POLYTECH
Peter the Great
St.Petersburg Polytechnic
University

Course

« Introduction to Biomedical Engineering»

Dr. Kirill Aristovich

Section 3: Microcontrollers/Arduino
Lecture 3.1.2: Basics of coding for digital controller

Basics of coding for digital controller
PART 2

If you need to do a repetitive task it's a good idea to use loops. The most common is 'FOR' loop. This example loop will start with I equals zero, and execute the command of writing I to analog pin 256 times for each i. Note the scope of variable I, which will only exist within the loop and be destroyed upon the execution.

There is also while loop, which executes the statements within curly brackets while the condition in round brackets is true. And finally, do-while loop, which comes handy if you would like to check the statement AFTER each iteration rather than before.

```
for (int i = 0; i <= 255; i++) {
   // do something repetative 256 times
   analogWrite(PWMpin, i);
}
```

```
int var = 0;
while (var < 256) {
   // do something repetitive 256 times
   var++;
}
```

```
int x = 0;
do {
   delay(50); // wait for sensors to stabilize
   x = readSensors();
   // check the sensors
} while (x < 100);
```

Figure 1 - Loops

If you need to use the same action several times under different conditions, you may consider functions. A function can take input variables, and can return output variables. The common practice is to declare the function after the main loop, or in a different file. In this example we define the function with two input variables, a and b, which are integers. The function is declared to be integer as well. Inside the function we compute a + b. the statement RETURN specifies the stuff that the function should output. In this case we output the variable 'result'. Make sure that the stuff you return matches the type of the function that you have declared.

Now, in the main loop we can use this function and compute a sum of two specific values. There are lots of pre-built functions in Arduino, and the help is great. Plus, there is a very large community where you can download functions that other people have created. You can get plenty of help on the website https://www.arduino.cc/reference/en/

Digital I/O
digitalRead()
digitalWrite()
pinMode()

Analog I/O
analogRead()
analogReference()
analogWrite()

Zero, Due & MKR Family
analogReadResolution()
analogWriteResolution()

Advanced I/O
noTone()
pulseIn()
pulseInLong()
shiftIn()
shiftOut()
tone()

Time
delay()
delayMicroseconds()
micros()
millis()

Math
abs()
constrain()
map()
max()
min()
pow()
sq()
sqrt()

Trigonometry
cos()
sin()
tan()

Characters
isAlpha()
isAlphaNumeric()
isAscii()
isControl()
isDigit()
isGraph()
isHexadecimalDigit()
isLowerCase()
isPrintable()
isPunct()
isSpace()
isUpperCase()
isWhitespace()

Random Numbers
random()
randomSeed()

Bits and Bytes
bit()
bitClear()
bitRead()
bitSet()
bitWrite()
highByte()
lowByte()

External Interrupts
attachInterrupt()
detachInterrupt()

Interrupts
interrupts()
noInterrupts()

Communication
Serial
Stream

USB
Keyboard
Mouse

Figure 2 - Functions

The operation of automation is impossible without logical statements, which you can use to create conditional statements. General syntax of if-else is on your screen, and you can use curly brackets to isolate the statements.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120) { digitalWrite(LEDpin, HIGH); }

if (x > 120) {
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}
```

```
if (condition1) {
  // do Thing A
}
else if (condition2) {
  // do Thing B
}
else {
  // do Thing C
}
```

Figure 3 - Logical statements

There are other bits worth knowing and reading about, some of which are specific to arduino, and you can read about them on their website.

Sketch
loop()
setup()

Control Structure
break
continue
do...while
else
for
goto
if
return
switch...case
while

Further Syntax
#define (define)
#include (include)
/* */ (block comment)
// (single line comment)
; (semicolon)
{} (curly braces)

Arithmetic Operators
% (remainder)
* (multiplication)
+ (addition)
- (subtraction)
/ (division)
= (assignment operator)

Comparison Operators
!= (not equal to)
< (less than)
<= (less than or equal to)
== (equal to)
> (greater than)
>= (greater than or equal to)

Boolean Operators
! (logical not)
&& (logical and)
|| (logical or)

Pointer Access Operators
& (reference operator)
* (dereference operator)

Bitwise Operators
& (bitwise and)
<< (bitshift left)
>> (bitshift right)
^ (bitwise xor)
| (bitwise or)
~ (bitwise not)

Compound Operators
%= (compound remainder)
&= (compound bitwise and)
*= (compound multiplication)
++ (increment)
+= (compound addition)
-- (decrement)
-= (compound subtraction)
/= (compound division)
^= (compound bitwise xor)
|= (compound bitwise or)

Figure 4 - Other bits worth knowing

For now, we are going to learn about very micro-controller specific tool, which is called interruption, and revise the cases where you might want to use those.

Normal flow of the program is to follow statements one by one in the main loop over and over again. Interruption is the built-in functionality to temporally stop that, and execute something completely different. This is handy for making things happen reliably in microcontroller programs, and can help solve timing problems. For example, think about catching the pulses from a rotary encoder, so that program never misses a pulse. Or trying to read a sound sensor that is trying to catch a click. Or Infrared slot sensor (photo-interrupter) trying to catch a coin drop.

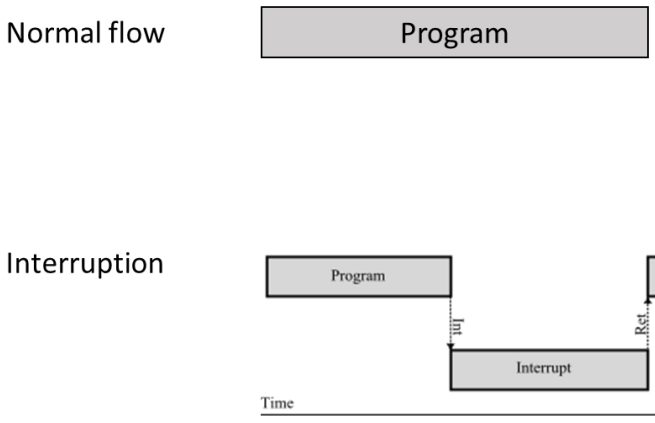

Figure 5 - Program flow and interrupts

In order to use interrupts, you need to declare them beforehand with the specific command attachinterrupt, which takes 3 parameters. The first specifies the pin which

needs to be monitored. The second needs to contain the name of the function that executes in the event of interruption. It is called Interrupt Service Routine, and you declare it like a normal function. And the third is the mode, this indicates what kind of action on the pin triggers the interruption.

Provided example switches the state of LED every time the voltage on the input pin changes the voltage. Take a bit of time to read through and familiarise yourself with it.

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode )
```

```
const int ledPin = 13;
const int interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() {
  state = !state;
}
```

Figure 6 - Using interrupts

ISRs are special kinds of functions that have some unique limitations most other functions do not have. An ISR cannot have any parameters, and they shouldn't return anything. Typically global variables are used to pass data between an ISR and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as volatile.

Right, if something is still not clear, the following example is really handy to understand what is going on. Our task is to design the pelican crossing controller. What we have is the Traffic light, zebra crossing, and button that pedestrian can press. It should work the following way: Always green to cars unless button is pressed. If button is pressed, turn red to cars, green to pedestrian. Wait 1 minute, then return to Always green to cars.

```
// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
    delay(60000); // 1 minute = 60 seconds
  } else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

Figure 7 - Example: pelican crossing algorithm (no interrupts)

We will simplify this for 1 built-in LED (‘on’ indicates safe to cross and stop for cars, ‘off’ indicates green for cars and red to pedestrians) and 1 built-in button.

Here is the first attempt for doing this. We will start the loop by checking the button state, then check if it is HIGH (button is pressed), we will turn the LED on and wait for 60 seconds. If the button is LOW, then we will send ‘off’ to LED. According to this code, we will keep it off, until the button press is detected in the beginning of the loop, where we turn the LED on and stop for a while. Can you spot the reliability and inflexibility issue? The only time the button is checked is short period in the beginning of the loop. Imagine now that we have much more code which takes a while to get through! What happens if button is pressed during the time LED is on? Or what do we do if we would like to let cars go for a while after pedestrian crossed to avoid traffic jam? Spend some time analysing this code until you get the issues.

Now, let’s design an interruption-based code to see how useful it is. Make a not that the buttonState is now declared volatile, so ISR can actually access it and change it. We have a very simple ISR now called GoPedestrian which changes this variable to ‘pressed’. And this happens reliably EVERY TIME the button is pressed irrespective of what is happening in the main loop.

In the loop now we are checking this status and immediately change it to 'un-pressed' before switching LED on. This way even if the button is pressed during 'on' we do not miss this and can repeat the cycle again for the next pedestrian. We can also easily add delay functionality if we want to. I would like you to download this and play around, making sure you understand everything that is going on, and add extra functionality to make it closer to the real pelican crossing.

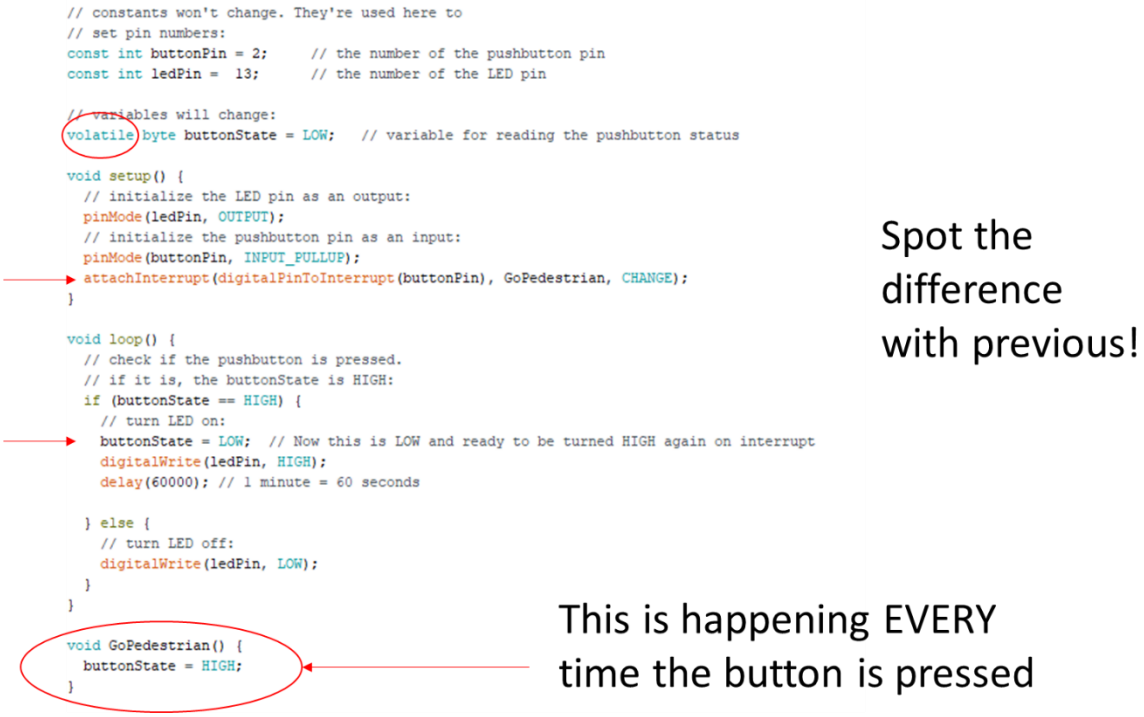

Figure 8 - Example: pelican crossing algorithm (interrupts)

Some graphic material used in the course was taken from publicly available online resources that do not contain references to the authors and any restrictions on material reproduction.

This course was developed with the support of
the "Open Polytech" educational project

Online courses from the top instructors of SPbPU