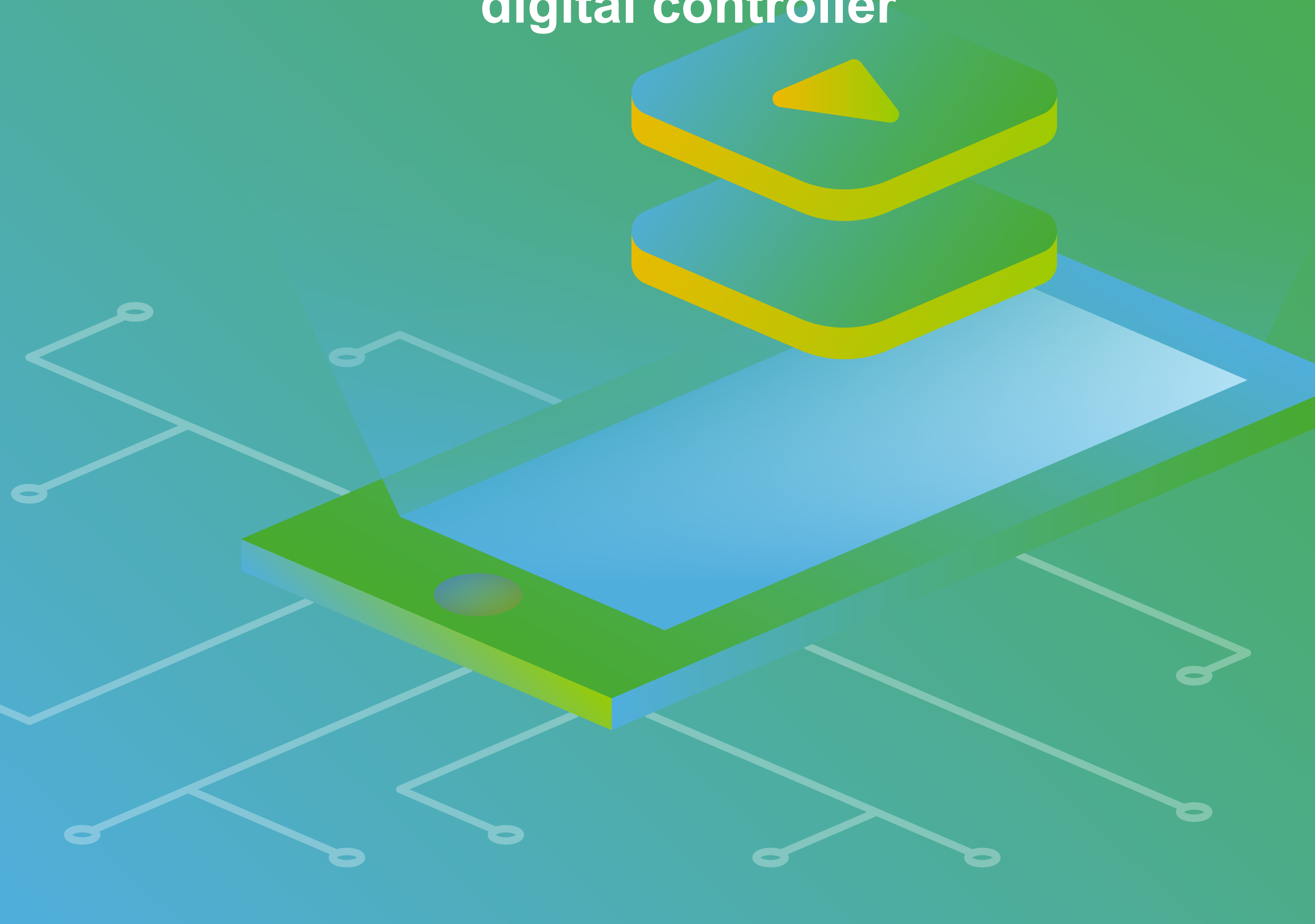
POLYTECH
Peter the Great
St.Petersburg Polytechnic
University

Course

« Introduction to Biomedical Engineering»

Dr. Kirill Aristovich

Section 3: Microcontrollers/Arduino
Lecture 3.1.1: Basics of coding for digital controller



Basics of coding for digital controller

Remember how digital electronics is much more flexible than analog? It is even more useful for so-called embedded systems, the systems which operate real-time and are used for 24/7 process, robot, or medical device control. Broadly speaking the boundary between embedded system and computer is thin nowadays. But we can divide by the following: the embedded system denotes reliable operations, task-specific, and unlikely to be changed or tempered by the user. Non-embedded: high-level, general-purpose, frequently updated or user-controlled.

There are two common classes of embedded controllers: Microcontrollers, and Field-programmable gate arrays or FPGAs. Microcontrollers are Universal, Versatile, support High-level, and generally can be used for anything provided correct embedded software, usually called firmware. FPGAs are pieces of hardware, which can change physical states by a programmer. They are Quick, Reliable, and Parallel. We will be considering only microcontrollers here since what we need is flexibility, and FPGAs require some specific knowledge and skills for programming.

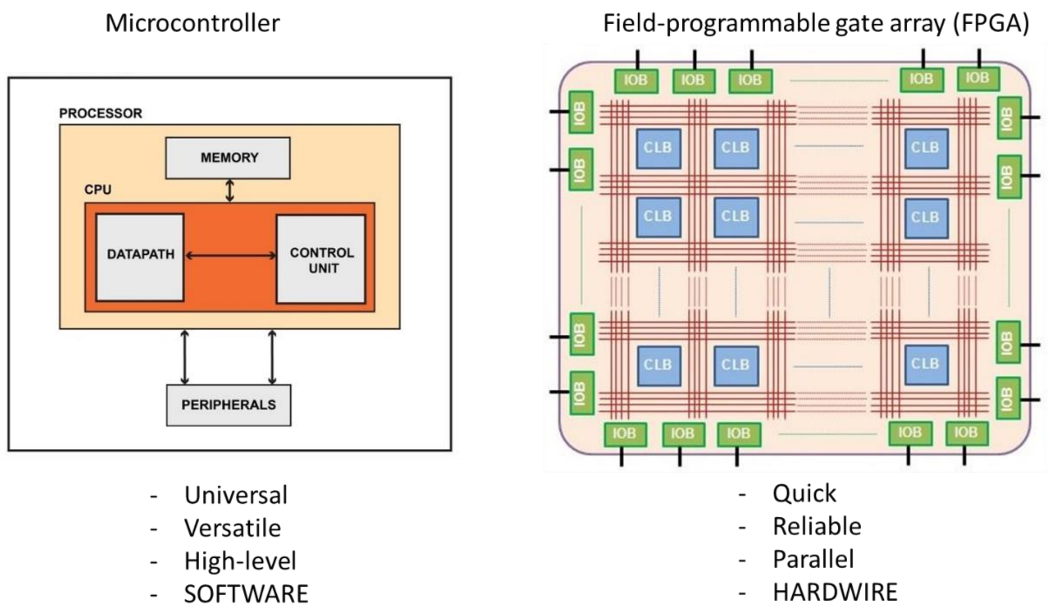


Figure 1 - Embedded controllers

Traditionally, the way microcontrollers are programmed includes three main components, the set-up, main loop which runs infinitely, and contains set of instructions and statements executed consecutively, and interruptions. The latter are executed once, upon specific hardware instructions. C language has been and still is the main language for programming those. It has official standard, but recently more and mare languages, some of them are much more high-level, are evolved and being used in some industries as standard as well.

Modern day microcontrollers operate with much more user-friendly environments, offering high-level access for all of its functions, so you do not have to know the specific chip layout in order to use the specific electronic feature. The languages also, in most of the cases, already have a build-in libraries with simple functions offering more flexible and friendly way for coding.

We will concentrate on Arduino here, as an entry level hacker-friendly controller with lots of features and broad consumer-friendly and socially-oriented support. You can start using it almost immediately, and all you need is to connect it to a computer via USB cable, and start programming.

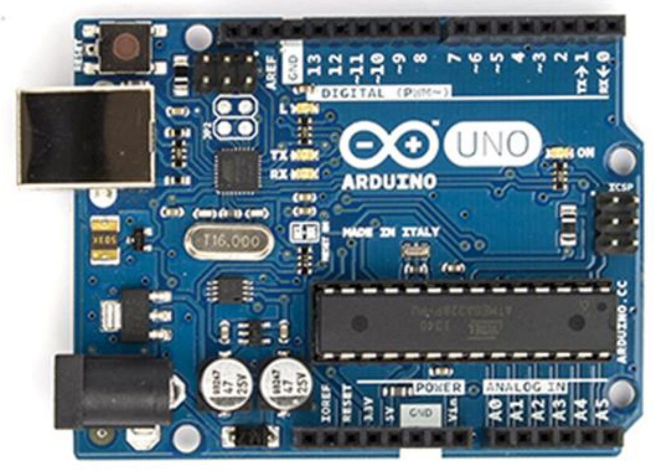
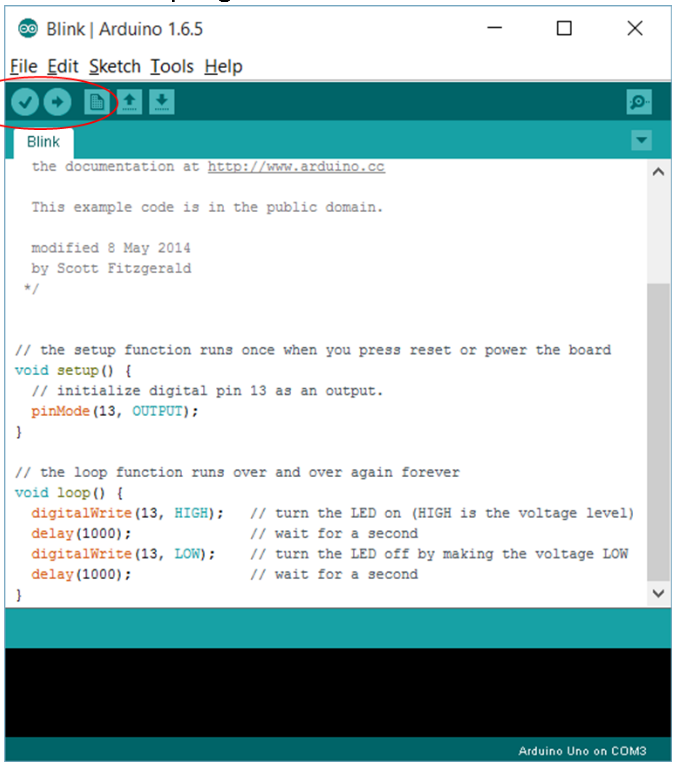


Figure 2 – Arduino base shield

Arduino programming environment (also called Integrated Development Environment or IDE) is free and easy to use. Once you loaded an example code, or written your own, you need to select the type of board you have, which port it is connected to, compile (check-button), download (arrow button), and observe the operation. That is very easy, comparing to some early-days microcontrollers I have dealt with, which required 3 days to set-up stuff just to be able to program it.

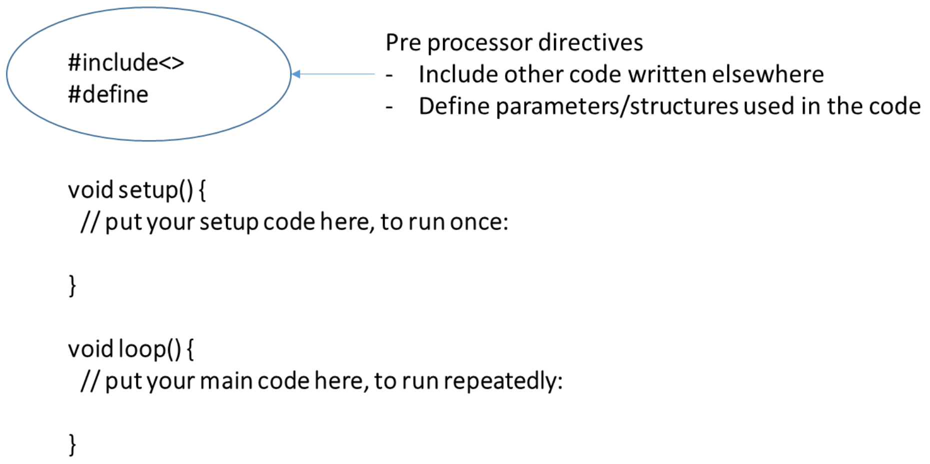


```
  the documentation at http://www.arduino.cc

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
 */


// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

Figure 3 - Arduino IDE overview

The best way to learn it is to download the code, run it to see what happens, and then read what the code is doing. This way you can copy and paste existing code components to use and modify for your own purposes.

Arduino program is called the sketch, it is C++ based language, and generally follows traditional structure: there is pre-processor section where you include libraries or other code and define some hard variables. 'Setup' section where you list stuff you want your controller to do at the start. And 'loop', the main section which runs forever repeatedly. Arduino IDE also helps you with color-coding, highlighting built-in functions, statements, and variables.

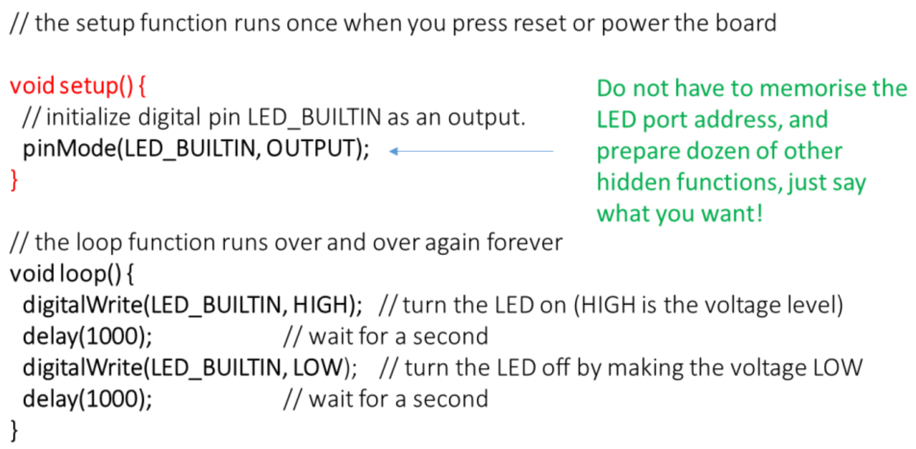


```
#include<>
#define

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Pre processor directives
- Include other code written elsewhere
- Define parameters/structures used in the code

Figure 4 - Arduino program structure

Let's follow my own advice, download the first example code 'blinking LED' and see what it does. After you turn the Arduino on, it goes to setup section, where there is a command which enables the LED_BUILDIN digital pin for output. Each digital pin of Arduino can be set for input (send voltage to the pin) or output (read voltage from the pin). The beauty is that you do not have to memorize the address of the pin or anything, Arduino knows what it needs to do. Each instruction have to be separated by a semicolon. That is a common source of errors and important to follow contrary to some other languages.



```
// the setup function runs once when you press reset or power the board

void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

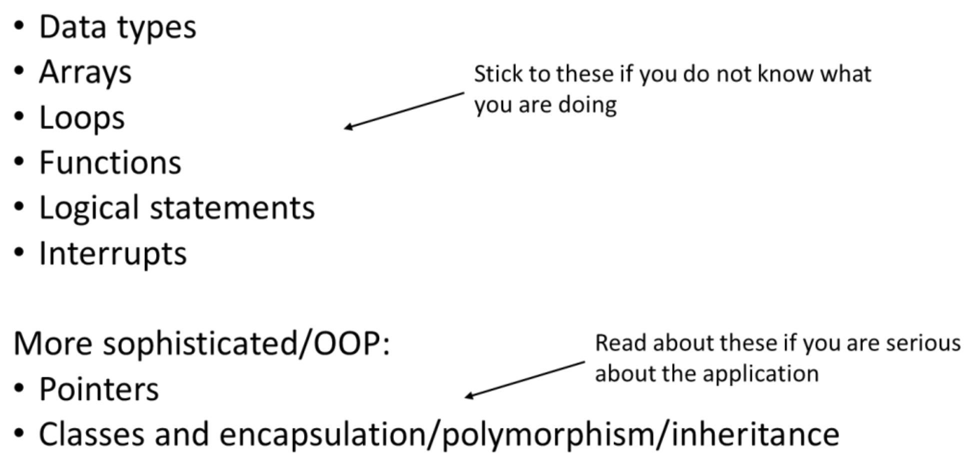
Do not have to memorise the LED port address, and prepare dozen of other hidden functions, just say what you want!

Figure 5 - blinking LED code

There is no more instructions in the setup so controller moves on to the loop, where we have digitalWrite instruction, which sends 1, or HIGH to the pin. This sets the pin to 5 Volts and that in turn makes LED glow.

Then there is a delay for 1000 ms, or 1 sec, after which we get digitalWrite with LOW, which sets the pin to 0 V, hence switching the LED off. Then another 1 sec wait, and the loop goes to the beginning, repeating the same sequence over and over again. The final result is blinking once a second LED.

There are several necessary and useful programming concepts to familiarize yourself with, before you can start coding confidently: The minimum list is on your screen, plus some additional stuff to learn if you would like to become a professional programmer. We will briefly touch each of them.



- Data types
- Arrays
- Loops
- Functions
- Logical statements
- Interrupts

Stick to these if you do not know what you are doing

More sophisticated/OOP:

- Pointers
- Classes and encapsulation/polymorphism/inheritance

Read about these if you are serious about the application

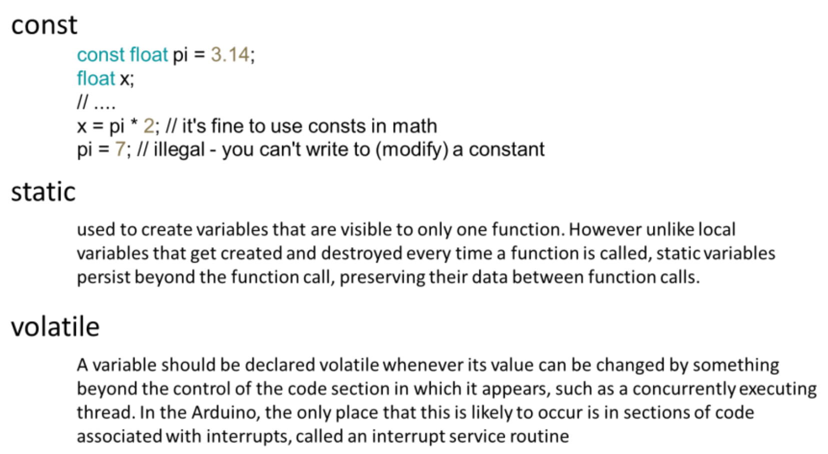
Figure 6 - Useful (necessary) programming concepts

The variables in Arduino have to be declared, meaning the controller needs to know which type the particular variable is. This is done to save limited memory. It can also be used for constrains, but sometimes causes hard to trace bugs and falling planes. So be careful with those and read carefully about each type when using it.

The main once are: integer (whole numbers), float (decimal with floating point), and characters. You can also use Byte and Boolean, and some additional types at your disposal.

The place where the variable is declared defines so-called variable scope. If you declare it outside of the main loop, it will be GLOBAL, and can be seen from any function and loop. The local variables, ones that are defined inside loops and functions, can only be seen from inside of those loops and functions. It gives you the tool to save memory, as local variables are cleared when function does not run, and ability to use the same name for different variables.

The scope qualifiers are used to create some specific features: ‘const’ declares a variable that cannot change. ‘Static’ lets you keep the variable inside the memory even when outside of the scope. You might use this to save time for new memory allocation each time the loop or function runs. ‘Volatile’ type declares the variable which value can be changed by something outside of the main code. We will talk about them later when learning interruptions.



const

```
const float pi = 3.14;
float x;
// ....
x = pi * 2; // it's fine to use consts in math
pi = 7; // illegal - you can't write to (modify) a constant
```

static

used to create variables that are visible to only one function. However unlike local variables that get created and destroyed every time a function is called, static variables persist beyond the function call, preserving their data between function calls.

volatile

A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine

Figure 7 - Variable scope qualifiers

When one variable is not enough, there are arrays, which store multiple values of the same type. There are multiple ways for declaring them. You can specify the size of the array, or leave it open. The access and change individual values happens by specifying the position within the array.