Behav Res (2012) 44:305–313
DOI 10.3758/s13428-011-0163-z

Arduino: A low-cost multipurpose lab equipment

Alessandro D'Ausilio

Published online: 25 October 2011

Abstract Typical experiments in psychological and neurophysiological settings often require the accurate control of multiple input and output signals. These signals are often generated or recorded via computer software and/or external dedicated hardware. Dedicated hardware is usually very expensive and requires additional software to control its behavior. In the present article, I present some accuracy tests on a low-cost and open-source I/O board (Arduino family) that may be useful in many lab environments. One of the strengths of Arduinos is the possibility they afford to load the experimental script on the board's memory and let it run without interfacing with computers or external software, thus granting complete independence, portability, and accuracy. Furthermore, a large community has arisen around the Arduino idea and offers many hardware add-ons and hundreds of free scripts for different projects. Accuracy tests show that Arduino boards may be an inexpensive tool for many psychological and neurophysiological labs.

Keywords I/O board · Cheap laboratory equipment · Experimental control · Arduino · TTLs read · TTLs write

Introduction

Every lab running some kind of behavioral research makes use of several types of equipment and software for experimental control. The goal is typically to record events (behavioral or physiological) and generate signals (i.e., to control or synchronize different machines). Such tasks, in most cases, require millisecond-to-millisecond accuracy and thus require particular attention, since modern operating systems (OS) are not designed to operate in real-time and with such accuracy (Canto, Bufalari, & D'Ausilio, 2011; Chambers & Brown, 2003; MacInnes & Taylor, 2001; Plant & Turner, 2009). Therefore, several different approaches are used, such as programming the experimental I/O task via dedicated and optimized software packages (i.e., E-Prime, Presentation, Psychophysics Toolbox for MATLAB, etc.) or devolving critical tasks to dedicated hardware with internal high precision clocks (i.e., external I/O boards from National Instruments, Measurement Computing, Cambridge Electronics Devices, etc.). In both cases, the solution tends to be expensive (especially for external boards), suboptimal insofar as it relies heavily on the OS's accuracy (this holds for all experimental control software), or unsuited for specific experiments. This latter point is particularly true when testing multiple participants at the same time or in experiments requiring portable or wireless battery-powered set-ups.

In some cases, however, many low-level I/O tasks do not need specific software packages or expensive boards. For example, if the experimenter wants an event to be triggered when an event is detected via some sensors (i.e., touch sensor, force sensor, etc.), it is not necessary to use expensive hardware or software. In fact, simple and cheap microcontroller boards may solve many of these laboratory I/O tasks. Such boards are physical computing platforms based on a simple microcontroller and a development environment for writing software. These devices can be used to develop interactive objects, taking inputs from a variety of switches or sensors and controlling a variety of lights, motors, and other physical outputs. Projects of this kind can usually be stand-alone, or they can communicate with software running on a computer. Such boards have been around for several years and typically offer similar characteristics differing only in processor architecture (ARM, ATMEL, etc.), programming language (C/C++,

A. D'Ausilio (✉)
RBCS - Robotics, Brain and Cognitive Sciences Department,
IIT - The Italian Institute of Technology,
Via Morego,
30-16163 Genova, Italy
e-mail: alessandro.dausilio@iit.it

BASIC, etc.), or other features (i.e., number of I/O channels, presence of analog channels, etc.). Several manufacturers have proposed quite popular solutions such as Parallax Inc., Coridium Corporation, FTDI, Picaxe, Arduino, as well as many others. All of these boards typically cost around 50€. However, programming these boards can be quite complicated, and the user requires at least some basic electronics knowledge. Thus, the main obstacle to widespread use of these boards in psychological and neurophysiological labs is the steep learning curve.

However, Arduino boards (Fig. 1) offer one critical advantage: the open source philosophy (both hardware and software), which capitalizes on the massive nonexpert community that has flourished around the Arduino concept. A very rough estimate of the size of the community can be gleaned from a Google search reporting more than 12 million hits for "arduino." In fact, a large user base and the growing market have shown increasing interest around the Arduino concept. There are hundreds of open-source projects one can use or modify according to specific (experimental) needs. Many web tutorials cover basic programming and electronics issues, and there are active forums for help. Thus, learning to use Arduino boards may be a lot easier than learning to use similar products from other manufacturers. Given the available support from the Arduino community, even researchers with little programming and electronics background should consider using Arduino rather than other similar boards.

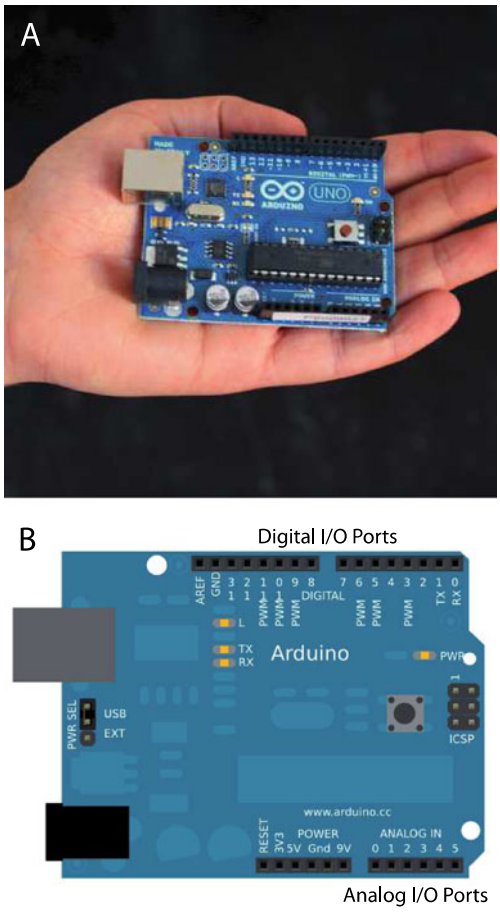

Fig. 1 Arduino board. **a** The Arduino UNO external appearance. Please note how small and portable is this device. **b** A simplified schematic of the digital and analog input/output ports

Arduino hardware consists of an open hardware design with an Atmel AVR processor. Arduino boards can be purchased preassembled, but hardware design information is also available for those willing to build or modify them (further information can be found at http://arduino.cc). Several third-party makers have produced Shields (add-on boards) that are able to extend the basic capabilities of an Arduino (an updated list is at http://shieldlist.org/). Among these shields, it is worth mentioning that the Motor Control Shield allows the control of DC motors and read encoders, the Xbee shield allows multiple Arduino boards to communicate wirelessly, and the Critical Velocity Accelerometer Shield integrates a 3-axis accelerometer. Additionally, third parties (+30) have released several variations based on the Arduino concept. These are companies building boards (typically with better specs or lower price) using the Arduino software.

The software consists of a standard programming language and a firmware that runs on the board. Arduino hardware is programmed using a language that is simplified C++, in a processing-based IDE. The software is then compiled and loaded on board. Arduino boards are compatible also with Flash, Processing, MaxMSP, and MATLAB, and a few lines of code often suffice to enable quite powerful behaviors (see http://arduino.cc/en/Reference/HomePage). The basic programming structure of an Arduino is composed of at least two parts. These are the setup and the loop components. In the set-up, which runs at the beginning and only once to set pin mode or serial communication, the variables are declared. The second part runs in a loop that enables the script to change, to respond, and to control the Arduino board. After declaring variables, controlling the Arduino involves classic control structures (IF, IF…ELSE, FOR, etc.), arithmetic operators (+, -, /, *, etc.), and comparison operators (>, <, etc.) or boolean (AND, OR, etc.). There is also a set of commands for analog and digital read and write such as digitalwrite() or digitalread(). Furthermore, other commands can set temporal delays in milliseconds, perform basic mathematical and trigonometry operations (min/max, absolute value, square root, sine, cosine, etc.), or generate random numbers. For more comprehensive information, please refer to web tutorials or to the official documentation (http://www.arduino.cc/playground/uploads/Main/arduino_notebook_v1-1.pdf).

As a practical example of use, suppose that the experimental participant has to reach and grasp an object, and a researcher wants to trigger another machine (i.e.,

Transcranial Magnetic Stimulation machine) 100 ms after object contact. The typical solution is a touch sensor communicating, via the parallel port, with a software program on a computer (i.e., E-Prime or Presentation). The software detects the state change, sets the delay, and sends a TTL over the parallel port. This solution is irreproachable, except that it requires the intervention of a PC and software that may introduce temporal accuracy problems (Plant & Turner, 2009). Simple I/O tasks, such as the one suggested, require a few lines of code and very little electronics on the Arduino boards (or any other similar products) to process the information without a PC. In this way, it may be possible to avoid temporal accuracy problems, because the code is compiled and runs at full speed without any other interfering process running in the background. In fact, Arduino boards prove extremely useful when acquiring sensor data and/or controlling motors as well as sending/receiving TTL pulses.

Moreover, these low-level I/O tasks may be necessary for unconventional set-ups in which additional hardware, programming skills, and often some creative solutions are usually needed. Consider, for example, this short list of open projects that may be very useful in psychological and neurophysiological laboratories. An easy-to-build and cheap touch sensor (http://www.arduino.cc/playground/Code/CapacitiveSensor) has been devised for use in behavioral research that involves the measurement of reaction times (RTs) of participants reaching and touching objects. Another project shows how to build a photodiode (http://www.arduino.cc/playground/Learning/LEDSensor) that may be used, for instance, to detect onscreen stimuli presentations. Finally, one other project shows how to introduce debouncing capabilities to any input (http://arduino.cc/en/Tutorial/Debounce), which is useful in view of the well-known fact that most button pads offer noisy signals, and that debouncing is a necessary step in order to pick the correct RT.

However, timing accuracy has the highest priority for all experimentalists, even if it is expensive. Here, Arduino boards should in principle be quite accurate, since the scripts are compiled and then loaded on the board memory. Thus, once the compiled script is loaded, it runs without any other OS intervention or communication and thereby avoids delay and accuracy bottlenecks. Accurate timing is ensured by the fact that the embedded microprocessor has its own clock and has only to run the script that has just been loaded. Nevertheless, a thorough testing of timing accuracy must be provided in common experimental settings, and data must be controlled via external reliable and research-grade hardware (Plant, Hammond, & Turner, 2004; Plant, Hammond, & Whitehouse, 2002; De Clercq, Crombez, Buysse, & Roeyers, 2003). In the present study, an Arduino board was verified in a series of common I/O tasks to assess its timing accuracy.

Method and Results

Arduino Board

The Arduino Uno is a microcontroller board based on the ATmega328. It has 14 digital input/output pins, and six of them can be used as Pulse Width Modulation (PWM) outputs. Furthermore, it is provided with six analog inputs, a 16-MHz crystal oscillator, a USB connection, a power jack and an In Circuit Serial Programming header. The Arduino Uno can be powered via USB connection or with an external power supply (i.e., a 9-V battery). The power source is selected automatically. The board can operate on an external supply of 6 to 20 volts. The ATmega328 has 32 KB (with 0.5 KB used for the bootloader). It also has 2 KB of SRAM and 1 KB of EEPROM (which can be read and written with the EEPROM library).

Each of the 14 digital pins on the Uno can be used as an input or output, using *pinMode()*, *digitalWrite()*, and *digitalRead()* functions. They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA, and has an internal pull-up resistor (disconnected by default) of 20–50 kOhms. In addition, some pins have specialized functions: Pins 0 and 1 may be used to receive (RX) and transmit (TX) TTL serial data; Pins 2 and 3 can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value; finally, Pins 3, 5, 6, 9, 10, and 11 provide 8-bit PWM output with the *analogWrite()* function. The Uno has six analog inputs, labeled A0 through A5, each of which provides 10 bits of resolution (1,024 different values). By default, they measure from ground to 5 volts, although is it possible to change the upper end of their range using the AREF pin and the *analogReference()* function.

Accuracy Measurements and Tests

All input and output to the Arduino UNO board were recorded in parallel via an external I/O board equipped with a high-precision clock (CED Power1401, Cambridge Electronic Devices, UK) and were controlled by Signal software (Version 4). Sampling rate was set to 10 KHz. Data were exported in ASCII format and were loaded in MATLAB for further analyses. Six tests were designed to verify timing accuracy with increasingly complex tasks. Appendix A contains the wiring script used in all tests. Figures 2, 4, and 6 contain a pictorial description of each one.

Test 1: oneTTL Test number one was aimed at verifying the basic timing capability of the system. The board had to generate signals 900 ms long with a 100-ms interval (Fig. 2). The compiled script was 1,116 bytes. The test consisted of 1,000 pulses. The analyses consisted of measuring the mean and standard deviation of TTL length and intervals between TTL onsets.

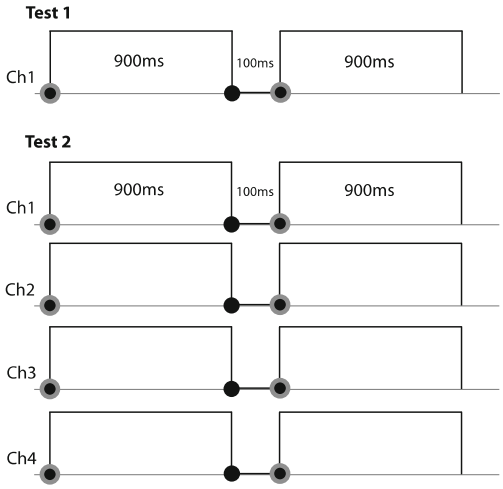

Fig. 2 Tests 1 and 2. The figure shows the TTLs durations and delays in Test 1 and 2. Test 1 has only one channel, whereas Test 2 has four. The analyses measured the TTL durations as well as the delay between TTLs (*black and gray dots*). In Test 2, we also measured the asynchrony error across channels

Test 1 showed a TTL mean length of 0.9004 s (reference value: 900 m) and a delay between TTL onsets of 1.0006 s (reference value: 1,000 ms). Standard deviations were, respectively, 0.000048 and 0.000048, thus showing a negligible error and extremely small variability.

Test 2: fourTTL The second test scaled up the previous test to four synchronous outputs. The aim was to verify whether the use of multiple concurrent channels degrades performance. At the same time, it was possible to measure synchrony errors across channels (Fig. 2). The compiled script was 1,176 bytes. The test consisted of 1,000 pulses in each channel. The analyses consisted in measuring the mean and standard deviation of TTL length and intervals between TTL onsets for each channel. Furthermore, TTL onset asynchrony across channels was measured and reported as cumulative mean error and standard deviation.

Test 2 revealed similarly good results for each channel. The TTLs' lengths were consistently 0.9004 in all four channels (reference value: 900 ms), whereas TTL onset delays were, respectively, 1.0007, 1.0008, 1.0008, and 1.0008 s (reference value: 1,000 ms) in channels 1, 2, 3 and 4. The standard deviations were—for TTL length—0.000043, 0.000043, 0.000042, and 0.000042, and for TTL onset delay, they were 0.000043, 0.000043, and 0.000042 among the four output channels. Thus, increasing the number of channels from two to four did not affect the accuracy of the system. Furthermore, the TTL onset asynchrony error across channels was only 0.000037 s on average, with a standard error of 0.000107, thus showing almost no output synchronization error.

Test 3: overlapTTL The third test used four output channels, as in test number two, but in an asynchronous manner. This test should have further taxed the timing accuracy of the system. Each TTL lasted 1,000 ms and had a 100-ms overlap with the subsequent TTL. The task was more complex here, since it tested multiple outputs with varying onsets and offsets (Fig. 3). The compiled script is 1,248 bytes. The test consisted of 800 pulses (200 for each channel). The analyses consisted of measuring the mean and standard deviation of TTL length in each channel. Furthermore, the mean delay and standard deviation between the first TTL onset and last TTL offset was measured.

Test 3 showed accurate TTLs with a length of 1.0005, 1.0005, 1.0005, and 1.0005 on average (reference value: 1,000 ms), and standard deviations of 0.000045, 0.00005, 0.000049, and 0.000046, respectively, in the four channels. The whole cycle of overlapping TTLs, from channel 1 onset to channel 4 offset, lasted 3.7019 (reference value: 3.7 s; see Fig. 4) with a standard deviation of 0.000049, showing a small but extremely stable (low standard deviation) delay of 2 ms.

Test 4: LOOPoverlapTTL The fourth test was similar to the previous one except for the programming structure. Here, we used a more compact and efficient *FOR* structure to test the impact of programming style differences in timing performances. The compiled script was 1,212 bytes. The test consisted of 800 pulses, 200 for each channel. The analyses were the same as in Test 3.

Test 4 replicated the previous test by using a different programming style. Here, the test showed accurate TTLs with lengths of 1.0005, 1.0005, 1.0005, and 1.0005, on average (reference value: 1,000 ms), and standard deviations of 0.000028, 0.000036, 0.00003, and 0.00002 in the four channels. The whole cycle of overlapping TTLs, from channel 1 onset to channel 4 offset, lasted 3.7019 (reference value: 3.7 s; see Fig. 4), with a standard deviation of 0.00005. Here, as in the previous test, there was a small but extremely stable (low standard deviation) delay of less than 2 ms.

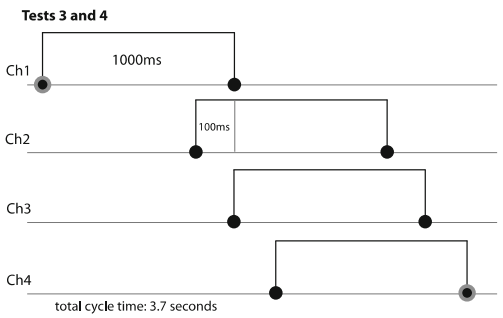

Fig. 3 Tests 3 and 4. The figure shows the TTLs durations and delays in Tests 3 and 4. Test 3 differed only in the programming style. The analyses measured the TTL durations as well as the cycle duration (*black and gray dots*)

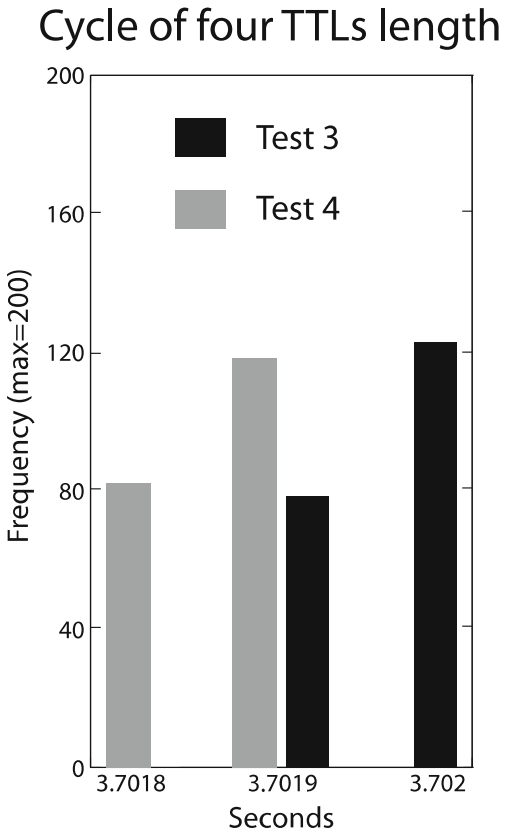

Fig. 4 Results of Tests 3 and 4 This histogram shows the length of the whole cycle of overlapping TTLs, from channel 1 onset to channel 4 offset (ground truth: 3.7 s), for Test 3 (black) and Test 4 (gray)

Test 5: ReadButtonWriteTTL In Test 5, we used a more realistic scenario in which a participant's button press started a cascade of timed events. This test was also important because it simulated a typical I/O task in which the output was associated to an input. In fact, 200 ms after the button was pressed, a 100-ms TTL was delivered by an output channel (Fig. 5a). The compiled script was 1,294 bytes. Each test was repeated 1,000 times. The analyses consisted in measuring the mean and standard deviation of TTL delay after the input signal is received.

In Test 5, the TTLs onset delay with respect to the input TTLs onset was 0.2001 (reference value: 200 ms), with a standard deviation below the measurement accuracy of our acquisition device (1.1107786e-15 s). The results of this test showed essentially perfect accuracy**:** All repetitions resulted in exactly the same delay in sending a timed TTL given an external trigger.

Test 6: Read_Write_different_TTLs The script in Test 6 read from two parallel input channels. Input TTLs were randomly presented in only one of the two channels at each trial. The script waited 200 ms and delivered a 100-ms TTL if the input was on channel 1, whereas if the input was on channel 2, the waiting time was 100 ms and the TTL was 200 ms long (Fig. 5b). The compiled script was 1,372 bytes. Each test was repeated 1,000 times. The analyses consisted in measuring the mean and standard deviation of TTL delay and length.

The results show a very high degree of accuracy. When the input was on channel 1, the TTL onset was 0.2002 s with a standard deviation of 6.1123e-16 and lasted 0.100 with a standard deviation of 9.0296e-16 (ground truth: 0.200 and 0.100). When the input was on channel 2, the TTL onset was 0.1001 s with a standard deviation of 1.8337e-15, and lasted 0.2001 with a standard deviation of 1.8337e-15 (ground truth: 0.100 and 0.200).

Test 7: ReadConditional_AND_WriteTTL In Test 7, we verified the accuracy of the Boolean operator "AND." The script read from two digital inputs separately, as if they were two independent buttons. When both were pressed at the same time, it waited 100 ms and deliverd a 100-ms TTL (Fig. 5c). The two simulated buttons (two digital inputs) were 100 ms long with a 50-ms overlap. The compiled script was 1,318 bytes. Each test was repeated 1,000 times. The analyses consisted in measuring the mean and standard deviation of TTL delay after the condition was met (both digital inputs were HIGH).

In Test 7, the output TTLs were generated when both input channels were in the logical HIGH state. Here, the mean delay between the moment the condition was met and the output TTL onset was 0.1997 s (reference value: 200 ms; see Fig. 6), with a standard error of 6.7185e-15. Thus, it appears that accuracy was nearly perfect.

Test 8: Read_four_Conditional_AND_WriteTTL In Test 8, we verified the accuracy of the Boolean operator "AND" on four inputs. The script read from four digital inputs separately, as if they were four independent signals (i.e., sensors or buttons), and when all of them were set to HIGH at the same time, the script waited 100 ms and delivered a 100-ms TTL (Fig. 5d). The compiled script was 1,318 bytes. Each test was repeated 1,000 times. The analyses consisted in measuring the mean and standard deviation of TTL delay after the condition was met (all digital inputs were HIGH).

In Test 8, the output TTLs were generated when both input channels were in the logical HIGH state. Here, the mean delay between the moment the condition was met and the output TTL onset was 0.3017 s (reference value: 300 ms; see Fig. 6), with a standard deviation of 0.0024. Therefore, it appears that this test was the most variable, with a larger standard deviation, as is demonstrated better by the large delta between the maximal and the minimal values, 0.3121 and 0.301 s, respectively.

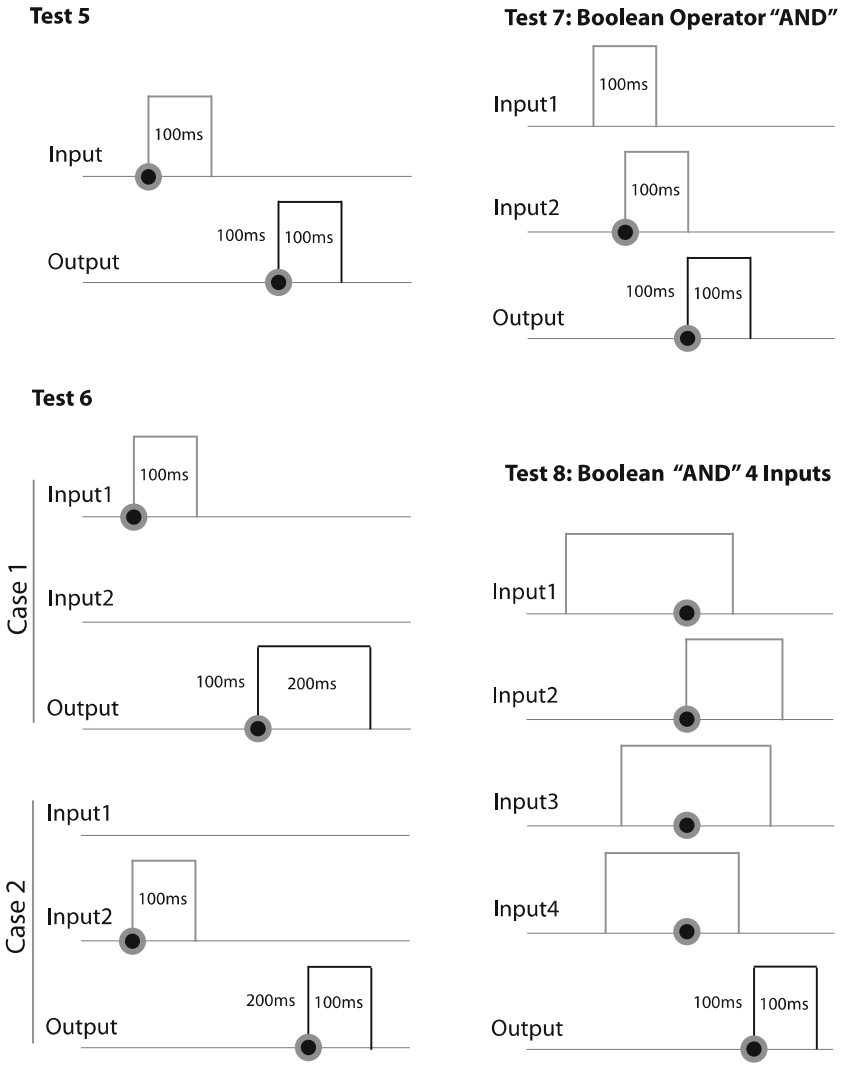

Fig. 5 Tests 5, 6, 7, and 8. The figure shows the TTL durations and delays in Tests 5, 6, 7, and 8. Gray square waves were generated by an external source and used as input signals, whereas the Arduino UNO generated output TTLs. In Test 5, one input triggered the output pulse generation. In Test 6, the Arduino generated either a long output TTL (200 ms) with a short delay (100 ms), or a short TTL (100 ms) with a long delay (200 ms), depending on which input signal was delivered. In Test 7, the Arduino generated the output TTL when two input channels both received a logical HIGH signal (Boolean AND). In Test 8, the Arduino generated the output TTL when four input channels received a logical HIGH signal (Boolean AND). The analyses measured the delay between the moment the condition was met and the actual TTL generation (*black and gray dots*)

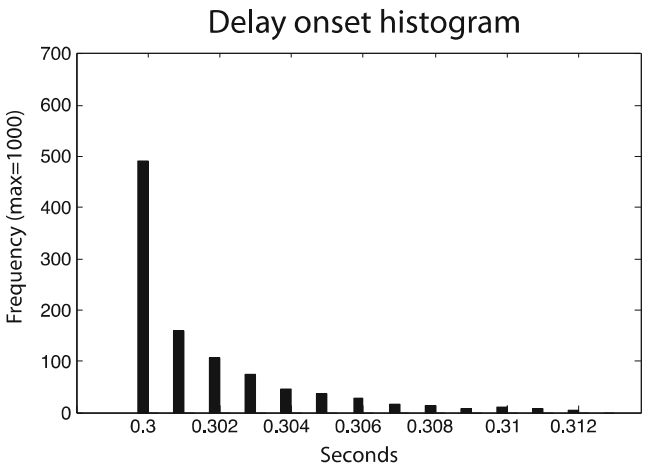

Fig. 6 Results of Test 8. This histogram shows the delay between the moment when the condition was met and the delivery of the output TTL onset (ground truth: 300 ms)

Discussion and Conclusion

According to the tests shown in the present article, the Arduino UNO is an accurate platform for a series of lab settings. The tests reported presently indicate good reliability in controlling TTL length and delays both in the single channel and in four-channel experiments (Tests 1, 2, 3, and 4). The results from Tests 1 and 2 indicate that the synchrony across channels is accurate and that scaling up the number of channels does not affect accuracy. However, one important factor in degrading or improving performance can be the programming style. This is true for all programming languages and platforms. The comparison between Tests 3 and 4 shows negligible differences (Fig. 5), from a practical point of view, both in accuracy and in variability. Nevertheless, the use of a more efficient programming structure (a "FOR" loop instead of a list of commands) in Test 4 enabled only a small (<1 ms) reduction in variability with respect to Test 3. Generally speaking, simple tasks are not affected by programming

style, and Arduino boards proved robust enough to be accurate even with poor programming. It is beyond the scope of the present research to investigate why such a (very small) difference was observed. However, this study is an important proof of concept regarding the possible impact of programming strategies in adjusting timing issues—especially when dealing with complex and more realistic scripts.

On the other hand, Tests 5, 6, 7, and 8 were typical input/ output tasks in which an output is produced when a condition is met. All of these tests simulated the typical situation in which an input from buttons or sensors must be checked before releasing an output signal. Specifically, Test 5 consisted of a TTL delivered after the detection of an input signal. This test showed an impressive accuracy: 1,000 trials led to the same exact value. Test 6, in contrast, produced different delays and TTL lengths according to the input provided. The results demonstrated an extremely high degree of accuracy. In Tests 7 and 8, a Boolean operator (AND) on two or four input channels was introduced. These two tests showed mixed results, and, in fact, accuracy was reduced when using the Boolean operator on four input channels as opposed to two. Nevertheless, the results still indicate a fairly high degree of reliability (standard deviation around 2 ms). Taken all together, these tests showed that Arduino boards can be quite accurate in simple I/O tasks.

In conclusion, the critical advantage of the Arduino concept is mostly related to the large open-source community that has flourished. There are forums, mailing lists, hardware schematics, and code freely available for many projects. Moreover, the Arduino hardware is open source, and many clones or special-purpose boards have been marketed. The large and lively user base has produced interesting projects that can be readily adapted to typical experimental needs of psychological and neurophysiological laboratories.

Importantly, these boards cost a few tens of Euros, which is one order of magnitude less than entry-level I/O cards. Therefore, Arduino can be a very cheap alternative to expensive hardware. However, it is difficult to define a threshold beyond which expensive research-grade equipment is preferable to an Arduino-like board. Laboratories with expertise in expensive equipment and/or software should probably avoid spending time learning a new platform. In fact, there is no doubt that any research-grade I/O board will be more powerful and accurate than an Arduino. However, there are a few interesting applications in which an Arduino is the only viable solution—for example, when running multiple participants at the same time with the same set-up. Suppose a researcher wants to run a test in different places with replicated set-ups. In this case, each standard set-up (professional I/O board, PC or Laptop, etc.) is very expensive and takes time to be assembled. An Arduino-based set-up is extremely portable (no need for a PC or Laptop), cheap, and easy to deploy in large numbers. Furthermore, the Arduino platform can be powered by standard 9-V batteries opening to a series of wireless or wearable solutions, which is impossible with standard I/O boards. Therefore, Arduinos may not necessarily replace existing platforms in traditional settings. Rather, thanks to the low price, small form factor, and ease of use, they can be used in simple I/O tasks or in a whole new set of scenarios. The critical advantage of the Arduino concept is that the boards can be used by people with modest programming and electronic backgrounds. In brief, an Arduino-like board might be an easy and cheap multipurpose tool for many labs.

Author Note I would like to thank Eleonora Bartoli, Laura Maffongelli and John Michael for the help in revising the manuscript.

Appendix A

```
// Arduino Accuracy Tests
   // Alessandro D'Ausilio
   // 13 April 2011 - Revised 5 September 2011
   int pin[] = {2,3,4,5,6,7,8,9};
   int count;
   void setup()
   {
   pinMode(pin[1],OUTPUT);
   pinMode(pin[2],OUTPUT);
   pinMode(pin[3],OUTPUT);
   pinMode(pin[4],OUTPUT);
   pinMode(pin[5],INPUT);
   pinMode(pin[6],INPUT);
   pinMode(pin[7],INPUT);
   pinMode(pin[8],INPUT);
   }
   void loop()
   {
   // remove double slashes to execute the script //
   //oneTTL (); // Test 1
   //fourTTL (); // Test 2
   //overlapTTL (); // Test 3
   //LOOPoverlapTTL (); // Test 4
   //ReadButtonWriteTTL (); // Test 5
   //ReadConditional_AND_WriteTTL (); // Test 6
   //ReadFour_AND_WriteTTL(); // Test 7
   //ReadTwo_WriteDiffTTL(); // Test 8
   }
   // Test 1 - one TTL 900 ms long with a delay of 100 ms
//
   void oneTTL (){
```

```
int delayTime = 100;
int durationTime = 900;
digitalWrite (pin[1], HIGH);
delay(durationTime);
digitalWrite (pin[1], LOW);
delay(delayTime);
}
// Test 2 - four synchronous TTLs 900 ms long with a delay of 100 ms //
void fourTTL (){
int delayTime = 100;
int durationTime = 900;
digitalWrite (pin[1], HIGH);
digitalWrite (pin[2], HIGH);
digitalWrite (pin[3], HIGH);
digitalWrite (pin[4], HIGH);
delay(durationTime);
digitalWrite (pin[1], LOW);
digitalWrite (pin[2], LOW);
digitalWrite (pin[3], LOW);
digitalWrite (pin[4], LOW);
delay(delayTime);
}
// Test 3 - four asynchronous TTLs 1000 ms long, each with a delay of 100 ms respect to the previous one //
void overlapTTL (){
int delayTime = 1000;
int overlapTime = 100;
digitalWrite(pin[1], HIGH);
delay(delayTime - overlapTime);
digitalWrite(pin[2], HIGH);
delay(overlapTime);
digitalWrite(pin[1], LOW);
delay(delayTime - overlapTime - overlapTime);
digitalWrite(pin[3], HIGH);
delay(overlapTime);
digitalWrite(pin[2], LOW);
delay(delayTime - overlapTime - overlapTime);
digitalWrite(pin[4], HIGH);
delay(overlapTime);
digitalWrite(pin[3], LOW);
delay(delayTime - overlapTime);
digitalWrite(pin[4], LOW);
delay(delayTime);
}
// Test 4 - four asynchronous TTLs 1 s long, each with a delay of 100 ms respect to the previous one, using a FOR structure //
void LOOPoverlapTTL (){
int delayTime = 1000;
int overlapTime = 100;
for(int j = 1; j < = 4; j++){
digitalWrite(pin[j], HIGH);
if(j == 1) {
delay(delayTime-overlapTime);
}
else if (j > 1){
delay(overlapTime);
digitalWrite(pin[j-1], LOW);
delay(delayTime-overlapTime-overlapTime);
}
if (j==4){
delay(overlapTime);
digitalWrite(pin[j], LOW);
}
}
delay(delayTime);
}
// Test 5 - read a button press, wait 100 ms and then write a 100 ms TTL
void ReadButtonWriteTTL(){
int delayTime = 100;
int DigitalValue = digitalRead(pin[5]);
if (DigitalValue == HIGH) {
delay(2*delayTime);
digitalWrite(pin[1], HIGH);
delay(delayTime);
digitalWrite(pin[1], LOW);
}
}
// Test 6 - read two button presses. If button one is pressed, wait 100 ms and then write a 100 ms TTL; If button two is pressed, wait 100 ms and then write a 200 ms TTL;
void ReadTwo_WriteDiffTTL(){
int delayTime = 100;
int DigitalValue1;
int DigitalValue2;
DigitalValue1 = digitalRead(pin[5]);
DigitalValue2 = digitalRead(pin[6]);
if (DigitalValue1 == HIGH) {
delay(delayTime*2);
digitalWrite(pin[1], HIGH);
delay(delayTime);
digitalWrite(pin[1], LOW);
}
if (DigitalValue2 == HIGH) {
delay(delayTime);
digitalWrite(pin[1], HIGH);
delay(delayTime*2);
digitalWrite(pin[1], LOW);
}
}
// Test 7 - read two button press, when both are pressed, wait 100 ms and then write a 100 ms TTL
void ReadConditional_AND_WriteTTL(){
```

```
int delayTime = 100;
int DigitalValue1;
int DigitalValue2;
DigitalValue1 = digitalRead(pin[5]);
DigitalValue2 = digitalRead(pin[6]);
if (DigitalValue1 == HIGH && DigitalValue2 == HIGH) {
delay(delayTime);
digitalWrite(pin[1], HIGH);
delay(delayTime);
digitalWrite(pin[1], LOW);
}
}
// Test 8 - read four button presses, when all of them pressed at the same time, wait 100 ms and then write a 100 ms TTL
void ReadFour_AND_WriteTTL(){
int delayTime = 100;
int DigitalValue1;
int DigitalValue2;
int DigitalValue3;
int DigitalValue4;
DigitalValue1 = digitalRead(pin[5]);
DigitalValue2 = digitalRead(pin[6]);
DigitalValue3 = digitalRead(pin[7]);
DigitalValue4 = digitalRead(pin[8]);
if (DigitalValue1 == HIGH && DigitalValue2 == HIGH && DigitalValue3 == HIGH && DigitalValue4 == HIGH) {
delay(delayTime);
digitalWrite(pin[1], HIGH);
delay(delayTime);
digitalWrite(pin[1], LOW);
}
}
```

References

Canto, R., Bufalari, I., & D'Ausilio, A. (2011). A convenient and accurate parallel Input/Output USB device for E-Prime. *Behavioral Research Methods, 43,* 292–296.

Chambers, C. D., & Brown, M. (2003). Timing accuracy under Microsoft Windows revealed through external chronometry. *Behavior Research Methods, Instruments, & Computers, 35,* 96–108.

De Clercq, A., Crombez, G., Buysse, A., & Roeyers, H. (2003). A simple and sensitive method to measure timing accuracy. *Behavior Research Methods, Instruments, & Computers, 35,* 109–115.

MacInnes, W. J., & Taylor, T. L. (2001). Millisecond timing accuracy on PCs and Macs. *Behavior Research Methods, Instruments, & Computers, 33,* 174–178.

Plant, R. R., Hammond, N., & Turner, G. (2004). Self-validating presentation and response timing in cognitive paradigms: How and why? *Behavior Research Methods, Instruments, & Computers, 36,* 291–303.

Plant, R. R., Hammond, N., & Whitehouse, T. (2002). Toward an experimental timing standards lab: Benchmarking precision in the real world. *Behavior Research Methods, Instruments, & Computers, 34,* 218–226.

Plant, R. R., & Turner, G. (2009). Millisecond precision psychological research in a world of commodity computers: New hardware, new problems? *Behavior Research Methods, 41,* 598–614.