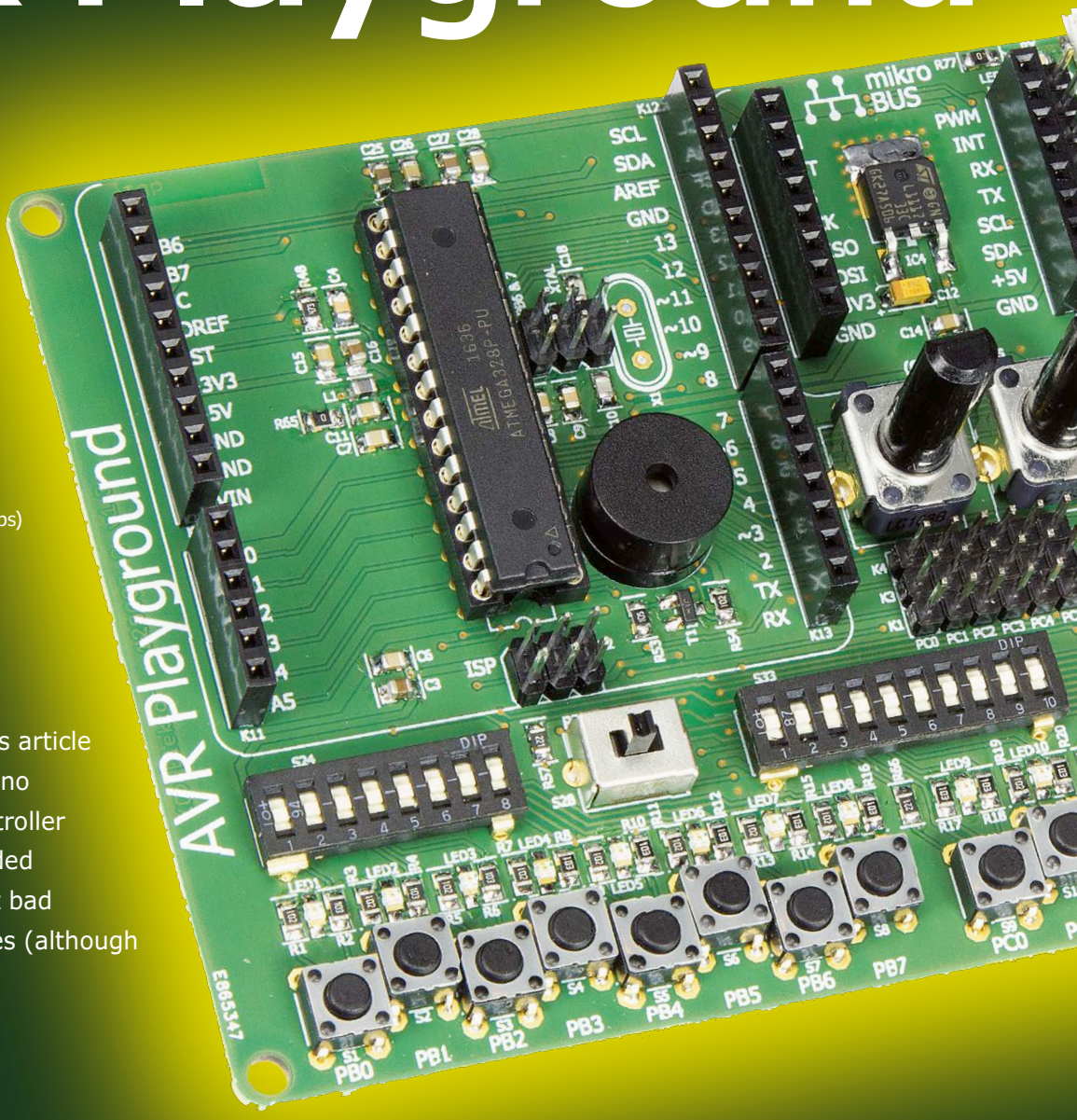
eLABS PROJECT

AVR Playground

Improves the way to do Arduino



By **Clemens Valens** (Elektor Labs)

The board presented in this article is a hybrid of an Arduino Uno and a traditional microcontroller development board, intended for 'doing Arduino' without bad connections and loose wires (although it doesn't disallow it).

Before the rise of Arduino, microcontroller development boards had on-board peripherals like pushbuttons and LEDs, a display, one or more potentiometers for analog signals, extension connectors, etc. and, of course, a decent power supply. The goal of these boards was to provide an easy way to start learning the microcontroller without having to solder or add other components. The AVR Playground – a playful reference to the Arduino forum known as the Arduino Playground – was designed with this in mind, and extended with things from Arduino that we have learned to appreciate.

The board can roughly be divided into four horizontal zones (**Figure 1**), with, from top to bottom, the following functions:

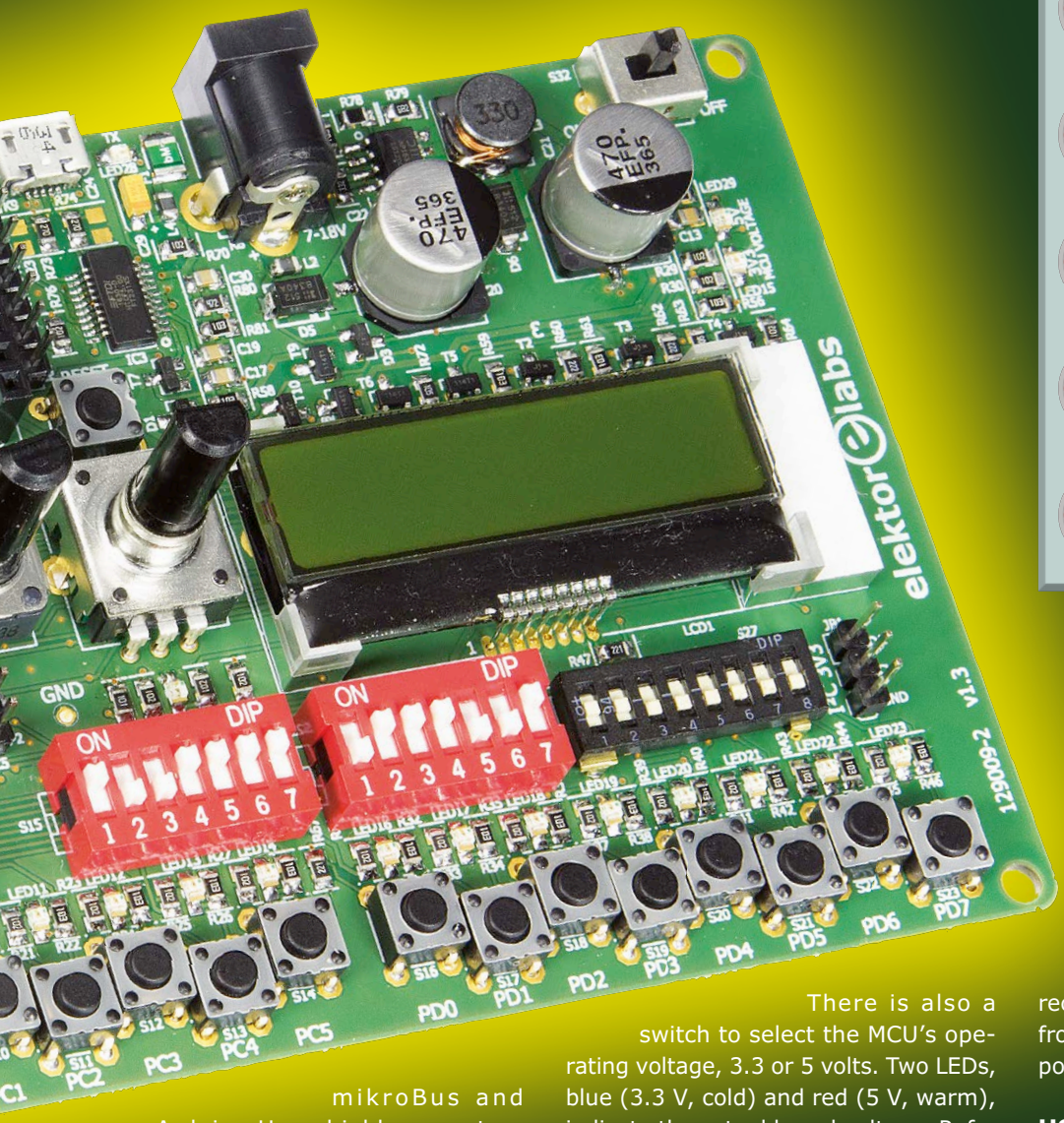
1. extension connectors, USB-to-serial converter and power supply;
2. user interface peripherals like a buzzer, rotary controls and a display;
3. configuration switches;
4. pushbuttons and LEDs.

If you want to see the schematics, the PCB or the component list of the AVR Playground, please refer to [1].

Extension connectors

The top-left corner of the board accommodates the microcontroller and Arduino Uno compatible extension connectors. This part of the board behaves exactly like an Arduino Uno.

Next to it sits a mikroBus expansion slot. This extension standard, developed by the Serbian company MikroElektronika, is gaining popularity thanks to the availability of hundreds of cheap extension boards, ranging from GPS receivers to humidity sensors and from LED arrays right up to phone modems. Together, the



mikroBus and Arduino Uno shield connectors give the AVR Playground user access to a huge library of extension boards.

Pushbuttons, LEDs & DIP switches

The bottom part of the board, zone 4, is occupied by pushbuttons and LEDs connected to every microcontroller port that can be used. The pushbuttons allow applying a logical level to a port, while the LEDs provide visual feedback. The DIP configuration switches in zone 3 of the board determine:

- if a GPIO port is pulled up or down or not at all;
- if pressing a pushbutton provides a logic low or high;
- if the LEDs are connected or not and if user interface peripherals from zone 2 are connected or not.

There is also a switch to select the MCU's operating voltage, 3.3 or 5 volts. Two LEDs, blue (3.3 V, cold) and red (5 V, warm), indicate the actual board voltage. Refer to **Tables 1 to 5** for details.

Power supply

The power supply was designed with quality and robustness in mind. Protected against short circuits and high temperatures, it is able to deliver 5 V at 1 A without complaining. There is one condition, though: you must power it from an external power source like a DC adapter (wall wart) capable of providing about 7 V minimum (and, of course, enough current). The USB port may also be used as a power source but this will limit the available current to 500 mA in order to protect your computer. The 5 V drives a beefy 3.3-V low-dropout voltage regulator so that in 3.3-V mode too, enough power is available for your experiments. An on/off switch can cut the power to the rest of the board, enabling safe hardware reconfiguration without disconnecting it from the computer and losing the serial port connection.

USB-to-serial converter

The USB-to-serial converter not only acts as a 5-V power source for the board, it is also the programming interface for Arduino sketches and, of course, a USB-compatible serial port for user applications. It can be disconnected entirely from the microcontroller, to free up GPIO pins for instance or to use it on other ports.

Human interface devices

In zone 2, i.e. the middle of the board, we find typical user interface devices like a buzzer, two analog controls in the shape of potentiometers, a rotary encoder and an alphanumerical display. Together with the pushbuttons and LEDs they allow the creation of human-friendly applications without soldering. With an Arduino Uno, a few prototyping boards and a spaghetti of connecting wires it is possible to achieve the same goals, but

PROJECT INFORMATION

Microcontrolers

Arduino | ATmega328 | Programming

entry level
➡ intermediate level
expert level

4 hours approx.

SMD Soldering

€75 / $80 / £65 approx.

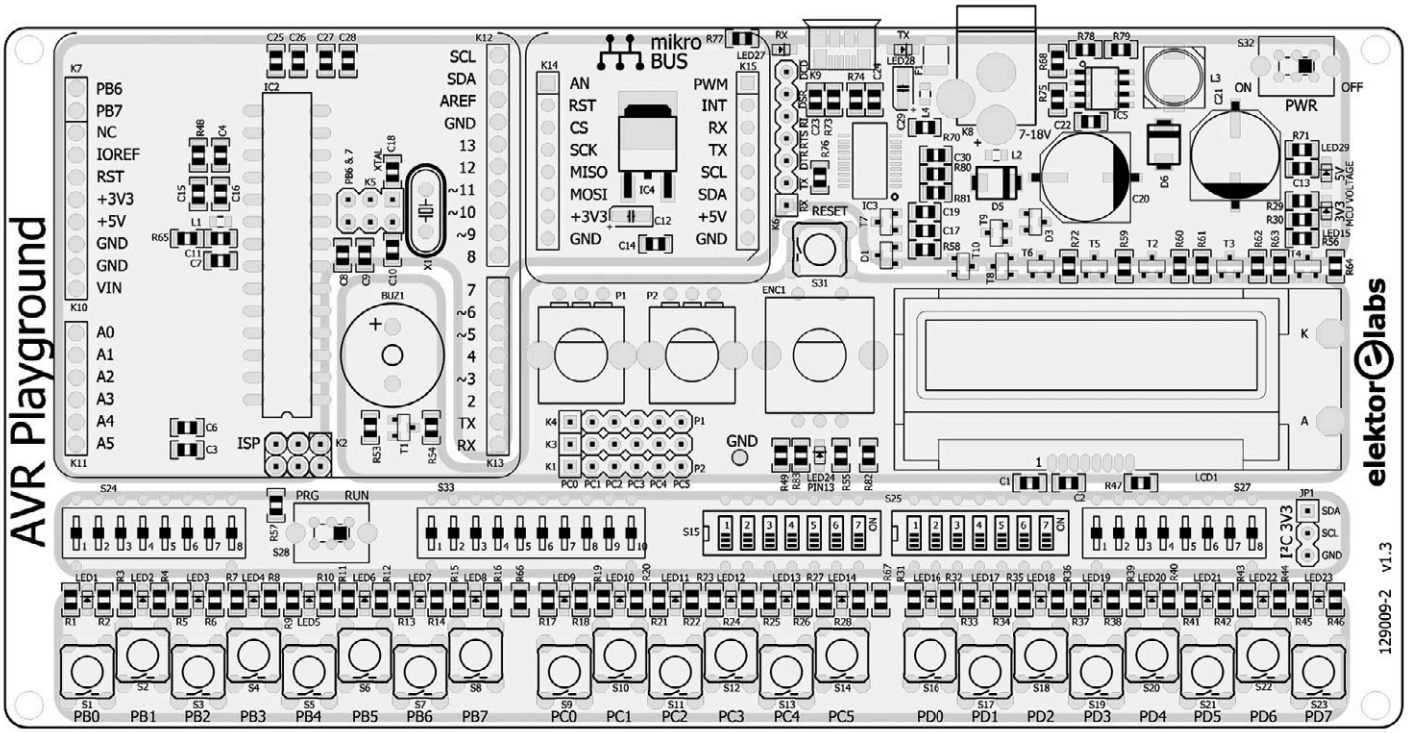


Figure 1. The AVR Playground can be divided into four functional zones.

things are much more comfortable and reliable when using a tool like the AVR Playground.

LCD

The display, a small LC one with backlight, has an I²C interface instead of the frequently seen 4-bit semi-parallel or 8-bit parallel interface. The evident advantage is that it only requires two wires for connecting it to the MCU. The inconvenience is the need for a special driver, but we took due care of that in the Boards Package (see below).

Table 1. The functions of DIP switch S15.

S15	Function	Off	On
1	Buzzer	Disconnected	Connected to PB1
2	LEDs Port B	Disconnected	Connected to GND
3	LEDs Port C	Disconnected	Connected to GND
4	LEDs Port D	Disconnected	Connected to GND
5	USB-to-serial RXD	Disconnected	Connected to PD1
6	USB-to-serial TXD	Disconnected	Connected to PD0
7	USB-to-serial DTR	Disconnected	Connected to Reset

Table 2. The functions of 7-way DIP switch S25.

S25	Function	Off	On
1	MCU voltage	5 V	3.3 V
2	'Arduino LED'	Disconnected	Connected to PB5
3	LCD SDA	Disconnected	Connected to PC4
4	LCD SCL	Disconnected	Connected to PC5
5	Not used		
6	LCD Backlight	On (if S25-7 in On position)	Connected to PD7
7	LCD Backlight	Off	Controllable

Table 3. The functions of DIP switch S24.

S24	Port	Down	Middle	Up
1	PB0	Pulled down	Not pulled	Pulled up
2	PB1	Pulled down	Not pulled	Pulled up
3	PB2	Pulled down	Not pulled	Pulled up
4	PB3	Pulled down	Not pulled	Pulled up
5	PB4	Pulled down	Not pulled	Pulled up
6	PB5	Pulled down	Not pulled	Pulled up
7	PB6	Pulled down	Not pulled	Pulled up
8	PB7	Pulled down	Not pulled	Pulled up

Rotary controls

Two potentiometers provide analog signals to the microcontroller. By positioning a jumper, they can be connected to any of the six analog inputs of the MCU without the risk of both being connected to the same input at the same time. The rotary encoder is in reality the same thing as (up to) three pushbuttons hence it's effectively connected in parallel to the pushbuttons on PD3, PD4 and PD5.

Arduino LED

The LED connected to PB5 (Pin 13) on the Arduino Uno is available on the AVR Playground too; it is located below the rotary encoder. This LED is used in many sketches, reason why it is present on this board.

Clock frequency & reset issues

The preferred clock oscillator for the AVR Playground is the MCU's internal 8-MHz RC oscillator. This ensures that the MCU will always work within its specifications, no matter if the MCU voltage is 5 V or

3.3 V. Because the quartz crystal is disconnected by default, ports PB6 and PB7 are available for user applications. If a crystal is required, it can be soldered on the board and connected to the MCU by moving two jumpers.
In normal operation port PC6 functions as the reset input of the MCU. It is possible to disconnect it from its reset function by programing the MCU's RSTDSBL fuse. However, that's not recommended as it will disable MCU programming over the serial port **and** through the ISP connector. The only way to reprogram the MCU is to remove it from the board and land it in a so-called parallel programmer. Because disabling the reset input is incompatible with the objectives of the AVR Playground, PC6 is not treated like the other port pins and there is no LED connected to it (there is, however, a pushbutton for it: Reset).

Table 4. The functions of DIP switch S33.

S33	Port	Down	Middle	Up
1	Port B pushbutton level	Low	Disconnected	High
2	Port C pushbutton level	Low	Disconnected	High
3	Port D pushbutton level	Low	Disconnected	High
4	PC0	Pulled down	Not pulled	Pulled up
5	PC1	Pulled down	Not pulled	Pulled up
6	PC2	Pulled down	Not pulled	Pulled up
7	PC3	Pulled down	Not pulled	Pulled up
8	PC4	Pulled down	Not pulled	Pulled up
9	PC5	Pulled down	Not pulled	Pulled up
10	Not used			

Table 5. The functions of DIP switch S27.

S27	Port	Down	Middle	Up
1	PD0	Pulled down	Not pulled	Pulled up
2	PD1	Pulled down	Not pulled	Pulled up
3	PD2	Pulled down	Not pulled	Pulled up
4	PD3	Pulled down	Not pulled	Pulled up
5	PD4	Pulled down	Not pulled	Pulled up
6	PD5	Pulled down	Not pulled	Pulled up
7	PD6	Pulled down	Not pulled	Pulled up
8	PD7	Pulled down	Not pulled	Pulled up

Installing the AVR Playground

Although it is possible to use the AVR Playground without installing any software, we would not recommend it. Because the board sports features not supported by the standard Arduino IDE, we prepared some libraries that make using the on-board peripherals easier. Having that said, if, for some reason, adding a board to the Arduino IDE is unwanted, know that the standard board 'Arduino Pro or Pro Mini' with as processor (from the 'Tools → Processor' menu) the 'ATmega328 (3.3 V, 8 MHz)' can be used instead. The voltage is not important, what counts is the frequency having to match that of the MCU's clock oscillator.

Adding a new board to the IDE is not difficult and there is even a special tool for this: the Boards Manager, accessible at the top of the 'Tools → Boards' menu. The Boards Manager allows installing, updating and removing third-party boards. For this to work, the manufacturer of such a board must provide a Boards Package telling the IDE what to download and use for the board.

The procedure to install the AVR Playground's Boards Package is quite simple but requires an Internet connection. It starts from the 'File' menu by opening the 'Preferences' dialog of the Arduino IDE (see **Figure 2**, Arduino version 1.6.13 or newer from arduino.cc; do not use 1.7.x from arduino.org). Copy the URL below or, even better, read the QR code from **Figure 3**:

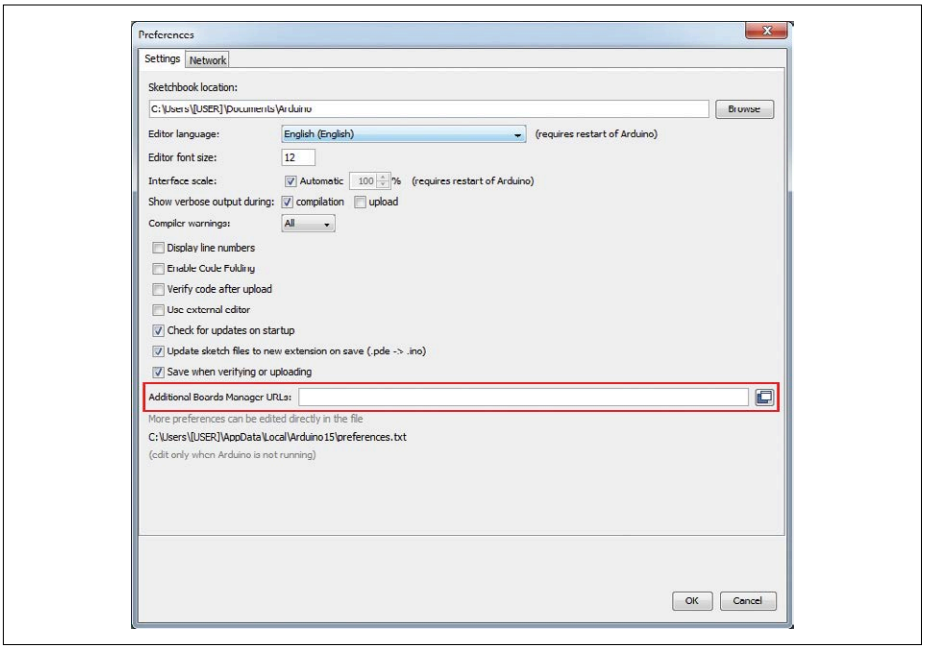


Figure 2. This is where you enter the url to access the AVR Playground Boards Package.

```
https://raw.githubusercontent.
    com/ElektorLabs/arduino/
    master/package_elektor_
    boards_index.json
```

Figure 3. Avoid typing mistakes by reading this QR code with your webcam, then copy-paste the URL into the 'Additional Boards Manager URLs' box of the 'Preferences' dialog.

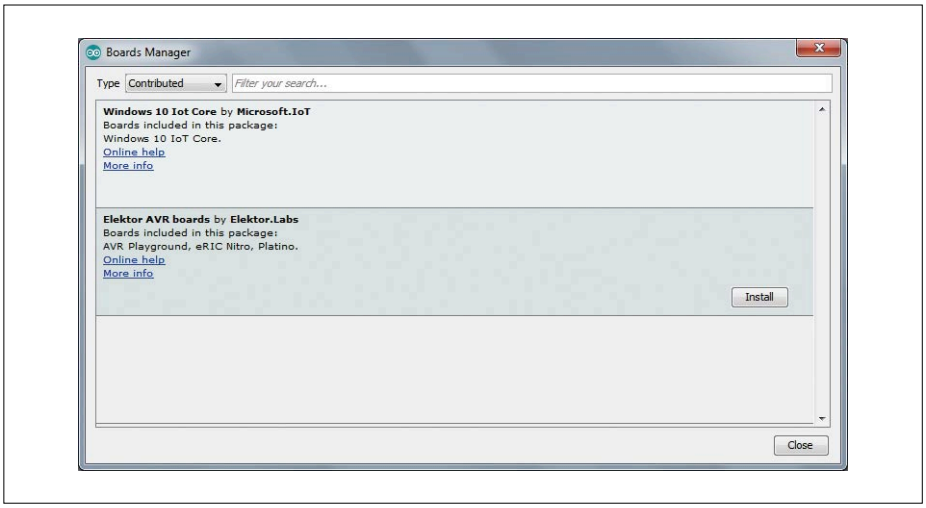


Figure 4. Once the IDE has found the AVR Playground Boards Package it will allow you to install it.

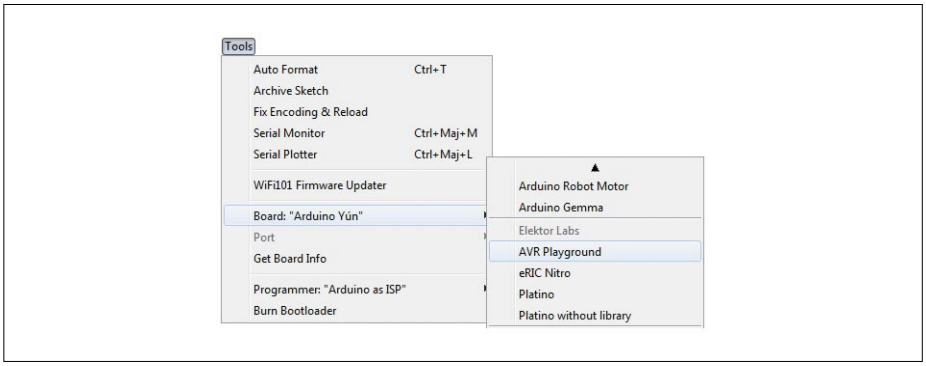


Figure 5. After installing the Boards Package, scroll through the Boards list and select the AVR Playground.

(one line, no spaces, beware of typing errors) into the 'Additional Boards Manager URLs' box of the 'Preferences' dialogue. Close it when done.

Open the Boards Manager ('Tools → Boards'). In the upper left corner of the window that opens select 'Contributed', look for the AVR Playground in the list that appears, click on it and then click the 'Install' button (**Figure 4**). The IDE will download the required files and copy them to right location. When done, close the window. Now the AVR Playground will be listed somewhere in the 'Boards' menu, under the header 'Elektor Labs' (**Figure 5**). Of course you should connect the AVR Playground to your computer before you can select its 'Port'.

Enter game programming

We developed a fun application using many of the board's options, it's a simple game called 'Simon Says' inspired by a famous game from the early years of the microcontroller. The original game came as a round black plastic box with four large, backlit, colored buttons: red, blue, green and yellow. The computer plays a random luminous sequence where every light is accompanied by a musical note. When done the player is invited to play the same sequence. If the player fails, the game restarts. If the player succeeds the sequence is extended by one randomly chosen color/note. The game is still rather popular among programmers and you can easily find a version of the game for your smartphone or to play online.

The AVR Playground has everything needed to create and play the game: LEDs, pushbuttons, a buzzer and, of course, a microcontroller. What's more, the game can be extended by adding the LCD to show things like instructions, the high score, and some other statistics. There are no colored lights but bits of colored plastic film or paper can help here.

In what follows, fragments of the demo program are highlighted, the complete source code can be downloaded freely from [1].

Table 6. The ports, their pins and their functions in the game. Ten GPIO pins remain unused, showing that a microcontroller with fewer pins might suffice, making the final design cheaper.

Port B	Function	Port C	Function	Port D	Function
PB0	Not used	PC0	Not used	PD0	RXD
PB1	Buzzer	PC1	Not used	PD1	TXD
PB2	Not used	PC2	Not used	PD2	LED0
PB3	Not used	PC3	Not used	PD3	LED1
PB4	Not used	PC4	LCD SDA	PD4	LED2
PB5	Not used	PC5	LCD SCL	PD5	LED3
PB6	Not used			PD6	LED4
PB7	Not used			PD7	LED5

Port mapping

The first step is to decide which ports will be used for what function (**Table 6**). This also greatly determines the setting of the configuration DIP switches (see **Tables 7 to 11**). The settings chosen will switch the LCD's backlight on, and, because the level of the pushbuttons on port D is set to logic high, an LED will automatically light when the corresponding button is pressed. To make this free visual feedback work properly, the pull-down resistors on these pushbuttons need to be activated.

Controlling the LEDs...

... is slightly more complicated than usual because the pushbuttons are connected to the same pins. Also, to improve flexibility, a lookup table is added allowing the use of other LEDs simply by modifying the table. **(Listing 1)**

Reading the pushbuttons...

... is based on the same technique as lighting the LEDs, except for an input now being read instead of an output being driven. Due to the way the DIP switches are set, a pressed button will produce a logic high.

Table 7. Settings of S15 according to our plans.

S15	Function	Position
1	Buzzer	On
2	LEDs Port B	Off
3	LEDs Port C	Off
4	LEDs Port D	On
5	USB-to-serial RXD	On
6	USB-to-serial TXD	On
7	USB-to-serial DTR	On

Table 8. Settings of S25 for our game.

S25	Function	Position
1	MCU voltage	Off
2	'Arduino LED'	On
3	LCD I²C SDA	On
4	LCD I²C SCL	On
5	Not used	
6	LCD Backlight	Off
7	LCD Backlight	On

Table 9. S24 controls Port B, it's not used in our game.

S24	Port	Position
1	PB0	Middle
2	PB1	Middle
3	PB2	Middle
4	PB3	Middle
5	PB4	Middle
6	PB5	Middle
7	PB6	Middle
8	PB7	Middle

Table 10. S33 controls Port C and the pushbutton's active levels.

S33	Function	Position
1	Port B pushbutton level	Middle
2	Port C pushbutton level	Middle
3	Port D pushbutton level	High
4	PC0	Middle
5	PC1	Middle
6	PC2	Middle
7	PC3	Middle
8	PC4	Middle
9	PC5	Middle
10	Not used	

Table 11. S27 controls Port D.

S27	Port	Position
1	PD0	Middle
2	PD1	Middle
3	PD2	Down
4	PD3	Down
5	PD4	Down
6	PD5	Down
7	PD6	Down
8	PD7	Down

A problem with pushbuttons is that they are mechanical devices in essence, exhibiting milliseconds or more of time lag for the contact to settle, hence it is necessary to 'debounce' the buttons. An easy way to do this when there are no particular (timing) constraints as in our simple game, is to scan the buttons a second time after having waited a short while. Only if a button is read twice as being pressed, the program decides it's a valid button press. By scanning periodically at a high enough rate (10 Hz or so) the chances of missing a key press are minimal. A logic AND ('&') of two scan results will remove inconsistent reads. **(Listing 2)**

Random numbers

The game needs a sequence of random numbers (corresponding to the LED numbers) that's long enough to make most players fail before reaching the end. The sequence can be stored in a table. The size of the table is important. If it is too small, it is too easy for the player to outplay the computer; if it is too large, the player gets discouraged and will give up. The table can be filled at the start of a game.

The function `rand` is available for producing random values in the range 0 to `RAND_MAX` (which corresponds to 32,767 in Arduino). A quick and dirty way to bring the output in the wanted range is by doing a modulo division ('%').

Listing 1.

```
#define LED_BUTTON_TABLE_SIZE  (4)
uint8_t led_button_table[LED_BUTTON_TABLE_SIZE] = { 4, 5, 6, 7 };

void led_set(uint8_t nr, uint8_t value)
{
  pinMode(led_button_table[nr],OUTPUT);
  digitalWrite(led_button_table[nr],value);
}
```

Listing 2.

```
uint8_t button_read_all_debounced(void)
{
  uint8_t buttons = button_read_all();
  delay(10); // Wait for any bouncing to stop.
  buttons &= button_read_all(); // AND the second scan.
  return buttons;
}
```

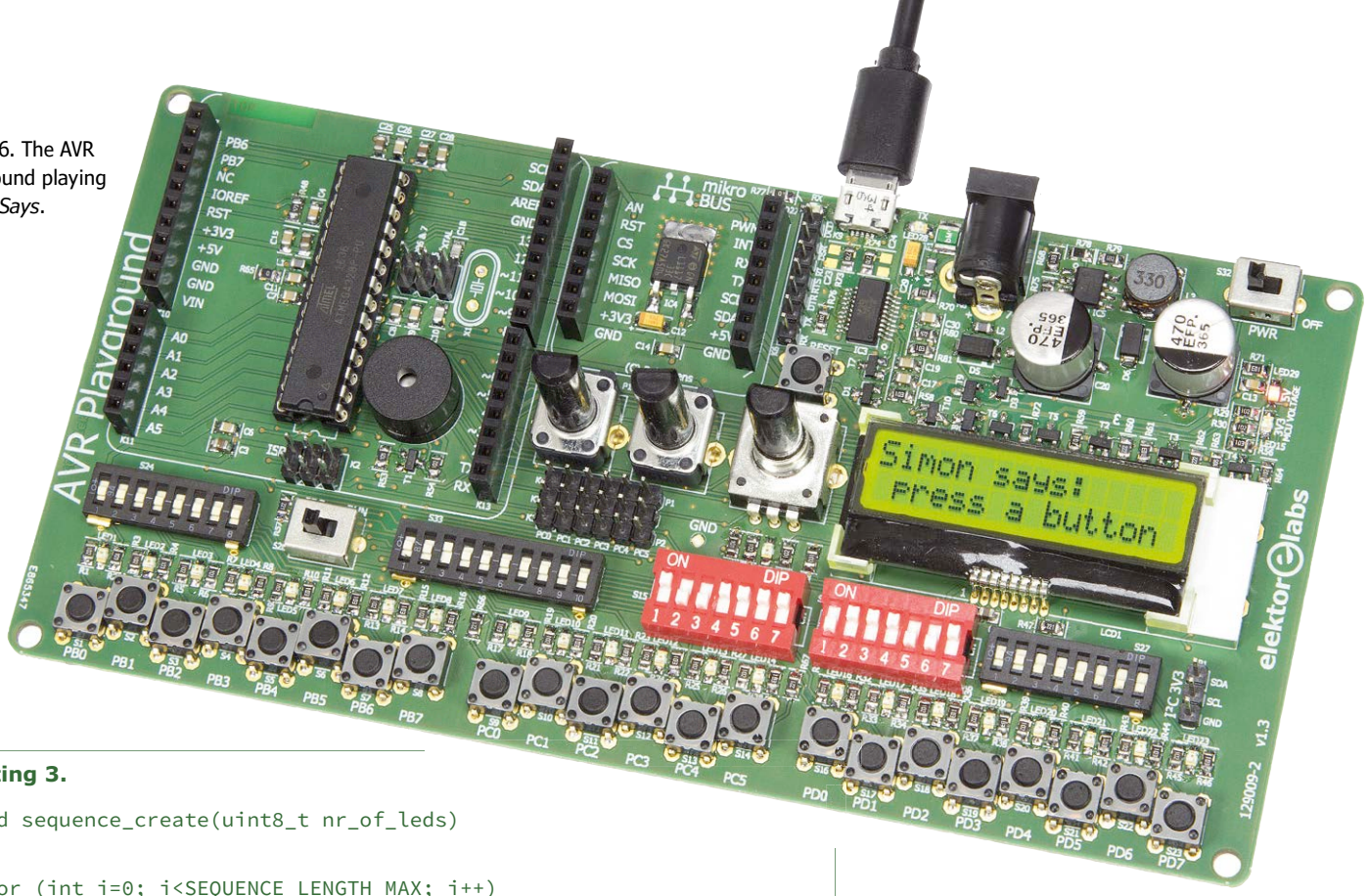


Figure 6. The AVR Playground playing *Simon Says*.

Listing 3.

```
void sequence_create(uint8_t nr_of_leds)
{
  for (int i=0; i<SEQUENCE_LENGTH_MAX; i++)
  {
    sequence[i] = rand()/(RAND_MAX/nr_of_leds + 1);
  }
}
```

Listing 4.

```
void setup(void)
{
  while (button_pressed()==false);
  srand(millis());
  sequence_create(LED_BUTTON_TABLE_SIZE);
  game_round = 1;
}
```

Listing 5.

```
const uint8_t buzzer = 9;
const uint16_t button_sound[LED_BUTTON_TABLE_SIZE] =
{ 554, 659, 880, 1319 }; // C#5, E5, A5, E6 in Hz

#define SEQUENCE_SPEED  (400) /* ms */

void sequence_play(uint8_t len)
{
  for (int i=0; i<len; i++)
  {
    tone(buzzer,button_sound[sequence[i]],SEQUENCE_SPEED/2);
    led_set(sequence[i],true);
    delay(SEQUENCE_SPEED);
    led_set(sequence[i],false);
    delay(SEQUENCE_SPEED/4);
  }
}
```

However, from a mathematical point of view this approach is questionable because the distribution of the output may no longer be uniform and independent due to the random number generator's implementation. **Listing 3** shows a better technique to produce random numbers in a small range.

Random numbers in programming are problematic because programming is deterministic by definition. For the function `rand` to work properly it must be initialized with a random number (the seed), and a chicken-and-egg problem is the result. A trick often encountered is to use the time as the seed, because time, as you know, never stops and thus makes an excellent seed… except in microcontroller systems where time is reset to zero at every (re)start.

For our game time can be used to seed the random number generator if we measure the time it takes for the player to press the first button. Measured in milliseconds the result will hardly ever be the same. **(Listing 4)**

Play a sequence, add sound

Every game round, the sequence is a bit longer and must be played to the user starting from the beginning. If the sequence is played too fast, the player may not be able to memorize it; if it is too slow, he or she may get bored.

To enhance Player Experience we add a musical note to every LED, made audible by the buzzer (connected to pin 9, PB1). **(Listing 5)**

The table `button_sound` holds the frequencies in hertz of the notes to play. The constant `SEQUENCE_SPEED` (in milliseconds) determines the replay speed of the sequence. The function `sequence_play` takes as its argument the length of the sequence to play.

Processing user input

Once the sequence has been played, it is time for the player to repeat it. As soon as the player makes a mistake the program can stop reading pushbuttons. There is one little complication to take care of: avoiding that a very long key press gets interpreted as two or more presses. **(Listing 6)**

Play the game

All that remains to do is to add the function `loop` to glue together the functions `sequence_play` and `sequence_read_buttons` and to keep track of the sequence length and the state of the game. **(Listing 7)**

The bodies of the `if-else` statements have been left largely empty here because it is up to you what to put inside. Consider playing a tune depending on the state of the game, a 'Well Done' tune, a 'You Win' tune and a 'Game Over' tune, most probably accompanied by flashing LEDs. Let your creativity run wild and have fun designing special effects.

(160316)

Web Link

[1] www.elektormagazine.com/160316

Listing 6.

```
bool sequence_read_buttons(uint8_t len)
{
  for (int i=0; i<len; i++)
  {
    uint8_t nr, buttons;

    // Wait for a button press.
    do
    {
      buttons = button_read_all_debounced();
    }
    while (buttons==0);

    // Check that we have a valid button press.
    nr = button_as_number(buttons);
    if (nr!=BUTTON_PRESS_INVALID && nr<LED_BUTTON_TABLE_SIZE)
    {
      // Play sound & wait until it finishes.
      tone(buzzer,button_sound[nr], SEQUENCE_SPEED/2);
      delay(SEQUENCE_SPEED/2);
      // Wait until the player releases the button.
      while (button_pressed()==true);
      // Fail on wrong button.
      if (sequence[i]!=nr) return false;
    }
    else return false;
  }
  return true;
}
```

Listing 7.

```
void loop(void)
{
  sequence_play(game_round);
  if (sequence_read_buttons(game_round)==true)
  {
    if (game_round<SEQUENCE_LENGTH_MAX)
    {
      // Next round, play 'well done' tune?
      game_round++;
    }
    else
    {
      // Player wins, play 'you win' tune?
      game_start();
    }
  }
  else
  {
    // Game over, play 'game over' tune?
    game_start();
  }
}
```