EduShields PortMaster
(hardware version 1.1.1)

Introduction

The EduShields PortMaster is a mini-shield for the Arduino that was designed as a tool for building beginning programming skills and for mastering bit-level operations, essential for performing sub-register input/output (I/O) operations. The shield is primarily comprised of ten LEDs and two push-buttons for interacting with the Arduino.

This board is intended to be used primarily for learning purposes – as a tool designed to enable users to develop significant introductory software skills without any hardware complexity getting in the way. It does not aspire to be a WunderTool that will be used for anything else than for practicing (important!) programming concepts – basic input, output, bit-manipulation (expressions of Boolean algebra), programming logic, timing/concurrency/asynchronicity and state machines. In spite of its simplicity and budget-friendliness, it is an interactive and moderately engaging learning tool for the Arduino.

The second design goal of this board is to be an introduction to soldering kit for those that wish to learn this basic skill. The kit tries to be representative of the most common assembly/soldering tasks one needs to acquire to assemble their own projects, especially for the Arduino. It is designed entirely for through-hole parts, although it may also be built substantially with surface-mount technology (SMT, 0805-size specifically).

The design of the PortMaster emphasizes the relationship between bits and bytes, the shared use of bi-directional general-purpose I/O (GPIO) ports and their associated special function registers (SFRs). These basic concepts are common to all microcontrollers. Together with appropriate practice exercises (many of which are provided), one should be able master reading and writing ports and registers directly, above and beyond solely relying on the Arduino library's (simple, albeit low-performing) `digitalRead()` and `digitalWrite()` functions. By learning both methods of access[1] (as well as through AVR assembly language, if one is so motivated), users should gain an appreciation for PORT-style's performance benefits as well as ease of programming where units of data are not comprised of single bits.

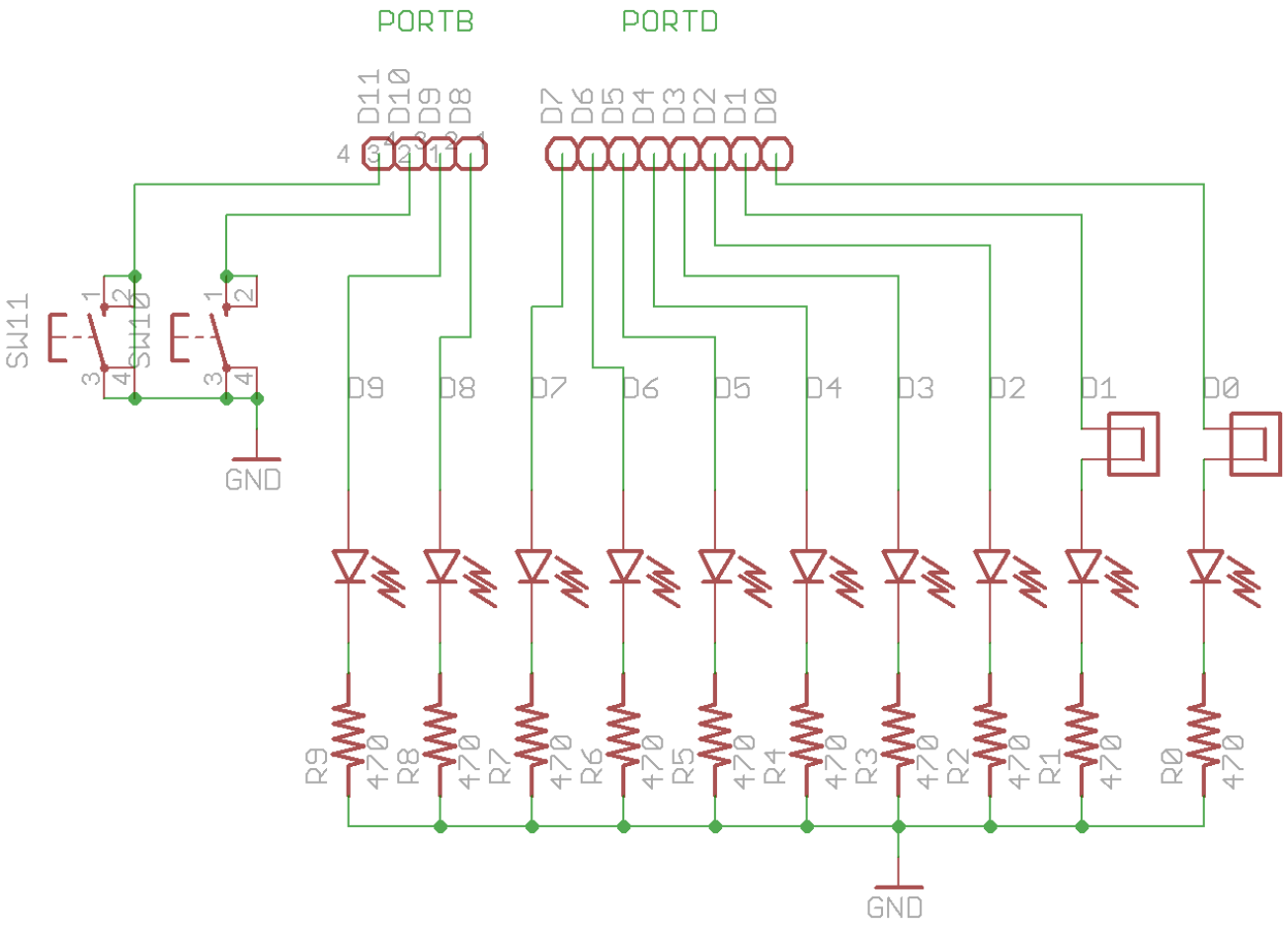
Board design details

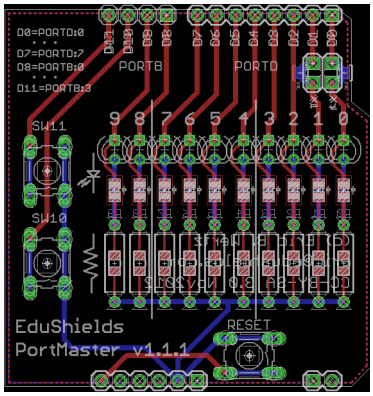
Immediately below are images of the schematic (logical electrical) design as well as the physical PCB (printed circuit board) layout. Documentation on the components and the recommended procedures for assembling a board from kit parts are described in a different document, should you choose or need to know.

Without going into too much detail here, suffice it to say that the LEDs are being driven at about 20% of their maximum continuous current, and the sum of ten of these LED's current comes to about a third

1 A few years ago I began calling these two styles of GPIO access “Arduino-style” and “PORT-style”, for “using the digitalRead/Write() functions in the Arduino library” and “direct reading and writing of the AVR PORTx registers”, respectively.

of what may be funneled directly through the microcontroller out to the LEDs.





The ten LEDs on the board are spread across both PORTB and PORTD (specifically, Arduino Digital Pins 0-9). The rightmost eight LEDs completely fill PORTD with the remaining leftmost two LEDs residing on the low two bits of PORTB. One pair of push-buttons are provided for input interaction.

The buttons SW10[2] and SW11 are wired to the high half of the low nibble of PORTB (Arduino Digital Pins 10 and 11). The remaining active element on the board is the RESET button in the lower, right corner. This button performs the same as the reset button on the standard Arduino itself, but which is almost always inaccessible when any type of shield is installed[3].

By design, this configuration of buttons and LEDs results in PORTB being shared between inputs and outputs – a common situation in many project designs that brings with it programming issues that need to be appreciated. Through directed exercises you will come to recognize, accommodate, and hopefully master conflicts within I/O ports shared between multiple components in your projects.

It should be explicitly noted that the on-board switches do not have pullup resistors installed, therefore these inputs will float if the MCU's internal pull-up resistors are not enabled. This is intentional as almost all MCUs contain programmable pullup resistors, and they are generally put to use in most designs, or at least when push-buttons/switches are used[4]. If you have not yet learned about "pull-ups", you will in the course of working with the PortMaster.

Arduino program uploading considerations

Another very important (but somewhat obscure) point is in regard to the pair of jumpers in the upper-right of the board labeled "tx" and "rx" (transmit and receive). Because of the way that the Arduino is designed, Arduino Digital pins 0 and 1 are used for serial (USB) communication with the PC for uploading sketches and exchanging data with the Serial Console. Because of this, one must either avoid connecting project circuits to them, or at least be aware of the consequences of doing so. Connecting circuitry to them one can interfere with their communications functions, making it impossible to change the code in the Arduino.

To mitigate these issues, an un-populated pair of jumpers for connecting and disconnecting the LEDs connected to these pins, D0 and D1, when re-programming is required. However, in practice we have found that using current-limiting resistors for the LEDs of at least 470ohms provides enough impedance in the output circuit (the LEDs, in this case) that it may remain connected even during programming. The footprints for these jumpers are still in the board layout even though, when using at least 470ohm resistors, they do not need to be installed. Rather, two solder bridge pads are located on the back of the board that can be soldered together leaving the output circuit always-connected. Not having to lift and replace the TX and RX jumpers is a huge convenience in using the board, so it is expected that the solder bridge pads are installed when the board is built. Should board builders choose to use lesser-valued current-limiting resistors than 470ohms, the jumper headers must be installed, and their connections opened when re-programming. If the PortMaster is used on other Arduino variants that exhibit different impedance characteristics that affect programming, either appropriate current-

2 The designator "SWn" stands for "switch *n*", which is the commonly used schematic designator for a "button". Sometimes you may see "PBn" for "push-button *n*", but "PBn" has the unfortunate dual meaning of "PORTB bit *n"* in Atmel literature – which is terribly confusing in this instance. Additionally, it's often challenging choosing between using the Atmel naming scheme of PORTn/PBn or the Arduino's D0-D13,A0-A5, and I've chosen to try to stick with the latter convention throughout.

3 Many other makers of Arduino-compatibles fix this shortcoming by placing their reset button on the board edge where they are accessible even when shields are installed.

4 One exception is when the project is an especially low-power design and higher-valued external resistors (~100Kohm) are used in place of the place of the more standard pullup resistors' values (10K-20Kohm).

limiting resistors must be employed, or the shunt blocks installed for manual manipulation. On a related note, on standard 5V Arduinos, the presence of lesser impedances in circuits connected to D1 seems to present a lesser problem than for similar impedances on D0.

Accompanying Programming Examples

PortMaster users have available to them numerous examples and exercises to help them master the material that this board is designed to teach. Within each example is an introductory comment block that details the key lesson(s) of the sketch. Most examples include questions about its contents and/or supplementary exercises that extend its functionality. In some cases a given example may be the start of a much more complex program built-up through perhaps a dozen directed exercises.

These examples are packaged as a directory hierarchy intended to be installed read-only into the user's own `My Documents\Arduino\libraries` directory[5]. This hierarchy is partitioned into functional areas from elementary programming through more advanced categories – starting with C-language and C pre-processor features, and eventually[6] to things like state machine exercises.

By unpacking the examples archive into `MyDocuments\Arduino\libraries`, you can easily navigate to the example set using `File->Examples` in the Arduino IDE. Should you choose to modify these programs, you will be prompted to provide an alternate location to save them to. This location should typically be your `My Documents\Arduino` directory, the default working directory for programs that you create with the Arduino IDE.

5 After you have downloaded, installed and run the Arduino IDE software, it will create the *Arduino* directory within your Windows' *MyDocuments* folder. Within the *Arduino* sub-folder, you will need to create the *libraries* folder manually. MacOS users probably have to perform a similar sequence of actions.

6 These more advanced examples have been created yet, but gradually are being added to the examples set.