (see #2).

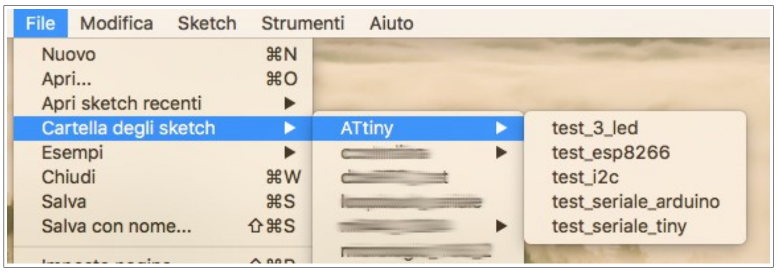
OK, now open again the Arduino IDE. If you correctly found all of the previous steps you should see under the "Tools/Board" four new entries (in the pictures below the names are in Italian, my native language):



Now, select a line of your choise, i.e. "*Attiny 24/44/84*", then open again the menu "*Tools*": you'll see a new entry, "*Micro*", from where you can choose the MCU of the family previously selected:


Strumenti
Aiuto
Formattazione automatica ⌘T
Archivia sketch...
Correggi codifica e ricarica
Monitor seriale ⇧⌘M
Plotter seriale ⇧⌘L
Scheda: "ATtiny24/44/84"
Micro: "ATtiny84 @ 16 MHz (external crystal; 4.3 V BOD)"
Porta
Programmatore: "Arduino as ISP"
Scrivi il bootloader
sketch_feb20a | Arduino 1.6.7
✓ ATtiny84 @ 16 MHz (external crystal; 4.3 V BOD)
ATtiny84 @ 8 MHz (internal oscillator; BOD disabled)
ATtiny84 @ 1 MHz (internal oscillator; BOD disabled)
ATtiny44 @ 8 MHz (internal oscillator; BOD disabled)
ATtiny24 @ 16 MHz (external crystal; BOD disabled)
void loop() {

Now check if the example sketchs are available too. Choose the entry "*File/Sketchbook*": you should find a nee sub-menu named "*Attiny*" with other entries inside, like in the following picture:



3. Wirings

To be able to program a microcontroller from the Attiny family we need to use the ISP (In-System Programming) communication because these chips don't reverse a memory area to store a bootloader like the ATmega328P of the Arduino UNO board nor they have the hardware support for the serial communication. The ISP programming uses the SPI (Serial Peripheral Interface) peripheral that is integrated inside these microcontrollers (it's also present in the ATmega328P). Two devices can exchange data over SPI using only 3 lines, MOSI, MISO, and SCK, that must be connected by wiring the SPI pins of the programmer device to the SPI pins of the programmed device. As a programmer we'll use a common Arduino UNO board thanks to the sketch *ArduinoISP* that is incuded into the Arduino IDE:

1. connect your Arduino board to the computer;
2. open the Arduino IDE;
3. choose the "*ArduinoISP*" sketch from "*File/Examples*";
4. select the board "Arduino UNO" from "Tools/Board" and then, from "Tools/Port" the logic port the board is connected to. This changes from system to system: under Windows it will be something like `COM10`; under Linux it will be like `/dev/ttyACM0`; under Mac OS X it will be like `/dev/tty.usbmodem14211`;
5. click on the upload icon or chose "*File/Upload*" to write the sketch on the Arduino UNO board;
6. at the end of the operation, disconnect the board from your computer.

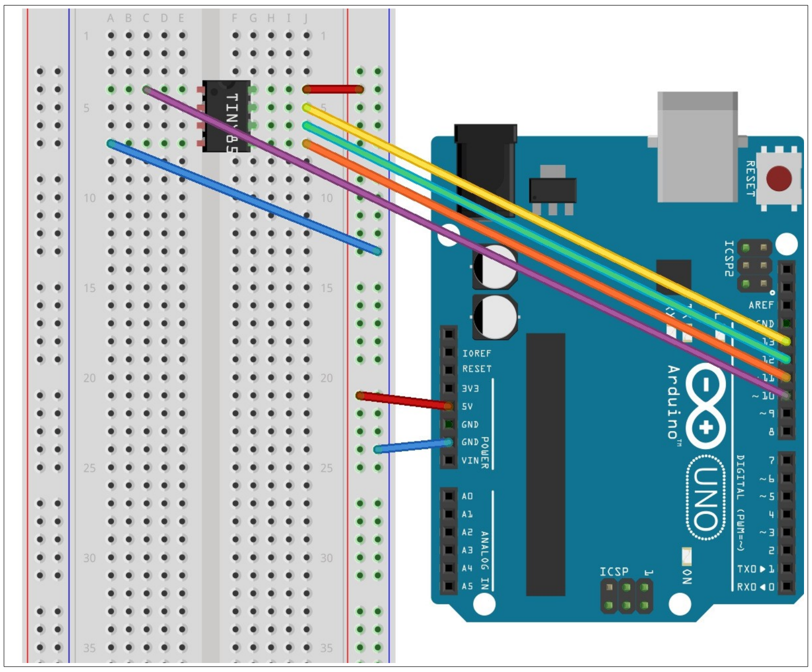
Now your Arduino contains the software that can emulates an STK500 programmer, so that you'll be able to use it to write the sketches on the Attiny you'll use.

If you're using a board model older than the UNO R3 you could experience the so-called autoreset problem, it means that the Arduino will reset itself when you'll try to upload the sketch to the Attiny. This problem comes from the bootloader that is installed on your board: the Arduino UNO R1 and part of the first UNO R2 had the Optioot 4.0 that was affected by this while the last UNO R2 and the UNO R3 use the Optiboot 4.4 that solved the issue. The ultimate solution is to write the newer bootloader (inlcuded into the IDE 1.5) on the board, but for this operation you need another Arduino board to use like a programmer or a third-party programmer (like USBtinyISP or other). A temporary solution is to use a polarized capacitor of 10uF that you have to insert into the pins RST (cathode of the capacitor) and 5V (anode of the capacitor), to be used only when you upload the sketch to the Attiny microcontroller and to remove immediately after that (it avoids the reset of the Arduino board so that you cannot program it).

Now get a breadboard and some jumpers to do the wirings. Replicate the wirings of the picture below (use the colors of the jumpers to help yourself doing the wirings).

Note:
every microcontroller has its own MOSI/MISO/SCK lines put on different pins. Check the datasheet of the chip you're using and change the connections as need (the MOSI/MISO/SCK pins are showed in the microcontroller pinout that's usually in one of the first pages of the datasheet).

Connection	**Arduino UNO**	**Attiny85**
VCC	Pin 5V	Pin 8
GND	Pin GND	Pin 4
MOSI	Pin 11	Pin 5
MISO	Pin 12	Pin 6
SCK	Pin 13	Pin 7
RESET	Pin 10	Pin 1



The first operation to be done is always the setting of the microcontroller's *fuses* to set the desired clock frequency. Every chip has several clocks to choose from: the Tiny core offers the 1 and 8 MHz clock speed because this frequencies are available with the internal oscillator of the chips, so that you don't need to use external crystals that should use 2 of the few pins offered by these chips. Moreover, these chips are many times used in battery-powered projects, where lower frequencies can permit to use lower supply voltages, for better power saving.

To select the clock source we just have to "write the bootloader on the microcontroller": I used the quotation marks because, as we said before, the Attiny microcontrollers don't reserve an area into the Flash memory to store a bootloader. This operatin is only executed to modify the *fuses* directly from the IDE by using an empty bootloader that serves to befool the Arduino IDE. Here is an example: let's set an Attiny85 to work at 8MHz.

1. Make the wirings as shown in the picture above;
2. connect the Arduino to the computer (remeber? We already had uploaded the ArduinoISP sketch);
3. open the IDE, then select "Attiny 25/45/85" from "Tools/*Board*";
4. select "*Attiny85@8 Mhz (internal oscillator; BOD disabled)*" from "*Tools/Micro*";
5. select "*Arduino as ISP*" from "*Tools/Programmer*": make attention, don't select "*ArduinoISP*", that refers to a new Arduino programming board, released recently;
6. select "Write bootloader" from "*Tools*".

Now the IDE will upload the empty bootloader on the Attiny85, and during this operation set up the *fuses* too. This operation also completely erases the microcontroller's memory: if you want to do a full clean of the Flash, do the upload of the bootloader (remember that avrdude, the software used to upload the sketches, only overwrites the Flash used by your program, leaving unchanged the rest of the memory).

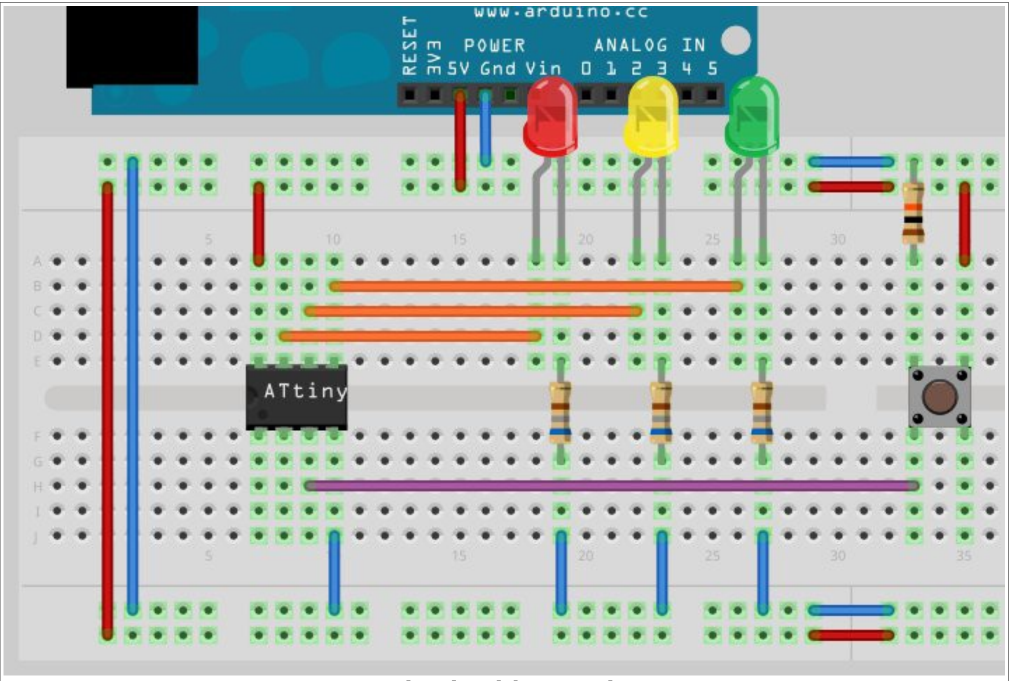
4. Programming an ATtiny85

Now we're ready to use the tinyAVR microcontrollers for our projects within the Arduino IDE, that we'll use to write the code and to send the sketch with few clicks.

Let's start with a little project, a circuit with 3 LEDs whose flashing schematics can be changed by pressing a switch, with an Attiny85 to control the whole system. The sketch that we're using can be found in "*File/Sketchbook/Attiny/esempi/test_3_led*".

Check if in "Tools/Micro" you selected the entry "*Attiny85 @ 8 Mhz*" (if doesn't, select it again), then press the icon "Upload" (or select the corresponding entry from the "*File*" menu).

Now unplug the Arduino, then unplug the ISP programming jumpers and finally make the following circuit:



Test circuit with an Attiny85

You'll need:

- 1x tactile switch for surface mounting;
- 3x resistors for the LEDs (I've used the value of 680 ohm);
- 1x 10Kohm resistor ;
- 3x LEDs, maybe of 3 different colors, e.g.: red, yellow, green.

Connect the 3 LEDs as follow:

- red LED: anode to pin #7 of the ATtiny85, cathode to GND with one of the 680 Ohm resistors;
- yellow LED: anode to pin #6 of the ATtiny85, cathode to GND with one of the 680 Ohm resistors;
- green LED: anode to pin #5 of the ATtiny85, cathode to GND with one of the 680 Ohm resistors;

The tactile switch must be connected as shown in the picture above, with the 10K pull-down resistor that goes to GND on the same line that is connected to pin #3 of the Attiny85, while one of the pins of the other side must be connected to +5V. Now plug again the Ardiuno to your computer, so that you can power the circuit through the USB port.

What are you seeing?
The first LED, the red one, blinks.

What can we do?
By pressing the switch, the yellow LED start blinking. Another press and the green LED start blinking. One more press and all the 3 LEDs blink sequentially. With the last 2 presses we can choose to have all the LEDs steady on or off.

How does the code work?
The code reads the switch pressings and change the value of a variable that selects the flashing scheme. Instead of delay(), millis() has been used to calculate the timings of the flashing schemes because the first function would stop the code execution so that the program wouldn't be able to read the pressings of the switch while flashing a LED.

5. Serial communication

After this first example, that we have used it to learn how to wire and program the Attiny85 through the Arduino board, let's examin another situation, how to use the serial communication. For this task will use again an Attiny85 because this microcontroller, like his little brothers Attiny25/45 but also the Attiny24/44/84, don't support the hardware USART (serial peripheral): the only microcontrollers of the Attiny family that have the integrated hardware serial are the Attiny2313/4313. To use the serial, we must implement it in software.

To do so we'll use the **SoftwareSerial** library, that is distribuited with the Arduino IDE, that make this: it emulates through software the hardware serial peripheral. There's a problem: the library has been written to work with microcontrollers that has more than one 16-bits timer, so we must modify it otherwise the compiler will return an error. Another problem of the library is that it doesn't support the Attiny microcontrollers in RX (receiving data) so we need to correct this aspect too.

Note:
the SoftwareSerial only works with microcontrollers with a clock of 8MHz and it doesn't support speeds of 1MHz.

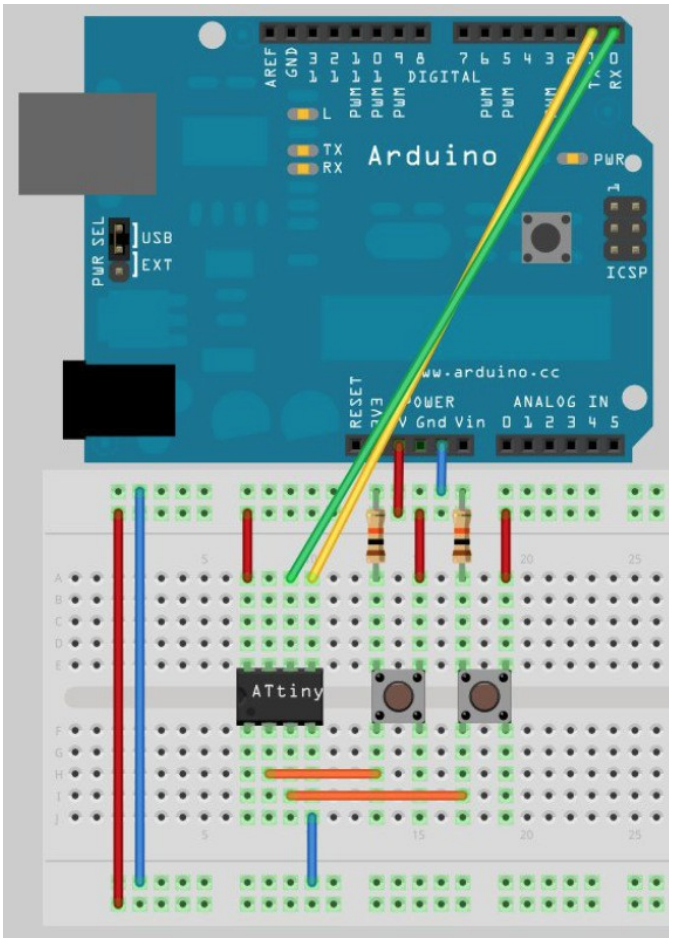
Now, let's wiring again our Attiny85 for the ISP programming and upload the sketch called *test_serial.ino* (you'll find it in "*File/Sketchbook/Attiny/esempi*").

Then we have to set up the circuit shown on the right. We need 2 switches and 2 10K resistors.

Wiring:

- connect the switches to pin #2 and #3 of the Attiny85, each of them with its pull-down resistor connected to GND;
- connect the pin #5 of the Attiny85 to the pin #D1 of the Arduino (it's the pin marked as "TX") and the pin #6 of the Attiny85 to pin #D0 od the Arduino ("RX");
- connect +5V and GND as in the picture.

The SoftwareSerial library can work using any pin of the microcontroller: we have decided to use a couple of pin from the side that is closer to the Arduino but we can choose any other pin. The switches will be used to send data to the Arduino.



Circuit to test the serial communication

Select the sketch “*File/Sketchbook/Attiny/esempi/test_seriale_tiny*”: it just sends through the TX line a different caracter for each of the buttons. Let's upload it using the method shown in the previous chapter. Now, unplug the Arduino and then remove the ISP wirings. Now upload to the Arduino board the sketch *“File/Sketchbook/Attiny/esempi/test_serial_arduino”:* it waits for data incoming via the software RX line and ping back any caracter to the hardware serial TX of the board. To do this, select the “*Arduino UNO*” board from “*Tools/Board*” and then upload the sketch.

Again, unplug the Arduino, make the wirings shown in the picture above and plug the Arduino another time. Now, open the IDE and then open the serial monitor (or another terminal if you don't want to use the console of Arduino): now, press the buttons on the breadboard. If everything is connected correctly, we'll see different characters (“1” or “2”) when we press the switches.

6. Using the I2C

The I2C (or I^{2}C) is a bus developed by Philips (now NXP) used to let two or more deviced to communicate between each other just by using only 2 wires. Every device is identified on the bus with a number, or ID, that uniquely addresses every device. Each device can work as a “master” (who send data requests) and as a “slave” (who sends the required data). Unfortunately, the Attiny25/45/85 microcontrollers don't have any hardware support for the I2C bus but instead they have a peripheral called USI (Universal Serial Interface) that can emulate the I2C through the 2-Wire (or TWI) protocol, very close to the I2C.

Thanks to the community of Arduino enthusistats, we can use a modified version of the Wire library that is compatible with the Attiny microcontrollers, the TinyWire library. Again, we have to modify the library because the standard TinyWire only works at clock speeds of 1MHz so it wouldn't work with our 8MHz chips. Morevoer, the original version of the library doesn't support the Attiny84 so we had to do another mod.

The TinyWireM and TinyWireS libraries included in this core have already been modified and can be used respectively when you set a device as a “master” (the former) or a “slave” (the latter).

Note:
by default, the RX and TX buffers of the TinyWireM and TinyWireS libraries are set respectively to 16 and 32 bytes. Keep in mind these values because they use this amount of memory space, and the Tiny microcontrollers don't have big quantities of RAM: if you would experience RAM overflows, you could need to change those buffers. If you want to do this, open and edit the following files:

- TinyWireM: file **TinyWireM.h**
  - change the value in this line:

    ```
    #define USI_BUF_SIZE 16 // bytes in message buffer
    ```
- TinyWireS: file **usiTwiSlave.h**
  - change the values in these lines:

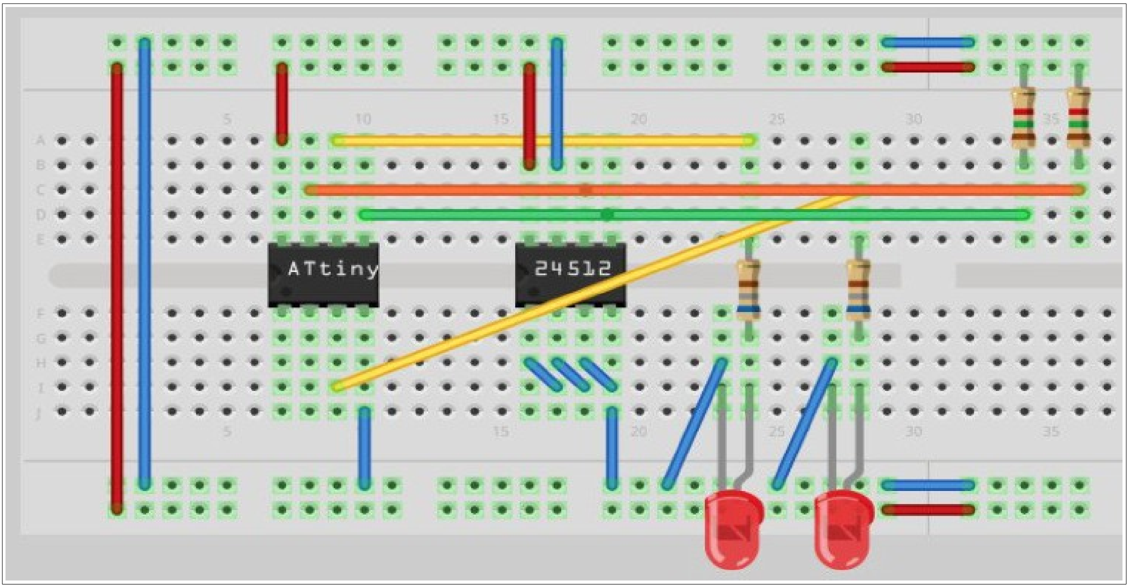
  -
    ```
    #define TWI_RX_BUFFER_SIZE (32)
    #define TWI_TX_BUFFER_SIZE (32)
    ```

To test the I2C communication we'll use an EEPROM memory chip (We've used a 24LC512 but you can use a 24LC256 or a 24LC128 too). Let's look for the following components: 2 LEDs with their resistors (in the example we used a couple of 680 ohm resistors) and 2 1.5Kohm resistors. Wirings:

- Attiny:
  - pin #8 to +5V
  - pin #4 to GND
- 24LC512:
  - pins #1, #2, #3, #4, and #7 to GND
  - pin #8 to +5V

- LED1:
  - anode to pin #6 of the Attiny, with the 680 ohm resistor in series
  - cathode to GND
- LED2:
  - anode to pin #3 of the Attiny, with the 680 ohm resistor in series
  - cathode to GND
- Bus I2C:
  - connect the pin #5 of the ATtiny to pin #5 of 24LC512 and then to +5V with the 1.5Kohm pull-up resistor
  - connect the pin #7 of the ATtiny to pin #6 of 24LC512 and then to +5V with the 1.5Kohm pull-up resistor

Here is the schematic:



Now we need to upload the sketch called *Attiny/test_i2c* on the Attiny by using the Arduino as an ISP programmer (see the previous chapters) and the look at how the LEDs flash.

When the sketch will start, the Attiny will light up both the LEDs for a short time to inform the user that the program is running. Then, the sketch will write into the first 10 bytes of the EEPROM memory some test values. After this, the LEDs will be light off and then only the second LED will light up again, to inform the user that the reading has been started. Now the first LED will blink only if a "1" will be read from the EEPROM memory, and will stay off if a "0" is read. After the 10 bytes, the cycle will start again.

7. Using the SPI

The microcontrollers of the Attiny family don't have a real SPI peripheral, instead the USI peripheral (seen in the previous chapter) is used to manage the Three-Wire connections, that are compatible with the SPI. The USI peripheral has to be set up via software but, luckily, there's a library developed to use the SPI communication on Attiny microcontrollers: it's name is tinyISP, originally written by Jack Christensen. I modified it by adding the support fo the Attiny2313/4313 MCUs.

At the moment, the library permits SPI communications only with MCU as a master. Moreover, it doesn't support any SS (Slave Select) pin so, if it's required from the application, then the programmer has to provide the proper functionalities via software.

The library can achieve a discrete transfer speed: the author claims that it's about 15 times faster than shiftOut speeds and, with a clock that usually it's about 1/10 of the system clock (if the MCU runs at 1MHz, the tinyISP sends datas at ~100Khz). The show the usage of tinyISP, the author provides a couple of examples, available in the IDE from "*Examples/tinySPI-master*". Due to memory requirements, if you use the library with Attiny MCUs with less than 4KB of Flash you won't have a lot of space for your code.

8. Conclusions and credits

The microcontrollers of the Attiny family are good chips with interesting features: they offer a very good computing power (they can work up to 20 Mhz, like the bigger Atmega micrcocontrollers) in small packages. The usage of the 8MHz internal oscillator allows to save 2 external I/O lines: this is very interesting because the Attiny microcontrollers don't offer a huge number of pins.

In my humble opion, these chips aren't widespread: though, they have interesting features. Another negative point is that, apart from Atmel Studio, they don't have much support from tools and IDEs. This is solved in part thanks to the core and libraries developed by the community, even if they lacks in documentation, so that the users have to find the issues and solve by themselves, like in the cases of SoftwareSerial and I2C shown above, sometimes with a lot of attempts and a bit of luck.

We would like to thank:

- the user PaoloP of the Arduino forum that has given a consistent contribution by modifiyng several configuration files to let the IDE be able to incorporate the core Tiny with older versions, work reused for the following releases of this guide;
- the user Coding Badly of the Arduino forum for his core Tiny;
- BroHogan for his library TinyWire;
- Jack Christenses for his library tinyISP;
- the Arduino staff for this beautiful project.

9. Licenses and warranties

Written by Leonardo Miliani

Version 1.0 – Revison date: 2011/04/10

Version 1.2 – Revison date: 2012/02/17

Version 1.5-04 – Revison date: 2013/11/11

Version 1.5.7 – Revison date: 2014/07/14

Version 1.5.8 – Revison date: 2014/10/13

Version 1.5.8-2 – Revision date: 2014/11/28

Version 1.6.7-1 – Revision date: 2016/02/20