Software Library for Fast Digital Input and Output for the Arduino Platform

JAN DOLINAY, PETR DOSTÁLEK, VLADIMÍR VAŠEK
Department of Automation and Control Engineering
Tomas Bata University in Zlin
nám. T. G. Masaryka 5555, 76001 Zlín
CZECH REPUBLIC
dolinay@fai.utb.cz

Abstract: - This article presents software library for the Arduino platform which significantly improves the speed of the functions for digital input and output. This allows the users to apply these functions in whole range of applications, without being forced to resort to direct register access or various 3rd party libraries when the standard Arduino functions are too slow for given application. The method used in this library is applicable also to other libraries which aim to abstract the access to general purpose pins of a microcontroller.

Key-Words: - Arduino, AVR, digitalRead , digitalWrite, embedded system, pin abstraction, software library

1 Introduction

Arduino is described as a prototyping platform which consists of a board with microcontroller, program libraries for this microcontroller and an integrated development environment (IDE) for writing the programs [1]. Started in 2005 to provide cheap microcontroller platform for student projects, Arduino gained enormous popularity in the do-it-yourself community and made its way into the educational institutions as well [2], [3]. There are many books focused on this platform available [4], [5] and it is even used in research projects as a cheap and easy-to-use platform for controlling various apparatus or performing experiments in many areas. For example, [6] describes the use of Arduino for controlling microfluidic devices, in [7] extensive timing tests of Arduino programs are described with respect to use of this platform in psychological and neurophysiological experiments and in [8] the platform is used for studying lymphatic biomechanics in vitro.

Analyzing the reasons for the popularity of this platform is beyond the scope of this article. However, we believe it originates from the ease of use combined with the low price of the hardware and open source license, which lead to sharply increasing number of users worldwide.

As it is clear with the hindsight, there was great demand for cheap, easy to use microcontroller platform, which would allow people without deep knowledge of electronics and programming create devices which interact with the outside world. In the "pre-Arduino" era, the efforts of many such users were fragmented among the various manufacturer-specific microcontroller boards and development environments. Once these users found a common platform (the Arduino), the community started growing quickly, attracting more and more users to the platform and generating large collection of libraries and example programs. Nowadays if one needs to work with any sensor, it is very likely there will be code examples and wiring diagrams for the Arduino for this sensor; we can say that Arduino became *de facto* prototyping standard.

Also educational institutions naturally follow this trend and begin to use Arduino for teaching embedded system design, programming courses and similar subjects [3], [9], [10]. It is possible that it will soon become *de facto* standard for this area also. Our perception is that there is increasing number of teachers who promote Arduino as educational tool opposed by a group of teachers who consider it inappropriate for serious education of embedded systems programming.

One of the main arguments for Arduino in education seems to be the ease of use, which helps to motivate students [2]. Also, the big community of users makes learning easier; students can obtain code examples and answers to their questions quickly. As stated in [2], including Arduino in the course allowed students to create more interesting projects because they were not limited by some of the challenges of working with more advanced development environments.

The opponents argue that there is a big gap between Arduino and the real embedded world which prevents most users from migrating to some real IDE and real-world projects. The Arduino IDE is very simple and there is no debugger. Another argument can be the quality of the microcontroller libraries provided with the platform. Interesting discussion on this can be found in [11] (including the reader's opinions on the article).

In our view the suitability of Arduino for any course depends on the aim of the course in the first place. For example, for a future embedded programmer, learning the high-level Arduino functions will not be sufficient; such a student should gain solid understanding of the device registers and skills in working with development environment including debugging the program. On the other hand, for students who are not supposed to make their living by embedded programming the platform may provide all that is needed.

At our faculty we have two different courses which deal with embedded systems – one is intended for students focused on information technologies and programming, the other is for students focused on security technologies, where programming is a marginal subject. For the latter, we plan to start using Arduino in the upcoming semester, replacing boards with Freescale HCS08 microcontroller used so far. The reasons discussed above apply to this course – it seems that the "barrier" of learning C language in a full-featured IDE is too high for these students. For the programming-focused course we will continue to use more traditional approach without Arduino, but in the oncoming modernization of the equipment we chose a board with Arduino compatible layout, with ARM based microcontroller), so the hardware will be "Arduino-friendly" even in this course. It should be mentioned that a trend can be seen in the evaluation boards offered by many microcontroller vendors to follow Arduino pinout, allowing the users to utilize Arduino expansion boards (shields). This means one can choose from many boards with different microcontrollers (and different development environments) for the course and still have the hardware compatible with other courses, institutions and the do-it-yourself community.

For both the scientific and educational use it is important to keep in mind that despite the advantages and popularity, the Arduino platform is not flawless. There is quite a big room for improvement in the microcontroller software libraries, although it is not easily visible to the typical Arduino user, who is usually not an experienced embedded developer.

In this article we want to present our software library for Arduino which significantly improves the speed of the functions for digital input and output (I/O). The digital I/O is one of the areas where the standard Arduino library leaves room for improvement. The default implementation of the digital I/O functions is not efficient and writing or reading pins is much slower than it could be. Although in most situations this is not a problem and the speed of I/O operations is more than sufficient, there are applications where the speed is critical and the default functions cannot be used. In such applications, the solution typically is using direct register access, by which the program loses the portability between different Arduino boards and the implementation becomes very vendor (hardware) specific. Another option is using 3rd party libraries for fast digital I/O, but these libraries typically improve the speed only in if the pin number is known at compile-time (it is a constant). In our solution we reach the goal of maximum speed for constant pin numbers but also improve the speed for non-constant pin numbers. This makes it possible to use the same functions in whole range of applications, without the need to resort to direct register access or various 3rd party libraries when the standard Arduino functions are too slow for the task. The method used in our library is applicable also to other libraries which aim to abstract the access to general purpose pins of a microcontroller.

2 Problem Formulation

Arduino board interacts with the outside world by means of input and output pins. The programming model identifies these pins by a number, starting from 0 up to the maximum available value, depending on the board variant.

For digital I/O, Arduino library defines functions *digitalRead* and *digitalWrite* which read (or write to) a pin. These functions take pin number as input argument. For example, the following line of code will set pin 13 to logical 1 level (high):

```
digitalWrite(13, HIGH);
```

However, in the actual microcontroller, which is the "brain" of the Arduino board, pins are not organized into flat space. Instead, pins are grouped into ports, each port having typically 8 pins. The ports are named by capital letters A, B, C, etc. Therefore, we have pins 0 thru 7 on port A, pins 0 thru 7 on port B and so forth.

In a traditional embedded program one controls the pins by setting and clearing bits in special registers

which control each port separately. This low-level programming is (arguably) hard to understand for beginners and one of the reasons for Arduino's success is probably the fact, that it hides this and other hardware related details behind easy-to-use functions, such as the digitalWrite. Nevertheless, the Arduino software library has to translate the linear pin number to the corresponding port and pin. The efficiency of this translation then determines the performance of the library. And what seems like an easy task proves to be a tricky problem.

2.1 Implementation used in Arduino

Arduino library uses the following approach to translate the Arduino pin number to microcontroller pin and port. There is an array stored in memory which maps pin number to port number. The number of elements in this array is the same as the number of pins available in given board.
Figure 1 shows part of this array for Arduino Uno board.

```
const uint8_t PROGMEM digital_pin_to_port_PGM[] = {
        PD, /* 0 */
        PD,
        PD,
        PD,
        PD,
        PD,
        PD,
        PD,
        PB, /* 8 */
        PB,
        ...
```

Fig. 1 Code which defines the mapping of pin number to port number

The PD and PB symbols seen in figure 1 are constants which represent zero-based port number (0 for port A, 1 for port B, etc).
As follows from the definition seen in figure 1, Arduino pins 0 through 7 are located on port D, pins from 8 up are on port B.
For mapping the port number to the actual address of the MCU register which controls this port there is a second array, shown in figure 2.
In fact, there are several similar arrays for the various registers involved in controlling digital I/O. This is also the reason why the pin-to-register mapping is two-staged (pin-to-register number; register number-to-register address).

```
const uint16_t PROGMEM port_to_output_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        (uint16_t) &PORTB,
        (uint16_t) &PORTC,
        (uint16_t) &PORTD,
};
```

Fig. 2 Code which defines the mapping of port number to port register address

If the mapping was simple pin-to-register address, then each of the arrays with register addresses would have to contain as many elements as there are available pins on the board; which can be a high number. For example, there are over 50 pins in Arduino Mega board. Such solution would consume too much memory.
As already mentioned, functions for digital I/O, such as digitalWrite, receive pin number as input argument. It is then used as the index into the first array to obtain register number for given pin. This register number is consequently used as the index into the second array to obtain the address of the register.
The arrays are saved in program memory (flash) rather than in RAM, because the amount of RAM is typically very limited in microcontrollers. The speed of reading from flash memory is lower than from the RAM, but not significantly. For example, the AVR MCU used in Arduino needs 3 clock cycles to read from flash (LPM instruction) and it needs 2 clock cycles to read from RAM (LDS instruction).

2.2 Existing solutions for faster speed

There are ongoing discussions about the digital I/O efficiency in Arduino, dating for many years back in the community, for example in [12] and [13]. These discussions seem to concentrate on two possible approaches. One is the approach used currently in Arduino library, which results in slow I/O operations no matter if the pin number is given as a constant known when the program is compiled (compile-time constant) or as a variable. The second approach uses preprocessor macros and results in a fast digital read and write if the pin number is compile-time constant and slow code if the pin

number is a variable. This second version is used in the Wiring framework [14], which is the predecessor of Arduino. It provides very fast operation for constant pins (0.2 microseconds) but is slow (5 microseconds) for pin numbers stored in variables. The first option, currently adopted in Arduino, is slow (5 microseconds) in both cases.

From this short summary it would seem that the latter method is better, but the implementation is more complicated and also the effort needed to switch the Arduino software from existing approach to a different one would be considerable.

While the discussions concentrate on arguments for or against one of the above mentioned approaches, neither of these methods provides fast results for pins stored in variables. We realized there is a third method which could solve this problem. This method is based on encoding the information about pin and port mapping into the pin name–which we used and which proved to provide fast code both for constant and non-constant pin numbers. The implementation and performance of our library based on this method is described below.

3 Problem Solution

In general, the translation of linear pin number used in Arduino to the port and pin as used in the actual hardware requires time. This time can be spent during the execution of the program (in the runtime), as it is in current Arduino implementation, or it can be spent during compilation of the program, as it is in the Wiring implementation in case the pin number is a compile-time constant. Obviously, the latter is better; the program can run much faster if the relatively low-performance MCU processor is not required to perform any calculations while writing or reading pins. However, this latter approach does not improve the situation for programs which do not use the pin as a compile-time constant but as a variable. In such case the translation must be again performed by the MCU in runtime.

In our implementation we aimed to minimize the need for computing power in the MCU and the result of experiments with various options showed that the best choice is to encode the port and pin information into the input argument of the digital I/O functions. We call this input argument *pin code*.

Apparently, if the input argument of the I/O functions is a simple integer 0 thru N, as it is in the Arduino library, it is not possible to encode any information into it. There needs to be one level of indirection added so that the input argument for the functions is not the pin number directly, but some symbolic name which can internally represent any value.

The native input argument for our I/O functions is therefore a symbolic name of the pin, such as DP1. Even though it should be very easy for the programmers to use pre-defined names for pins instead of simple integers, for example, writing

digitalWrite(DP1, HIGH)

instead of

digitalWrite(1, HIGH),

we realize that it could be off-putting for some users. Therefore we created also I/O functions which take pin number directly as it is common in the Arduino library. This additional layer performs translation of the simple pin number into the pin code used in the lower level. This translation is performed in the runtime, but is very simple (mapping one integer to another), so the resulting speed is still considerably higher than in the standard Arduino implementation.

Our functions are organized into library called *DIO2* and described in the following section.

3.1 Implementation of the DIO2 library

As mentioned earlier, we encode the port address and pin number into a single number, called pin code. Our functions for digital I/O receive this pin code from the caller and parse it to obtain the appropriate port register address and pin mask. Describing details of the implementation in C language is beyond the scope of this article. Let us just outline the basic principles here.

In general, the pin code is an integer, but it is more suitable to use enum data type in C language and define the available pins for given board this way. Required size of the pin code depends on the MCU architecture; it needs to hold the address of an I/O registers of the MCU (or suitable offset from some common base address) and also the pin bit mask. For the 8-bit Atmel AVR microcontrollers used in Arduino Uno and Mega boards, 16-bit size of pin code is sufficient. For 32-bit MCUs the size could be 32-bits without affecting the performance.

Figure 3 shows the structure of the pin code for the AVR microcontroller:

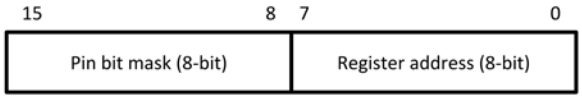

Fig. 3 Structure of the pin code used in our DIO2 library

As can be seen in the figure, the lower byte stores the address of the port register and the upper byte holds the bit mask for the pin within this register.

For example, Arduino pin 13 (where LED is connected on the standard board) is actually pin 5 on port B in the microcontroller namespace (pin PB5). The address of the data register for port B is 25 hexadecimal (25h). The bit mask for pin 5 is 00100000 in binary, which is 20h. The resulting pin code for pin 13 is therefore: 2025h.

It is worth mentioning that the decision to store pin bit mask rather than pin number in the pin code is based on the fact that the seemingly simple operation of bit shift for obtaining pin mask from pin number is costly in terms of CPU time if the pin number is not known at compile-time. This is because it results in a loop with number of repetitions (shifts) depending on the (variable) pin number.

As stated earlier, there is also additional layer in the library which allows using the new functions in the same way as the original ones, i.e. with simple integer pin numbers. At this level, pin number is converted into pin code using array in program memory where the index of the element is the pin number and the value stored in this element is the pin code. This decreases the speed, but the functions are still significantly faster than the original I/O functions in the Arduino library.

3.2 Speed comparison

To evaluate the speed of our library, we performed speed tests with the digitalWrite function from our library and with the standard Arduino implementation of this function. These tests were carried out with the standard Arduino Uno board and the Arduino Mega board. These are the only two Arduino variants for which the library is currently available.

It should be mentioned that the tests are intended to show the speed of the two implementations relative to each other rather than to measure the exact time it takes to execute the functions. As we are working with deterministic digital computer system, the speed of the functions would be accurately given in number of clock cycles and this number of clock cycles would be obtained by analyzing the resulting assembly code for the functions. By measuring absolute time we would be actually measuring the variation of the frequency of the clock generator and other components of the system, which will dependent on many external factors such as input voltage, temperature, etc. Measuring the execution time is therefore not a good method if the purpose was to provide precise timing information, but it is easy to perform and for the purpose of comparison, it is acceptable.

The comparison of digitalWrite function in our DIO2 library and the standard Arduino implementation is given in Figure 4. The results were obtained using simple program which toggles output pin at full speed and measuring the resulting signal on the pin with oscilloscope Owon HDS1022M. The graph shows how long it takes to execute the digitalWrite function once, i.e. set an output pin high or low. The times on the horizontal axis are given in microseconds.

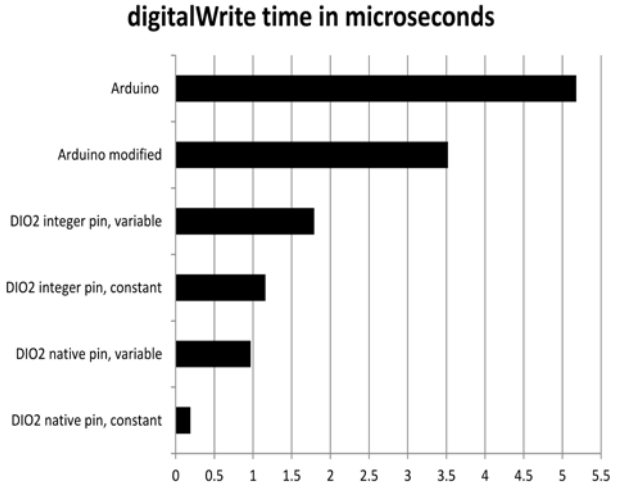

Fig. 4 Comparison of the time it takes to perform single digitalWrite

The meaning of the labels used in the figure is as follows:

- Arduino – standard implementation as it is included in the Arduino library.
- Arduino modified – standard implementation in Arduino but without code which checks whether given pin is used by a timer. This check is part of the standard Arduino digitalWrite implementation even though it seems unnecessary. It is not implemented in our library, so we measured the speed of the standard Arduino implementation also without this check to provide fair comparison.
- DIO2 integer pin – our functions with the additional layer which allows calling these functions with simple pin number in the same way as the standard Arduino functions.
- DIO2 native pin – our functions without the additional layer; these functions take pin code instead of simple integer.

For the DIO2 functions there are always two values given: one for the case when the pin is known at compile-time (constant) and the other for pin not known at compile-time (variable). For the Arduino version only one value is given, because the speed is the same for both compile-time constant pin numbers and pin numbers stored in variables.

As can be seen in the figure, it takes about 5 microseconds to change the output pin level with the default Arduino implementation. With our implementation, it is possible to achieve less than 0.2 microseconds if constant pin code is used – which is possible in most applications. In this case the code of digitalWrite function translates into single processor instruction after compilation so the function is as fast as if one used direct register manipulation.The program used for testing is shown in figure 5.

3.3 Use of the library

The library offers the same digital I/O functions as the standard Arduino library, with "2" added to the name: digitalRead2, digitalWrite2 and pinMode2.

```
uint8_t pin = 8;
//const uint8_t pin = 8;
//GPIO_pin_t fast_pin = DP8;
//const GPIO_pin_t fast_pin = DP8;

void setup() {
  // put your setup code here, to run once:
  pinMode(pin, OUTPUT);
  //pinMode2(pin, OUTPUT);
  //pinMode2f(fast_pin, OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  while(1)
  {
    // Default arduino
    digitalWrite(pin, HIGH);
    digitalWrite(pin, LOW);

    // DIO2 with integer pin
    //digitalWrite2(pin, HIGH);
    //digitalWrite2(pin, LOW);

    // DIO2 with native pin (pin code)
    //digitalWrite2f(fast_pin, HIGH);
    //digitalWrite2f(fast_pin, LOW);
  }
}
```

Fig. 5 Program used for comparison of the speed of the DIO2 and standard Arduino digitalWrite.

These functions are fully compatible with Arduino version, taking pin number as input argument. The faster, native functions which take pin code as input argument have "f" added to the name: digitalRead2f, and so on.

Typical way of extending the Arduino software is using the Arduino library mechanism available in the IDE. Therefore the DIO2 is available as a library, which the user can import into his or her Arduino installation. However, there is one extra step required – a file with definition of the pin codes for each board needs to be copied into appropriate location in the Arduino directory structure. This is because in the Arduino library mechanism there is no compile-time information about the board selected and no way to place board-specific files in some subfolder within the library's folder. Alternatively, the user can copy all the DIO2-library files directly into the Arduino directory structure and bypass the library mechanism.

The usage of the DIO2 function in user program is the same as with the original functions, only the names are slightly different, as described

above. When using the faster (native) functions with pin code, the variables for storing pins must be declared with the proper type instead of the int type used normally in Arduino. The following figure shows program which blinks LED on pin 13 using the faster functions:

```
#include "arduino2.h" // include the fast I/O 2 functions
// The I/O 2 functions use special data type for pin – pin code.
// Pin codes, such as DP13 are defined in pins2_arduino.h
const GPIO_pin_t led_pin = DP13;
void setup() {
  pinMode2f(led_pin, OUTPUT);
}

void loop() {
  digitalWrite2f(led_pin, HIGH);
  delay(1000);
  digitalWrite2f(led_pin, LOW);
  delay(1000);
}
```

Fig. 6 Example program using the presented DIO2 library

4 Conclusion

In this article we described our library for the Arduino platform which allows performing operations on digital pins faster than the standard version shipped with the platform, while remaining easy to use and portable. It is currently implemented for three Arduino variants, Uno, Mega and micro, but porting to other boards is simple.

The library makes it possible to use the same functions in whole range of applications, without the need to resort to direct register access when the standard Arduino digital input and output functions are too slow for given task. Given the fact that Arduino is now widely used both in computer education and scientific experiments, the library can contribute to easier and more reliable realization of these activities. In addition, we believe that the method for abstracting access to general purpose pins of a microcontroller described in this article represents solution which leads to good performance in any library with this purpose and can therefore be interesting to anyone who needs to implement such a library, for example when creating his or her own .

The work was performed with financial support of research project NPU I No. MSMT-7778/2014 by the Ministry of Education of the Czech Republic and by the European Regional Development Fund under the Project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089.

References:

[1] Arduino, *Open-source electronics prototyping platform*, Available: http://arduino.cc.

[2] P. Jamieson, Arduino for Teaching Embedded Systems. Are Computer Scientists and Engineering Educators Missing the Boat?, Available: www.users.muohio.edu/jamiespa/html_papers/fecs_11.pdf, 2015.

[3] P. Bender, K. Kussmann, Arduino based projects in the computer science capstone course, *Journal of Computing Sciences in Colleges*, Vol. 27, No. 5, 2012, pp. 152-157.

[4] J. Oxer, H. Blemings, *Practical Arduino: Cool Projects for Open Source Hardware*, New York, Springer-Verlag, 2009.

[5] M. Banzi, *Getting Started with Arduino*, O'Reilly Media Inc., Sebastopol, 2009.

[6] E. T. da Costa, M. F. Mora, P. A.Willis, C. L. do Lago, H. Jiao, and C. D. Garcia, Getting started with openhardware: Development and control of microfluidic devices. *Electrophorensis* 35 (2014), pp. 2370–2377.

[7] A. D'Ausillio, Arduino: A low-cost multipurpose lab equipment. *Behavior Research Methods*, 2011, 44, 305-313.

[8] J. A. Kornuta, M. E. Nipper, J. B. Dixon, Low-cost microcontroller platform for studying lymphatic biomechanics in vitro, *Journal of Biomechanics*, 2013, 46, pp. 183-186.

[9] B. M. Hoffer, Satisfying STEM Education Using the Arduino Microprocessor in C Programming, 2012, *Electronics Theses and Disertations*, Paper 1472.

[10] S. Jacques, Z. Ren, S. Bissey, A. Schellmanns, N. Batut, T. Jacques, E. Pluvinet, An innovative Solar Production Simulator to better teach the foundations of photovoltaic energy to students, *WSEAS TRANSACTIONS on ADVANCES in ENGINEERING EDUCATION*, Vol. 11, 2014, pp. 11-20.

[11] Why Arduino is not the right educational tool, Available: http://www.hackvandedam.nl/blog/?p=762.

[12] Arduino, *Arduino Issues list*, Available: https://code.google.com/p/arduino/issues/detail?id=140.

[13] Arduino, *Arduino forum*, Available: http://arduino.cc/forum/index.php/topic,46896.0.html.

[14] Wiring, *Home page*, Available: http://wiring.org.co.