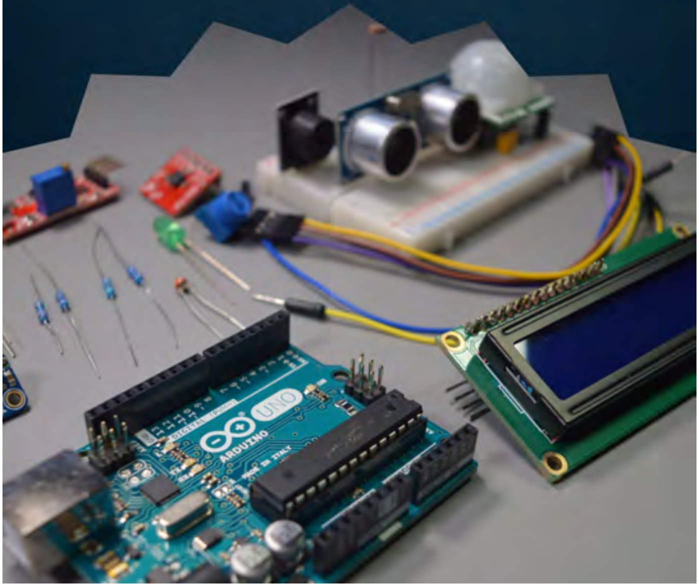
Tech Explorations

www.techexplorations.com

ARDUINO PROGRAMMING

GET THE MOST OUT OF YOUR ARDUINO WITH ARTICLES FROM THE TECH EXPLORATIONS BLOG



Peter Dalmaris, PhD

Arduino Programming Tips and Tricks

Get the most out of your Arduino with articles from the Tech Explorations Blog

Welcome to this special collection of articles, meticulously curated from the Tech Explorations blog and guides. As a token of appreciation for joining our email list, we offer these documents for you to download at no cost. Our aim is to provide you with valuable insights and knowledge in a convenient format. You can read these PDFs on your device, or print.

Please note that these PDFs are derived from our blog posts and articles with limited editing. We prioritize updating content and ensuring all links are functional, striving to enhance quality continually. However, the editing level does not match the comprehensive standards applied to our Tech Explorations books and courses.

We regularly update these documents to include the latest content from our website, ensuring you have access to fresh and relevant information.

1. PWM and buffer overflow

Arduino programming guide series

PWM and buffer overflow

What happens if you write a PWM value that is larger than the maximum value that the Arduino's analogWrite() function can accommodate? This is an interesting case of "buffer overflow".

The Arduino Uno is able to produce Pulse Width Modulation signals via pins 3, 5, 6, 9, 10, and 11. With PWM, you can approximate analog output programmatically and do things like fade an LED on and off or control the speed of a motor.

PWM values and register bits

In the Atmega328 (the chip that powers the Arduino Uno), the register that is used by the PWM function has a resolution of 8 bits. This gives you a total of 256 possible "analog" output levels, from 0 to 255.

If you attach an LED to a PWM-capable pin, you can drive it to 256 different brightness levels, from totally off (PWM value "0") to totally on (PWM value "255").

And if you attach a motor, you can drive it to 256 different speed levels.

Why 256? because the register contains 8 bits, and a binary number with 8 bits can be one of 2 ^ 8 = 256.

You can set a PWM value by using the analogWrite(pin, value) instruction.

So, **analogWrite(3, 125)** would set pin 3 to value 125.

How to overflow the PWM register

Now, here is where it gets interesting.

What happens if we set analogWrite to a value bigger than 255? Say, 256?

Let's think about this for a minute.

If the PWM value is 255, the binary version is 11111111 (total is 8 bits) is stored in the PWM register (feel free to use this calculator for such binary to decimal conversions). A connected LED would light up in maximum brightness.

Let's add 1 to the register, and make the PWM value 256.

The binary version of 256 is 0000000100000000 (total is 16 bits) since now we need two bytes to represent this value.

But, the Arduino (in reality, its Atmega328P chip) can only fit the first byte in its PWM register, the one in green.

The effect of PWM register overflow

The second byte will overflow and "disappear" (the red part).

So, what is actually stored in the PWM register is 00000000. This is decimal "0", which means that your LED is turned off.

In other words, **analogWrite(3, 0)** and **analogWrite(3, 256)** would have the exact same effect on an LED or a motor.

Add another "1" to the register, and the PWM value now is 257.

The binary version of 257 is 0000000100000001**.** The byte in green is stored in the PWM register, and the rest (in red) disappears. In the register now the decimal value "1" is stored.

The lesson to take home is that although you can set the PWM value in analogWrite to any decimal you like, only the first byte of this number will fit in the PWM register.

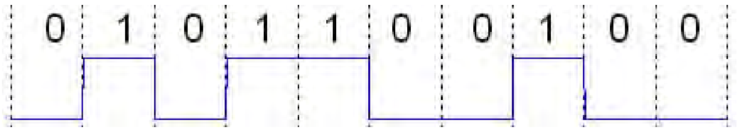
The rest will overflow and disappear.

2. What is the baud rate?

Arduino programming guide series

What is the “baud” rate?

The “baud” rate is a unit used to describe the speed of serial communications between two electronic devices. The exact meaning of this unit is often clout in confusion. What exactly is “baud”?



The baud rate (the symbol is “Bd”) is unit we use to describe the “speed” of communication between the two electronic devices. Your computer and the Arduino talk via the USB/RS-232 or similar (serial) interface. There are a few parameters that control this communication, but all of them are standardized (so you don’t have to worry about them) except for the speed.

If you don’t set the two devices to the same speed, then one device will be sending data to the other faster or slower than expected. When two serial devices are not set on the same communications speed, data interchange will not be reliable.

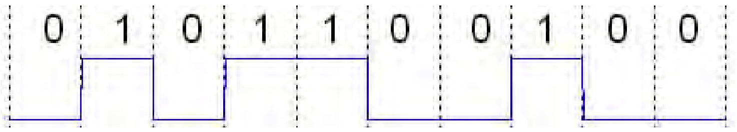
What is a "baud"?

A speed of 9600 bauds means that data will flow between the devices at a rate of 9600 **signal changes** per second.

A signal change is an event such as a change in one of the signal's electrical characteristics, such as the voltage, phase, or frequency.

A common cause of confusion here is that often people incorrectly perceive that 1 signal change is the same as 1 bit. Therefore, people expect that 9600 bauds are the same as 9600 bits per second, which it isn't.

At the electrical level, a computer uses voltages to transmit data (a computer can also use multiple voltage levels plus the signal phase to encode multiple bits in a single signal change, but let's keep things simple here).



By El pak at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=20667098

Data is transmitted by changing the signal parameters rapidly. For example, you can create a protocol where binary "1" is transmitted by changing the voltage from HIGH to LOW, and a "0" by changing the voltage from a LOW to HIGH.

When you have a protocol where 1 signal change encodes 1 bit, then 1 baud = 1 bit per second.

It is possible to have multiple bits encoded in a single signal change, as in Phase-shift keying (PSK). In PSK, each signal change encodes 2 bits. So, 1 baud = 2 bit per second.

In quadrature amplitude modulation (QAM), we have 4 bits for every signal change, so 1 baud = 4 bit per second.

And the list goes on.

COM speed in the Arduino

In practical terms, you must ensure that your devices are communicating at the same baud rate.

For the Arduino, do this:

1. Look in your Arduino sketch for a line that looks like this (in the setup function): **Serial.begin(9600)** — The number may vary.2. Set the speed of your serial terminal on the computer to the same number.

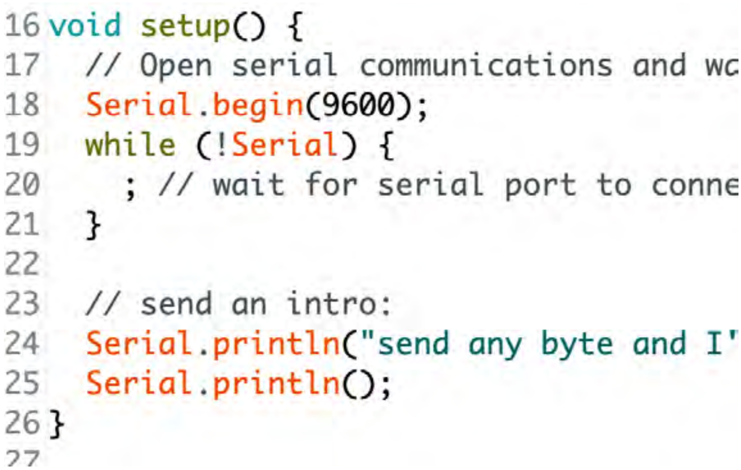
This should do it!

3. Focus on the type parameter in println()

Arduino programming guide series

Focus on the type parameter in “println()”

We use the println() function to print a string to the serial monitor. The same function is overloaded with a second parameter that allows us to designate the type of data we want to display.



```
void setup() {
  // Open serial communications and wc
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to conne
  }

  // send an intro:
  Serial.println("send any byte and I'
  Serial.println();
}
```

“Serial.println()” is one of the most useful functions in the Arduino language. It will print a string or a number to the serial monitor. It is an easy way to get data out of a sketch.

The most common way to use “println()” is like this:

Serial.println("This will print this message to the Serial monitor");

In this example, the single parameter of the function contains a String.

But, did you know that this function is overloaded, and can accept a second parameter?

The second parameter gives you the opportunity to specify the type of the data you specify in the first parameter.

Here's two examples:

- **Serial.println(21, DEC)** will print "21" in the serial monitor.
- **Serial.println(21, BIN)** will print "00010101 " in the serial monitor.

The "println()" function accepts the following types:

- DEC for decimals
- HEX for hexadecimals
- OCT for octals
- BIN for binary numbers

The type parameter is optional, so you can choose to not include one in your println instruction. In that case the output will default to decimal (DEC).

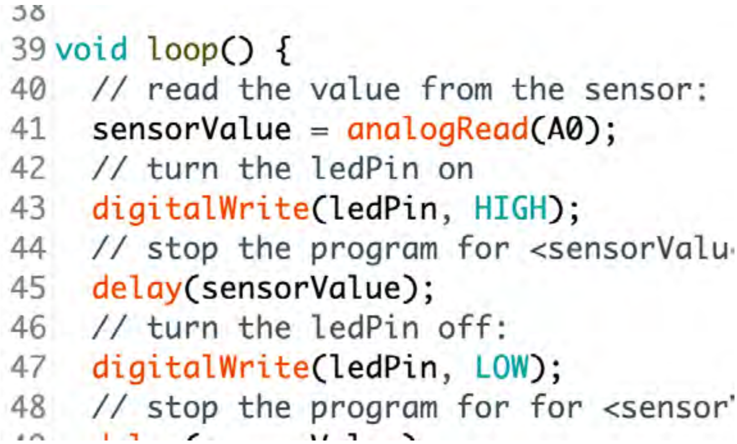
More details about this parameter are available in the

4. &#8221;0&#8243; or &#8220;A0&#8221; when used with analogRead()?

Arduino programming guide series

“0” or “A0” when used with analogRead()?

The short answer is, it does not matter. But there’s a catch.



```
void loop() {
  // read the value from the sensor:
  sensorValue = analogRead(A0);
  // turn the ledPin on
  digitalWrite(ledPin, HIGH);
  // stop the program for <sensorValu
  delay(sensorValue);
  // turn the ledPin off:
  digitalWrite(ledPin, LOW);
  // stop the program for for <sensor'
```

When you use the analogRead() function, “0” and “A0” refer to the same pin, the analog pin 0.

These two invocations of analogRead() are equivalent:

analogRead(0);

analogRead(A0);

Therefore, when you use analogRead() or analogWrite(), it doesn't matter which pin name or number you use.

However, there's a "catch".

It is possible to use an Arduino analog pin as if it was a digital pin.

For example, you can write:

digitalRead(A0);

This will return either HIGH or LOW, instead of a number from 0 to 1023 (assuming you are using an Arduino Uno).

Points to remember:

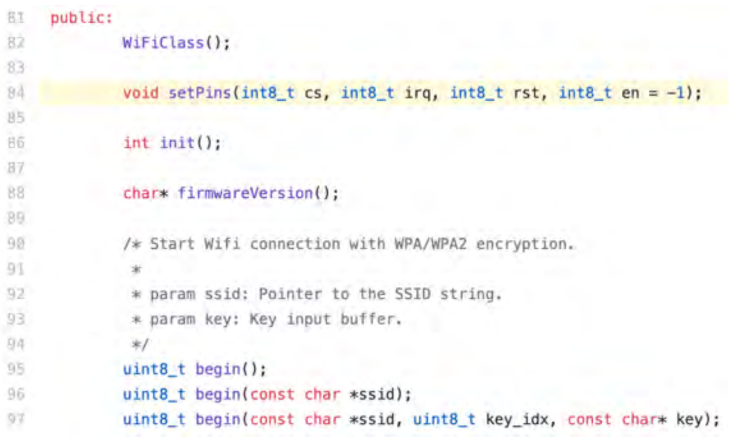
- If you use an analog pin 0 with digitalWrite() or digitalRead(), then you must use the full analog pin designator ("A0", "A1", etc).
- If you use an analog pin with analogRead() or analogWrite(), you can simply use the number of the analog pin ("0", "1", etc.) or the full analog pin name ("A0", "A1", etc.).

5. What is the _tin uint8_t ?

Arduino programming guide series

What is the "_t" in "uint8_t"

The Arduino language contains several easily recognizable variables, like "bool", "byte", "int" and "char". But, below the surface, the Arduino language is really a subset of the C language that works on microcontrollers. With it, you will find many specialized data types designed to ensure compatibility across devices that don't always treat a byte the same way.



```
public:
        WiFiClass();

        void setPins(int8_t cs, int8_t irq, int8_t rst, int8_t en = -1);

        int init();

        char* firmwareVersion();

        /* Start Wifi connection with WPA/WPA2 encryption.
         *
         * param ssid: Pointer to the SSID string.
         * param key: Key input buffer.
         */
        uint8_t begin();
        uint8_t begin(const char *ssid);
        uint8_t begin(const char *ssid, uint8_t key_idx, const char* key);
```

If you have looked at more advanced Arduino code, perhaps looking at Arduino sources on Github, you may have noticed that a lot of variable types end with "_t".

You probably already know that the Arduino "language" is based on the C language. I am not exaggerating when I say that our modern civilization depends on C and its object-oriented cousin, C++. No matter what gadget (computer, tablet, phone) you are reading this on, your electronic device functionality infrastructure is written in C and C++.

Because C and C++ is used on so many different platforms, from microcontrollers to supercomputers, there was a need for types (like integers, floats, etc.) that are compatible across all these platforms.

So, in the C99 standard (the ISO standard for the C language), types that are designed to be cross-platform compatible are marked with a "_t". "t" stands for "type."

This way, the programmers know that the **uint8_t** is a byte with 8 bits no matter which platform the program runs on.

If you strive to write code that can be executed on different computer or microcontroller systems, then it is good practice to use data types with the "_t" extension for this reason.

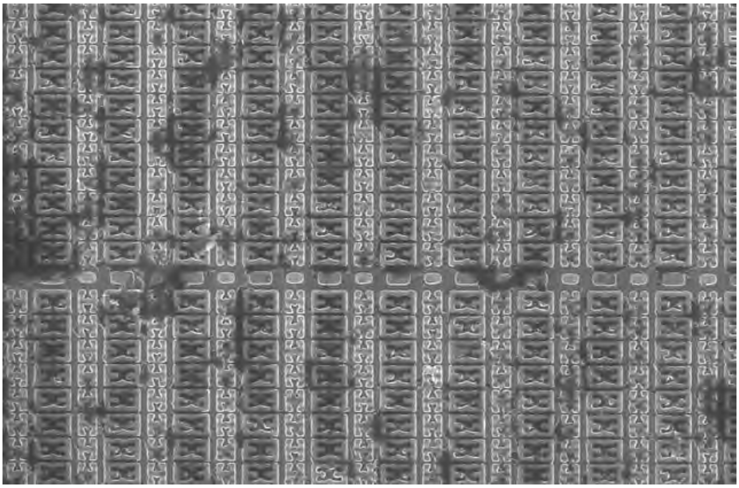
For the Arduino, we tend not to use C99 cross-platform compatible types because most often our code is not meant to run on other systems.

6. How can you use the F() function to save on RAM?

Arduino programming guide series

Save SRAM with the F() macro

In embedded devices, like the Arduino, RAM is limited. However, there are a few simple methods you can use to reduce the memory footprint of your sketches so that they will fit in the available memory space. The F() macro is one such method.



The image above is a photo of static RAM (SRAM) cells of a STM32F103VGT6 microcontroller, taken by a scanning electron microscope. By ZeptoBars – http://zeptobars.ru/en/read/open-microchip-asic-what-inside-II-msp430-pic-z80, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=25044206

The Arduino Uno's ATmega328P only has 2048 bytes of SRAM (static RAM). It also has 32 KBytes of flash memory.

The flash memory is where your sketch is stored when you upload it. Flash memory persists when you turn the Arduino off.

The SRAM is where the various variables in your sketch are stored during execution. Unlike flash memory, the contents of the SRAM do not persist when you turn the Arduino off.

In the Arduino Uno, there is a lot more flash than SRAM space.

Let's look at an example of how SRAM space is used, and how easy it is to misuse it.

Take this simple instruction:

```
Serial.println("Hello world!");
```

It doesn't do anything fancy. It will simply print a short string of text to the serial monitor.

The string of text in the double quotes will occupy 12 bytes in the SRAM. If you have a few additional println() statements like that, your sketch will easily fill the SRAM with static strings of text.

If your program is large with a lot of variables like this, there is a good chance you will run out of RAM, and your program will crash.

Because the string in this example is static, i.e. it does not change, you can easily optimize your sketch by storing the string in the plentiful flash memory instead of the SRAM.

Serial.println(**F("Hello world!")**);

I have marked the F() macro used in the println() instruction in bold.

The F() macro tells the compiler that the contents of the parameter should be stored in the flash memory (program memory or PROGMEM), instead of the SRAM, hence saving you this valuable resource for better use.

The F() macro provides us an easier syntax for marking compatible data type values to be stored in program memory, in place of the PROGMEM keyword modifier. If you wanted to use the raw PROGMEM modifier instead of the F() macro, you would do it like this:

const char **a_string**[] PROGMEM = "Hello world!";

void setup(){ Serial.begin(9600); for (byte k = 0; k < strlen_P(**a_string**); k++) { myChar = pgm_read_byte_near(**a_string** + k); Serial.print(myChar); }}

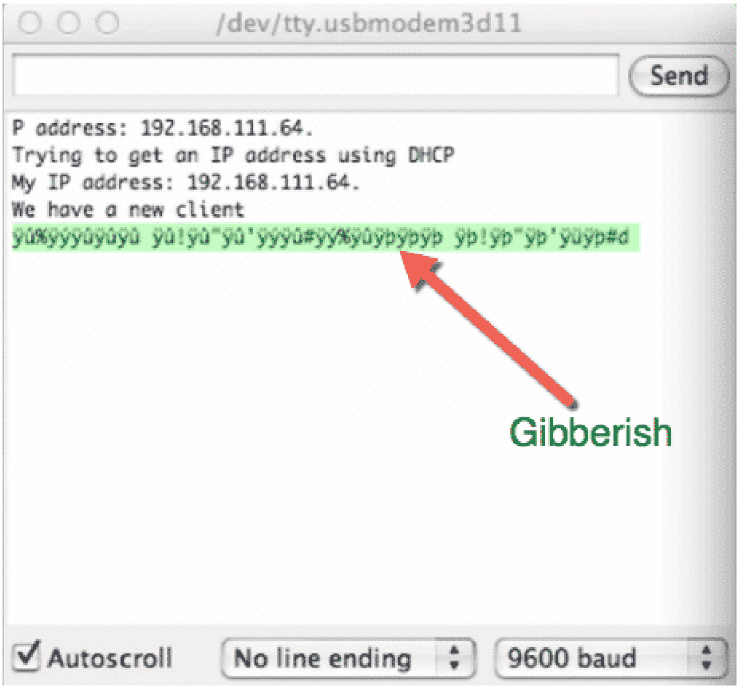
I find the F() macro much easier compared to manually marking a constant string for storage in program memory and then reading it back one byte at a time.

7. What is the gibberish in your Telnet output?

Arduino programming guide series

What is the gibberish in your Telnet output?

When the Serial object in your sketch and the Arduino IDE serial monitor are set to the same speed, you would expect that clear, readable text would appear on your screen. Sometimes it doesn’t. What’s going on? is it a glitch? what can explain the gibberish that appears?

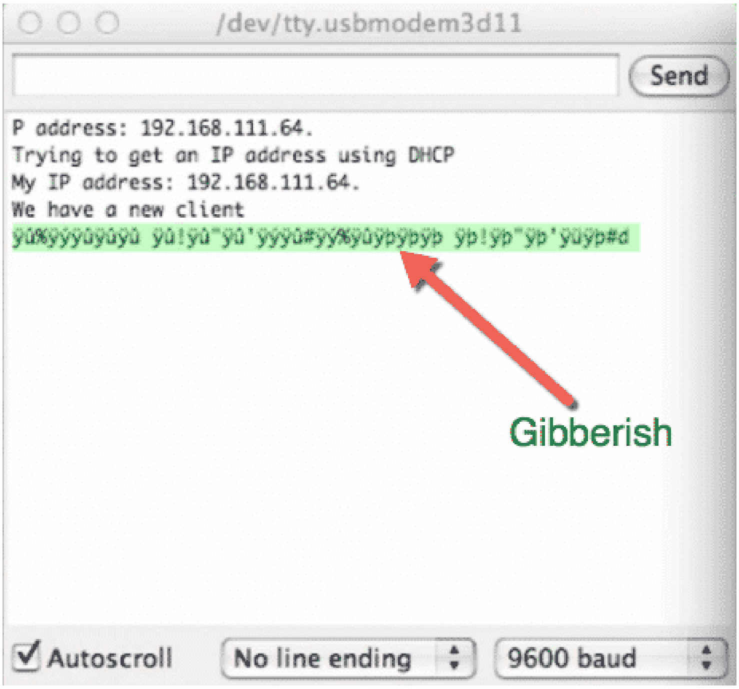


```
P address: 192.168.111.64.
Trying to get an IP address using DHCP
My IP address: 192.168.111.64.
We have a new client
ÿû%ÿýÿûÿûÿû ÿû!ÿû"ÿû'ÿýÿû#ÿý%ÿûÿþÿþÿþ ÿþ!ÿþ"ÿþ'ÿûÿþ#d
```

No line ending
9600 baud

If you an eye for details, you may have noticed that when you start a Telnet connection between your computer and an Arduino equipped with an Ethernet shield, some "gibberish" text appears as soon as the connection is established and before you can start sending text messages back and forth.

It looks like this:



```
P address: 192.168.111.64.
Trying to get an IP address using DHCP
My IP address: 192.168.111.64.
We have a new client
ÿû%ÿÿÿûÿûÿû ÿû!ÿû"ÿû'ÿÿÿû#ÿÿ%ÿûÿþÿþÿþ ÿþ!ÿþ"ÿþ'ÿûÿþ#d
```

What is the origin of this "gibberish"?

Even though Telnet is a simple protocol that runs over TCP (the Internet's Transmission Control Protocol), it still requires that a handshake process takes place before a connection between a client and a server is established.

The handshake process is a quick "conversation" between the client and the server. The two terminals are negotiating the minimum set of parameters that are acceptable by both. Once they agree on this, the communication can start.

These parameters are formally called "Telnet options."

Although these options can be negotiated at any time, they typically are when a session is first established.

The Arduino with it's Ethernet shield, and a basic Telnet server sketch will ignore all of the handshake communication since it is not equipped to deal with them. The client will then drop down to the most basic level of Telnet communication.

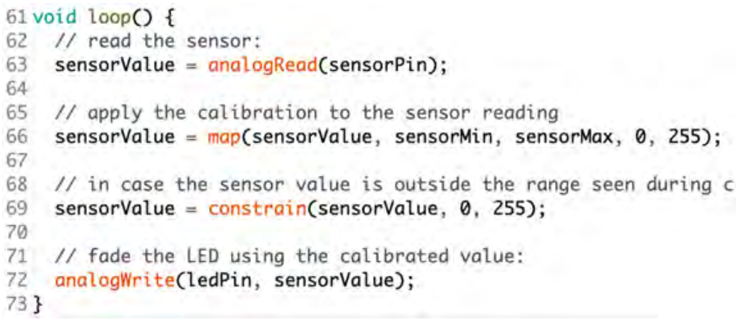
If you are curious, here's some more information about Telnet connections.

8. The Arduino map function

Arduino programming guide series

The Arduino map() function

The map() function makes it easy to convert numbers from one range to another. Here's a simple example of its usage.



```
void loop() {
  // read the sensor:
  sensorValue = analogRead(sensorPin);

  // apply the calibration to the sensor reading
  sensorValue = map(sensorValue, sensorMin, sensorMax, 0, 255);

  // in case the sensor value is outside the range seen during c
  sensorValue = constrain(sensorValue, 0, 255);

  // fade the LED using the calibrated value:
  analogWrite(ledPin, sensorValue);
}
```

The map() function makes it easy to convert a value from one range into a proportional value of another range.

Let's use an example that involves a potentiometer and an electrical motor.

We can sample the potentiometer with one of Arduino's analog inputs, which have a resolution of 1024 values (10 bits). For this purpose, we use the analogRead() function.

To control the motor, we use Pulse Width Modulation which has a resolution of 256 values (8 bits). For this purpose, we use the analogWrite() function.

Therefore, whichever value we measure in the analog inputs, we have to convert it to a proportional value inside the PWM range of values.

Using the map function, this is as simple as this:

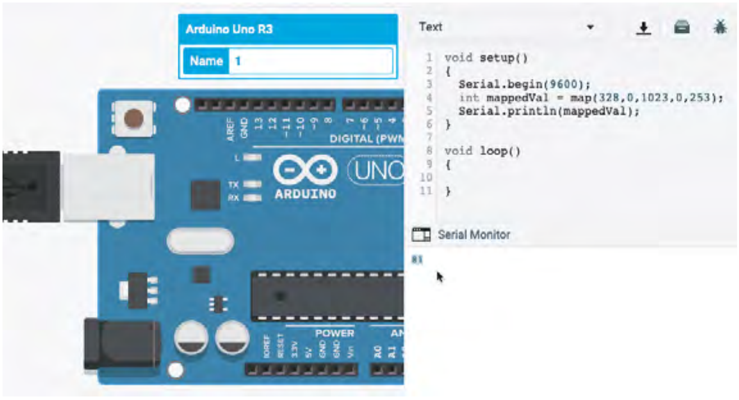
int mappedVal = map(analogRead(0),0,1023,0,254);

Say that the analogRead() function read the value "328" from the potentiometer. The call to the map() function will look like this (I have replaced the call to analogRead() with the explicit value "328"):

int mappedVal = map(328,0,1023,0,253);

The result of the mapping will be 81, which will be stored in the mappedVal variable.

Would you like to try this yourself? Here's my Tinkercad project. It contains a virtual Arduino Uno with a simple sketch that contains the map() function example.



```
void setup()
{
  Serial.begin(9600);
  int mappedVal = map(328,0,1023,0,253);
  Serial.println(mappedVal);
}

void loop()
{

}
```

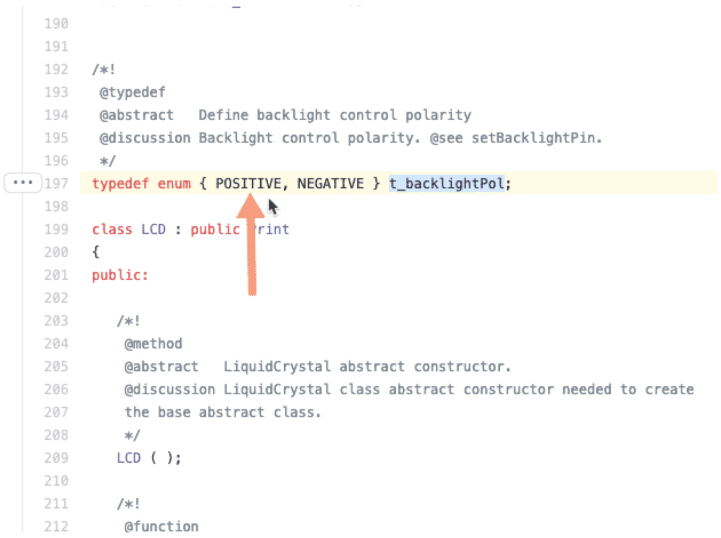
```
81
```

9. Confusing keywords? follow the source code trail

Arduino programming guide series

Confusing keywords? follow the source code trail

A student asked me: What is the meaning of a “weird” keyword used in a constructor? It was the keyword “POSITIVE” in one of the constructors from the LiquidCrystal_I2C class.



```
/*!
 @typedef
 @abstract   Define backlight control polarity
 @discussion Backlight control polarity. @see setBacklightPin.
 */
typedef enum { POSITIVE, NEGATIVE } t_backlightPol;

class LCD : public Print
{
public:

   /*!
    @method
    @abstract   LiquidCrystal abstract constructor.
    @discussion LiquidCrystal class abstract constructor needed to create
    the base abstract class.
    */
   LCD ( );

   /*!
    @function
```

This keyword indeed is not part of the C or C++ language.

So what could it be?

It is a keyword that has been defined in an included class or the current sketch.

To find out exactly where it is defined, you must follow the trail of added classes, load the open-source code in your text editor, and find out where the mysterious keyword is defined.

I have converted this student question into an example so I can show you how you can go about finding the origin of a keyword that you can readily recognize.

Here is the exchange between the student an myself:

Student question (verbatim):

My IDE is not accepting the keyword POSITIVE in the lcdi2c declaration line of code.

Why do we insert the library in this foramt("RTClib.h")rather than this format(<RTClib.h>)?

Student in Arduino Step by Step

Here's my response:

You probably have the wrong library installed. There are a few around with the same name.

The constructor that you refer to accepts keywords for controlling the backlit. In line 197 of the LCD.h file, these keywords are defined using a typedef construct (more about this here). The name of this typedef is *t_backlightPol*.

In line 61 of the LiquidCrystal_I2C.h, which is the constructor we use in the sketch, this keyword (POSITIVE) is used via the *t_backlightPol* typedef. It's just easier to remember and understand what the purpose of the parameter is when we use simple words instead of things like 1, 0, HIGH, LOW.

The pre-processor of the compiler (the program that works out where the different parts that make up your program are, as well as a few other things) will decide how to insert the file that you want to include into your program based on whether the name is within brackets or double-quotes.

If you use brackets, then the pre-processor will look for a header file and insert it into your program at the location of the include statement.

If you use a double-quote, then the pre-processor will look for the file (header or not, it doesn't matter), and also insert it into your program at the location of the include statement.

In the case of *#include "RTClib.h"*, we include a header file so I guess we could have used *#include <RTClib.h>* instead (you may want to try and see if that works).

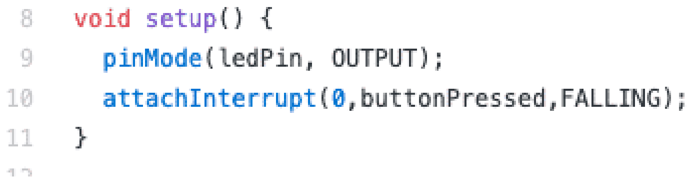
The details are available in the C standard document (PDF – N1570 Committee Draft, April 12, 2011). See section 6.10.2, "Source file inclusion." The difference between the two methods is very subtle, and my self-don't fully understand it (nor I am curious to do). The preprocessor is responsible for the implementation of this standard, and I know that different preprocessors follow slightly different approaches.

10. The interrupt service routine and volatile variables

Arduino programming guide series

The interrupt service routine and volatile variables

Interrupt service routines (ISR) must be as small as possible. If your sketch has to do time-consuming work after an interrupt event, you can use volatile variables to capture data and process it outside of the ISR.



```
void setup() {
  pinMode(ledPin, OUTPUT);
  attachInterrupt(0,buttonPressed,FALLING);
}
```

If your sketch is trying to do too many things inside an interrupt service routine, it may will run into problems that will be very difficult to troubleshoot.

For example, trying to update a port expander (such as the MCP23017 from Microchip) from an ISR will fail.

Why?

The problem with doing a write operation to the expander is that the operation may be take too much time to complete through the SPI interface.

In general, the workload inside an interrupt service routine should be very small. How much is “very small”? Think of operations as simple as updating an microcontroller register or memory location.

That’s it.

Operations like writing to the serial monitor that take more than a few processing cycles should be avoided.

In place of longer-running operations inside the ISR, you should write code that simply updates a volatile variable. When your sketch returns to the loop() function, it can read the data from the same volatile variable and process it as needed.

In other words, a volatile variable allows you to get data out of an ISR, to the rest of your sketch. This way, you can do the longer processing you need, without increasing the footprint of the ISR.

To declare a volatile variable, use the “volatile” keyword, like this:

```
volatile int temp = 0;
```

Then, you can use it like any other variable. Inside the ISR:

```
void isr() { temp = analogRead(0); //Keep this block very short! }
```

Finally inside the loop():

```
loop() { Serial.println(temp); // plus lots more code}
```

The volatile keyword informs the compiler that the flagged variable can change at any time, from any part of the sketch or program. In the Arduino that only has one processing unit and no true parallel programming is possible, this happens in the interrupt service routine only.

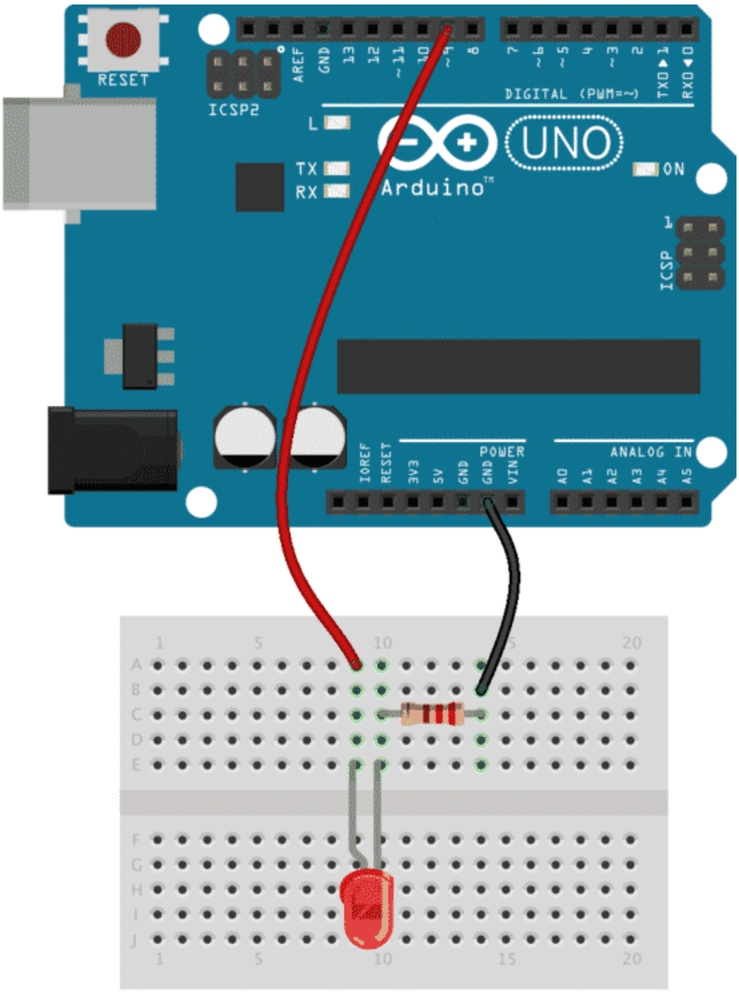
If you want to learn more about port expanders, interrupts and the volatile keyword, consider our comprehensive course Arduino Step by Step Getting Serious.

11. The problem with delay()

Arduino programming guide series

The problem with delay() and how to fix it

Microcontrollers have scarce resources. Perhaps their most important resource is compute time, followed by RAM and Flash memories. Therefore, we should not waste our MCU's time. But when you use delay(), you actually waste time.



The Arduino delay() function is a convenient way to delay the execution of a sketch. It is an intuitive instruction that most learners seem to be able to grasp and use without difficulty.

It something drilled to new Arduino makers early on, when they get their first LED to blink:

void loop() { digitalWrite(13, HIGH); **delay(1000);** // <– here it is, delay for a second digitalWrite(13, LOW); delay(1000); }

The delay() function makes programming easy. In the example above, we are simply making an LED blink.

No big deal.

You Arduino doesn't have anything else to do, anyway.

But, what if your Arduino had more work to do?

It might need to check a sensor and communicatee with another Arduino.

It might also need to record sensor data to an Internet of Things service and turn on a relay.

With the 1000ms delay that we have imposed with the delay() function, the Arduino is actually forced to do nothing (other than counting milliseconds) twice, in a single loop.

That is a waste of computing cycles!

The problem with the delay() function is that it is "blocking." The functions blocks the execution of any other code, except for interrupt service routines.

Obviously, this is not an efficient way to do programming.

We can improve the situation by exchanging simplicity with somewhat more complicated code. In the example below, I use a construct that checks periodically to determine if a given amount of time has elapsed. If it has, then it toggles the LED. If not, the execution continues.

```
1. const int ledPin = 13; // the number of the LED pin2. int
ledState = LOW; // ledState used to set the LED3. unsigned
long previousMillis = 0; // will store last time LED was
updated4. const long interval = 1000; // interval at which to
blink (milliseconds)5. void setup() {6. pinMode(ledPin,
OUTPUT);7. }8. void loop() {9. unsigned long currentMillis =
millis();10. if (currentMillis - previousMillis >= interval) {11.
previousMillis = currentMillis;12. if (ledState == LOW) {13.
ledState = HIGH;14. } else {15. ledState = LOW;16. }17.
digitalWrite(ledPin, ledState);18. }19. }
```

When this code runs on the Arduino, it will make the LED blink on and off once every second. It is the exact outcome of the version of the code that uses the delay() function, except that it works without blocking other code.

Let’s drill into the code.

- Line 10: check if the time interval has elapsed.
- Line 11: if the set time interval has elapsed, record the current millis reading (this will be used in the next interval check in line 10).
- Line 3: We use a variable of type long for this since we use the function millis() to get a current time reading.
- Lines 12 to 16: toggle the LED on/off depending on the value of the *ledState* variable.

The *millis()* function returns a value that represents the number of milliseconds since the Arduino was powered up. In other words, *millis()* gives us a way to track the passage of time; it is not able to tell us what time it is. Since we are interested only in elapsed time, not a time and date, this reading is good enough.

In line 10 the sketch subtracts the *previousMillis*(the time we last checked the elapsed time) from the *currentMillis* to work out whether the required interval has elapsed, and if it has, the sketch will toggle the LED.

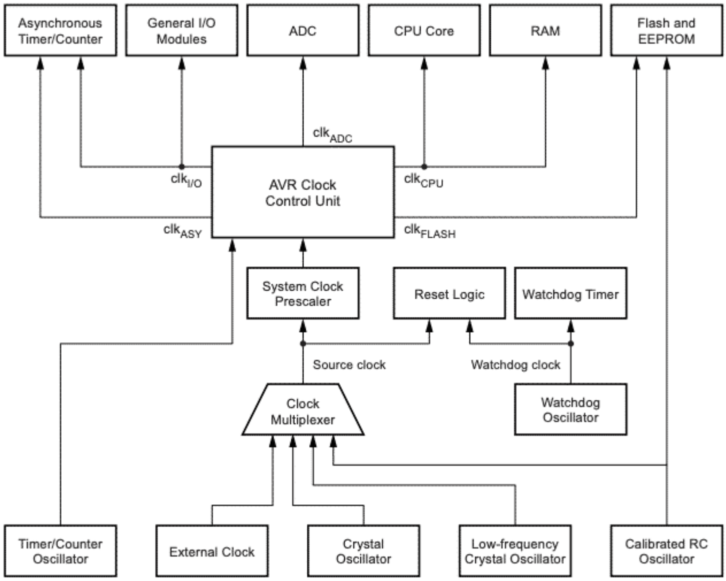
If the interval has not elapsed, then the sketch will continue with other instructions, so the cost of this operation is only a couple of compute cycles; this is a massive gain in efficiency. When you work on busy sketches that flash LEDs or check sensors at regular intervals, this approach is worth considering.

12. How to deal with the millis rollover

Arduino programming guide series

How to deal with the millis rollover

The Arduino contains a 32-bit register that is actually a counter. It counts the number of milliseconds elapsed since the time you powered-up the Arduino. We use this counter to count time. But, what happens when the counter reaches its maximum value? Let's figure it out with the help of an example.



> *The image above is borrowed from the Atmega328P datasheet, page 24. It depicts the clock distribution block diagram.*

In a separate article in this series, I explained that the problem with the delay() function is that it blocks execution. As a result, your Arduino is stuck at counting milliseconds instead of doing useful work.

A solution to the delay() blocking problem looks like this:

```
if (currentMillis - previousMillis >== interval) {
previousMillis = currentMillis; //... work to be done
periodically goes here }
```

… where *previousMillis* is an unsigned int that stores reading from the last call to *millis()*, and *currentMillis* contains the *millis()* reading for this check.

Apart from the additional code that you need to support this option, one significant problem you have to deal with is that the millis register will roll-over after around 50 days. That means that its register (that holds an unsigned long has a width of 4 bytes) will return to zero, and then start counting again towards its maximum value, 4,294,967,295 (= 2^32)

So how can we deal with this problem?

It turns out that this is not a problem at all. This code can deal with the millis register rollover without any modification.

Let's have a quick look at why it works, by considering a rollover situation.

The millis register is 4 bytes in width, so the largest unsigned number it can hold is:

11111111 11111111 11111111 11111111.

Let's say that we are interested in tracking a duration of 10 seconds.

That's 10,000 millis.

This duration, in binary, is (I keep the duration value also as a long int):

00000000 00000000 00100111 00010000

In the next couple of steps, I will calculate the difference between the current millis and the last millis at two times:

- before the rollover,
- after the rollover.

Before the rollover, the *previousMillis* variable will contain a number smaller than the millis max, and smaller than the current max.

Let's make:

previousMillis = 11111111 11111111 10110001 11011111

In the decimal system, *previousMillis* contains 4,294,947,295.

If we test the elapsed time at 5,000 millis after previous Millis was taken, we will have this subtraction:

111111111 11111111 10001010 1100111 – 11111111 11111111 10110001 11011111 = 1001110001000

In decimal notation, this is:

3,000 – 4,294,947,295 = 23,001 > 10,000

So, the duration of 10 seconds has elapsed and despite the rollover, we were able to detect it.

Remember that these values are unsigned, so the subtraction does not yield a negative number, but a positive that is the result of the difference between 4,294,947,295 and 4,294,967,295 (the max value for a 4-byte unsigned number) **plus** 3,000.

If all this looks complicated and you are not convinced that the rollover can be dealt with without any special provisions, use your Arduino and run this sketch on it:

00000000 00000000 00001011 10111000 – 11111111 11111111 10110001 11011111 = 00000000 00000000 01011001 11011001

In decimal, this is:

3,000 – 4,294,947,295 = 23,001 > 10,000

So, the duration of 10 seconds has elapsed. Despite the rollover, we were able to detect this. Remember that these values are unsigned, so the subtraction does not yield a negative number, but a positive that is the result of the difference between 4,294,947,295 and 4,294,967,295 (the max value for a 4-byte unsigned number) PLUS 3,000.

If all this looks too complicated and you are not convinced that the rollover can be dealt with without any special provisions, use your Arduino and run this sketch on it:

```
void setup() { Serial.begin(9600); // Prior to rolling over
unsigned long currentMillisA = 4294952295; // An earlier
time than currentMillis unsigned long lastMillisA =
4294947295; Serial.print("1. Difference: "); // Find out the
difference // between the two times
Serial.println(currentMillisA-lastMillisA); Serial.println(); //
Rolled over unsigned long currentMillisB = 3000; // Prior to
rolling over unsigned long lastMillisB = 4294947295;
Serial.print("2. Difference: "); // Find out the difference //
between the two times Serial.println(currentMillisB-
lastMillisB); }void loop() { }
```

This sketch does the same calculation I described earlier.

In the first subtraction, both current millis and last millis are before the rollover.

In the second subtraction, the millis register has rolled over, and the current millis is 3000.

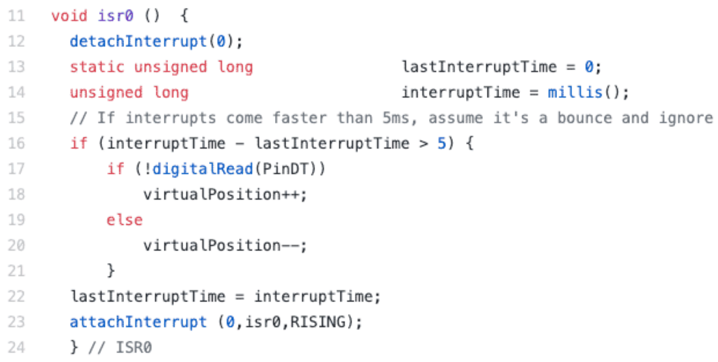
Feel free to try out different value to see how they are handled.

13. Can you use delay() inside Interrupt Service Routine?

Arduino programming guide series

Can you use delay() inside Interrupt Service Routine?

An interrupt service routine (ISR) looks like a regular function. It can hold any code you want, and it will work as it would in any other function. But, the ISR is not a regular function, and you should treat it as special.



```
void isr0 ()  {
  detachInterrupt(0);
  static unsigned long                  lastInterruptTime = 0;
  unsigned long                         interruptTime = millis();
  // If interrupts come faster than 5ms, assume it's a bounce and ignore
  if (interruptTime - lastInterruptTime > 5) {
      if (!digitalRead(PinDT))
          virtualPosition++;
      else
          virtualPosition--;
      }
  lastInterruptTime = interruptTime;
  attachInterrupt (0,isr0,RISING);
  } // ISR0
```

The ISR is a function that is registered to be called when an interrupt event occurs. For example, this interrupt can be a change in the state of an interrupt-capable pin, or the expiration of a timer (here's an example sketch of a timer interrupt).

A question I often receive is about the contents of the ISR:

- "Can I include arbitrary code?"
- "Can I include slow functions like delay() or even Serial.print()?"

Although this is possible, you should not.

Both *delay()* and *Serial.print()* functions are "blocking". This means that they stop anything else from happening in the Arduino.

The *delay()* function will make the Arduino stop until your specified interval has expired. The *Serial.print()* function will also make the Arduino stop until the entire message has been printed to the serial monitor at the slow communication speeds of the serial interface.

An interrupt service routine should be as light as possible so that it can service an interrupt quickly. The objective is to allow the Arduino to continue doing what it was doing before the interrupt.

If you use a *delay(5)* inside the ISR, you will be blocking the processor for at least 5ms, which for a computer is a lot of time.

If you need to check the passage of time inside the ISR, it is much better to use an "if" statement, like I discussed in a separate article.

The Arduino will be able to process the "if" statement within a few nanoseconds, and quickly return control of the program to the code outside of the ISR.

Here is an example sketch showing how to write an ISR that uses the “if” statement to assess the passage of time. This example is from Arduino Step by Step Getting Serious, where I show how to use a rotary encoder.

I have highlighted the relevant block in bold.

You can see this code on Github.

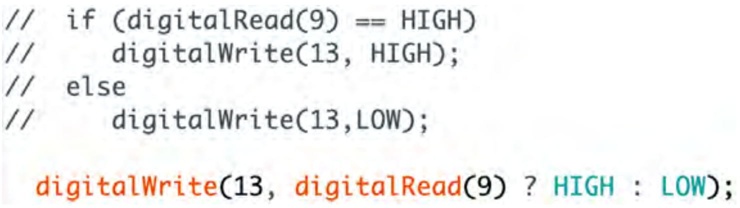
//Original sketch: https://bigdanzblog.wordpress.com/2014/08/16/using-a-ky040-rotary-encoder-with-arduino///Modified by Peter Dalmaris, July 2015const int PinCLK=2; // Used for generating interrupts using CLK signalconst int PinDT=3; // Used for reading DT signalconst int PinSW=8; // Used for the push button switchvolatile long virtualPosition =0; // must be volatile to work with the isrvoid isr0 () { detachInterrupt(0); static unsigned long lastInterruptTime = 0; unsigned long interruptTime = millis(); // If interrupts come faster than 5ms, assume it’s a bounce and ignore **if (interruptTime – lastInterruptTime > 5) { if (!digitalRead(PinDT)) virtualPosition++; else virtualPosition–; }** lastInterruptTime = interruptTime; attachInterrupt (0,isr0,RISING); } // ISR0void setup () { pinMode(PinCLK,INPUT); pinMode(PinDT,INPUT); pinMode(PinSW,INPUT); attachInterrupt (0,isr0,RISING); // interrupt 0 is always connected to pin 2 on Arduino UNO Serial.begin (9600); Serial.println(“Start”);}void loop () { int lastCount = 0; while (true) { if (!(digitalRead(PinSW))) { // check if pushbutton is pressed virtualPosition = 0; // if YES, then reset counter to ZERO while (!digitalRead(PinSW)) {} // wait til switch is released delay(10); // debounce Serial.println(“Reset”); // Using the word RESET instead of COUNT here to find out a buggy encoder } if (virtualPosition != lastCount) { lastCount = virtualPosition; Serial.print(“Count:”); Serial.println(virtualPosition); } } // while}

14. The ternary operator

Arduino programming guide series

The ternary operator

As you become more skilled in programming, you will begin to notice that style matters almost as much as functionality. An element of style in programming is the ability to shrink code without altering its functionality. This results in smaller, more concise programs.



```
//  if (digitalRead(9) == HIGH)
//      digitalWrite(13, HIGH);
//  else
//      digitalWrite(13,LOW);

  digitalWrite(13, digitalRead(9) ? HIGH : LOW);
```

I'll explain with the help of an example.

We often find ourselves writing code like this:

if (digitalRead(9) == HIGH) digitalWrite(13, HIGH); else digitalWrite(13,LOW);

An alternative way of writing the same functionality is this:

digitalWrite(13, *digitalRead(9) ? HIGH : LOW)*;

It is shorter, with all functionality included in a single line. Some, might say, it is beautiful.

The short version of the code contains a *ternary operator*, marked in italics and underlined.

A ternary operator looks like this:

OPERATOR ? TRUE EXPRESSION : FALSE EXPRESSION ;

The OPERATOR contains a regular boolean expression, like “analogRead(0) < 500”.

If this expression returns TRUE, then the TRUE EXPRESSION after the “?” delimiter will be executed.

If the OPERATOR returns FALSE, then the FALSE EXPRESSION after the “:” delimiter will be executed.

Ternary operators can make your code shorter, and often reduce the chance of error because they allow for the complete expression to be written in a small amount of space, making it easier to inspect.

Of course, the expression for this particular example can be shortened even further.

Because the *digitalRead()* function returns a boolean, you can write this:

digitalWrite(13, digitalRead(9));

The example I show here is trivial, and the use of a tertiary operator is not truly necessary.

But, what if you had a more complicated case?

Let’s say that you needed to call an appropriate function depending on the state of digital pin 9. In that case, the ternary operator would be useful and your code would look like this:

```
digitalRead(9) ? functionA() : functionB();
```

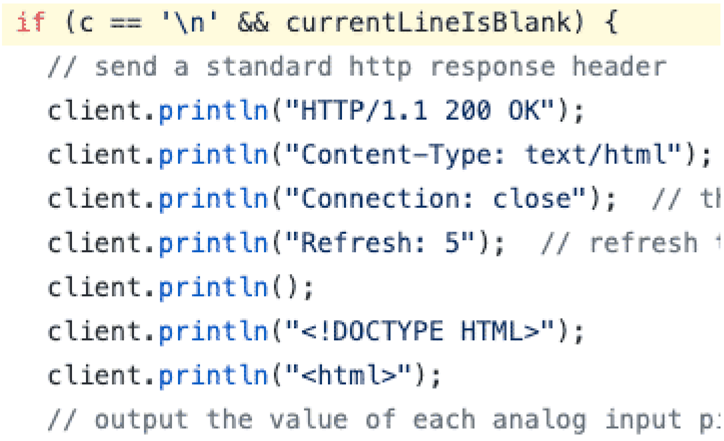
As you can see, the C language offers quite a few options to help you produce concise (and beautiful) code.

15. A closer look at line feeds and carriage returns

Arduino programming guide series

A closer look at line feeds and carriage returns

If you have an Arduino application that must parse an HTTP response, it helps to understand the difference between the two most common special characters: line feed and carriage return. Once you understand this, you will be able to reduce the complexity of your parser by half.



```
if (c == '\n' && currentLineIsBlank) {
  // send a standard http response header
  client.println("HTTP/1.1 200 OK");
  client.println("Content-Type: text/html");
  client.println("Connection: close");  // tl
  client.println("Refresh: 5");  // refresh
  client.println();
  client.println("<!DOCTYPE HTML>");
  client.println("<html>");
  // output the value of each analog input p:
```

When writing Arduino sketches that parse HTTP requests, you come across code like this:

if (c == ‘n’ && currentLineIsBlank) { client.println(“HTTP/1.1

200 OK”); client.println(“Content-Type: text/html”); client.println(“Connection: close”);

In this code, there’s a special character: “n”.

This special character is a line feed.

In some cases, you may also see the Carriage Return (“r”) special character.

In this article, I will discuss Line Feed and Carriage Return because they seem to be causing a lot of confusion to learners.

In a nutshell:

- “n” is the *newline* character, or “line feed (LF).” It is the character that marks the end of a line and the beginning of a new line. In ASCII code, the new line character is encoded as decimal 10.
- “r” is the *carriage return* (CR) character. This character returns the cursor to the beginning of the line, as opposed to creating a new line (this is what line feed does). In ASCII code, the carriage return character is encoded as decimal 13.

In modern computers (as opposed to mechanical typewriters where these operators find their roots), how exactly LF and CR work depend on the operating system.

In Unix and Unix like systems, like Mac OS X and Linux, an LF creates a new line and returns the cursor to the start of the line.

In Windows, to achieve the same outcome, you must combine CR with LF. This will result in, first, the generation of a new line, and then, the return of the cursor to the start of the line.

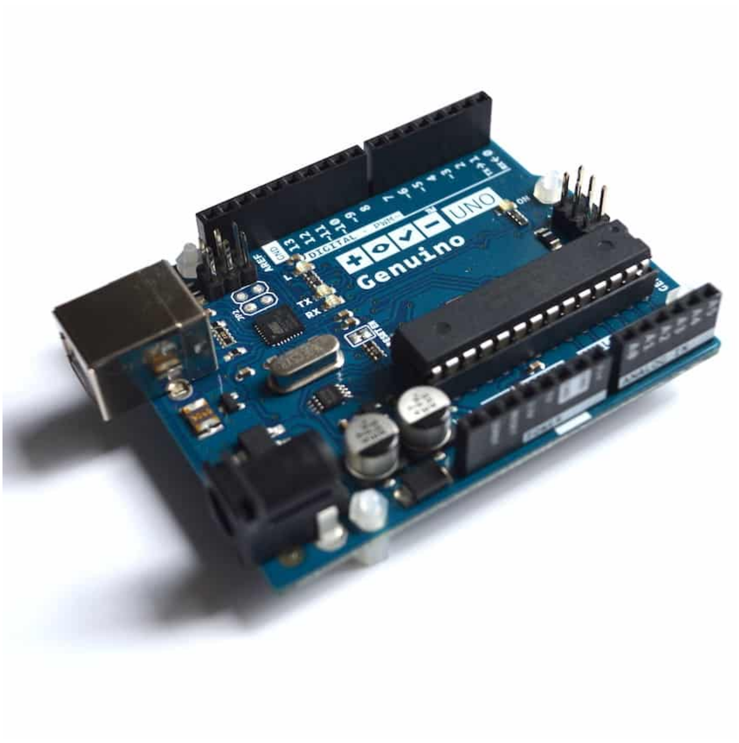
Which of these special characters should you use in your Arduino sketches?

Well, it depends on the application (you probably expected this).

For example, if the application involves the parsing of a web page, which is encoded in an HTTP message, and we want to know when the header has been received, then we can look for a CR + LF combination. We search for CR + LF because this is required by the HTTP standard.

But here's a simple trick: because in HTTP messages, CR and LF almost always appear together, we can reduce the workload of the Arduino by trying to detect one or the other. This realisation cuts not only the workload, but also the complexity of the parsing code.

Another thing to consider is that the HTTP standard is tolerant regarding the use a line feed (LF) to mark the end of a line; this is not strictly correct, but it is widespread and works just fine. This is why in Arduino sketches that parse HTTP messages (like the one in the top of this article), you see that the code will try to detect line feeds (LF, "n"), and ignore carriage returns (CR).



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Learn more

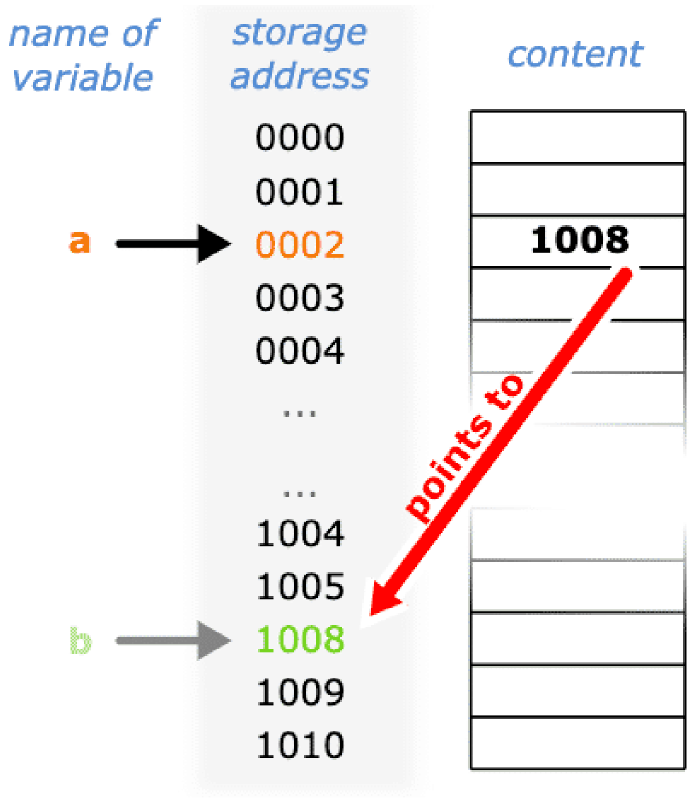
Jump to another article

16. Understanding references and pointers

Arduino programming guide series

Understanding references and pointers

The topic of memory pointers in C and C++ is a known cause of intense headaches for many of us. It is the one topic that will most often scare people away from C/C++ and into “higher level” languages like Python and Ruby. But, with a bit of patience, you can understand it.



Attribution of the image above: This file was made by User Sven. Original: https://commons.wikimedia.org/w/index.php?curid=8861432

Even though microcontrollers are becoming increasingly powerful and able to support languages like Python, Lua and Javascript, the traditional C/C++ toolbox is still dominant and not going anywhere. With a bit of effort, you can learn how to use C and C++.

And yes, that includes figuring out how memory pointers work!

In this article, I will help you understand how the "*" and "&" memory operators work with the help of a simple example.

I will use this sketch below (you can also see it on Github):

[tcb-script src="https://gist.github.com/futureshocked/c548c64abb5caced4bdc4981ec02df1b.js"][/tcb-script]

In the sketch, I manipulate primitive and object variables by **reference**, **pointers**, and **value**.

Every time I make a change to a stored value, pointer or reference, I print out the remaining memory on the Arduino; this allows us to get a glimpse of the effect that a particular operation has on the memory, and in particular the static RAM (SRAM).

A few things are worth noting here:

The reference operator "**&**" is used to extract the memory location where the data of a variable is stored.

For example, look at line 29:

pointer = &a;

Here, we are getting the memory location where the value for the int variable "**a**" is stored, and we store this memory location to pointer "**pointer**".

The pointer operator "*****" designates a variable as a pointer. This means that whatever is stored in this variable is a pointer to another memory location. Again in line 29, the variable “**pointer**” will not contain the value that is stored in “**a**“, but the memory location that we stored in variable “**a**“.

You can re-assign pointers just like any other variable. For example, in line 40, I re-assign the “**pointer**” variable to point to int “**b**“, instead of the original “**a**“.

When it comes to passing values to functions, it can be done in three ways: by **reference**, **pointer**, or by **value**.

When you pass an actual value to a function (instead of a reference or pointer to that value), you are creating a **copy** of that value that “lives” inside the context of the function.

When the value is a primitive (like an int or a char), this not too expensive in terms of memory utilization.

But when it is an object, like a String, then the memory consumption can be significant.

The lesson to take home: pass object values to functions as a reference or as a pointer.

See the examples in lines 50 and 56, and note how these target functions are declared.

If you want to pass a primitive, you can consider using references or pointers if the size of the primitive is larger than the size of the pointer.

In the Arduino Uno, an int takes up 2 bytes in SRAM. The SRAM address uses 1 byte. Therefore, by using a pointer to reference a value, you can save 1 byte of SRAM.

A double primitive consumes 4 bytes, so by using a pointer you can save 3 bytes.

What’s better: **references** or **pointers**?

My research does not indicate strongly that one is better than the other. I personally prefer references.

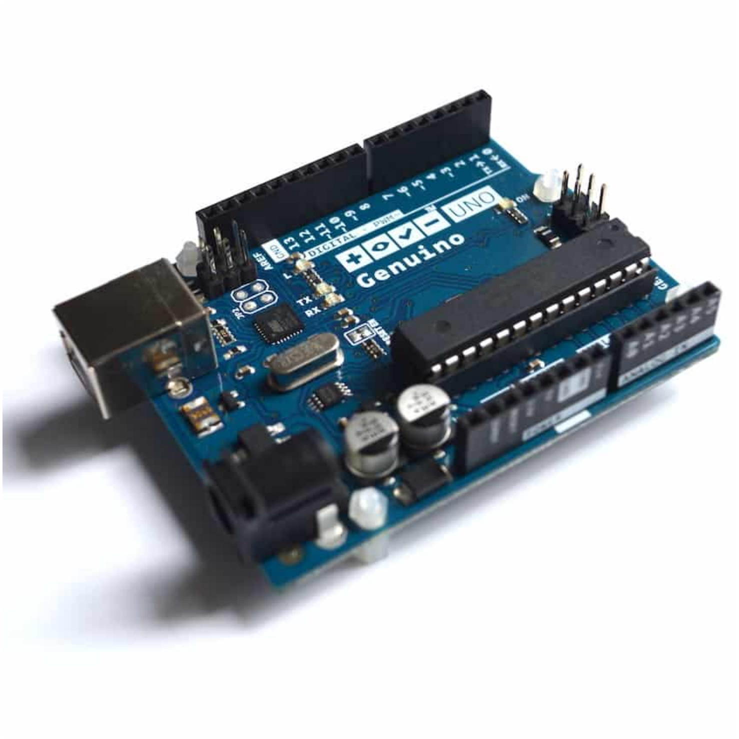
Have a look at line 84, and compare it with line 89. I think that the calling code in line 84 looks cleaner. There seems to be no difference in terms of memory or compiler efficiency.

Feel free to play around with the sketch. Experiment with it by making changes to the memory operators so you can get a first-hand impression of the memory footprint of the examples.

Experiment output:

```
freeMemory()=1774 freeMemory()=1774 1 —– a —– a=5 a=6
freeMemory()=1774 2 —– b —– b=10 b=11
freeMemory()=1774 3 —– Passing by reference —–
6freeMemory()=1770 3 —– Passing by reference —– 11
freeMemory()=1770 4 —– Passing by pointer —– 11
freeMemory()=1770 4 —– Passing by pointer —–
6freeMemory()=1770 5 —– Passing by value—–
6freeMemory()=1770 5 —– Passing by value—– 11
freeMemory()=1770 6 ——– String a —– string a=Hello
World! freeMemory()=1744 7 ——– String b —– string
b=Hello again! freeMemory()=1744 8 —– Passing String by
reference —– Hello World! freeMemory()=1740 8 —– Passing
String by reference —– Hello again! freeMemory()=1740 9
—– Passing String by pointer —– Hello World!
freeMemory()=1740 9 —– Passing String by pointer —–
Hello again! freeMemory()=1740 10 —– Passing String by
value—– Hello World! freeMemory()=1725 10 —– Passing
```

String by value—– Hello again! freeMemory()=1725



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

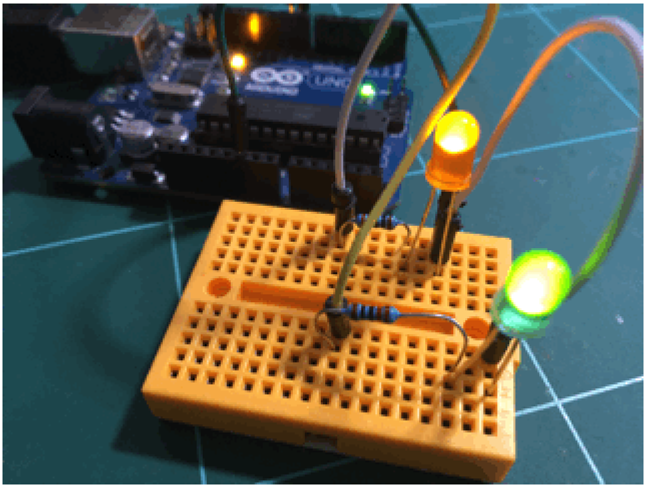
Learn more

17. Simple multitasking on the Arduino

Arduino programming guide series

Simple multitasking on the Arduino

Simple multitasking on the Arduino



The bulk of CPUs used in desktop or laptop computers ten or fifteen years ago where also mostly single-core, as opposed to modern multi-core systems. Despite this "handicap", they were able to execute multiple processes at, seemingly, the same time (concurrently). You could have your web browser rendering a web page while your email client was downloading a bunch of emails.

How was this possible?

And if it was possible to have multitasking on a single-core computer CPU ten years ago, why not have the same ability on a single-core microcontroller?

The key to multitasking is efficient and clever programming. Thanks to multitasking infrastructure integrated into modern operating systems, it was possible to divide CPU time into small chunks and allocate each chunk to a different process. The operating system was responsible for allocating these time chunks to the processes that needed them based on various scheduling algorithms. Even though the resources that a desktop CPU has been vastly superior to those that a microcontroller has, the key ingredient is the same: time.

The Arduino has no operating system.

Therefore, if we want to implement multitasking, we will have to create at least a basic infrastructure that supports it. We also have to create a sketch that also implements the functionalities we need.

In a desktop operating system, scheduling does exactly what the word says:

1. *Start Process A at time X and stop it after Yms.*
2. *Start Process B at time X+5 and end it at time Zms.*

And so on.

Let's say that Process A is turning a red LED on (connected to pin 9), and Process B is turning a green LED on (connected to pin 8).

Usually, if we want to turn an LED on for 100ms, then turn it off, we would do it like this?

digitalWrite(9,HIGH);delay(100);digitalWrite(9,LOW);

The LED will turn on, then 100ms later will turn off. The microcontroller will be locked for anything other than an external interrupt for 100ms.

If we needed to turn on the green LED 5ms after the red LED turned on, we would not be able to do it using delay() because the microcontroller is still counting milliseconds for the red LED.

We will have to wait until the red LED is turned off, a whole 95ms after then the actual time that we wanted the green LED to turn on:

digitalWrite(9,HIGH);delay(100);digitalWrite(9,LOW);digital Write(8,HIGH);delay(100);digitalWrite(8,LOW);

This is an example of how using delay() forces us to implement strictly single-processing systems, and how it forbids certain functionalities, like turning the green LED on while the microcontroller is locked in the delay() function.

In a separate article, I explained how to use a common technique that can help us to avoid the use of the delay() function. By avoiding the delay function, we can utilze the otherwise wasted compute cycles and get closer to a multi-tasking environment on an Arduino board.

Here, I would like to expand on what I wrote about delay() in that article and give an example of how you can use the technique I described there to implement basic multitasking.

In the example that follows, I use this technique to show you how to create a sketch that blinks two LEDs, red and green, according to my desired schedule. For example, turn the red LED on at time X, green LED on at time X+5, red LED off at time X+100, and green LED off at time X+103.

Notice: the actual timings in the sketch may be different, as I have been playing around with it.

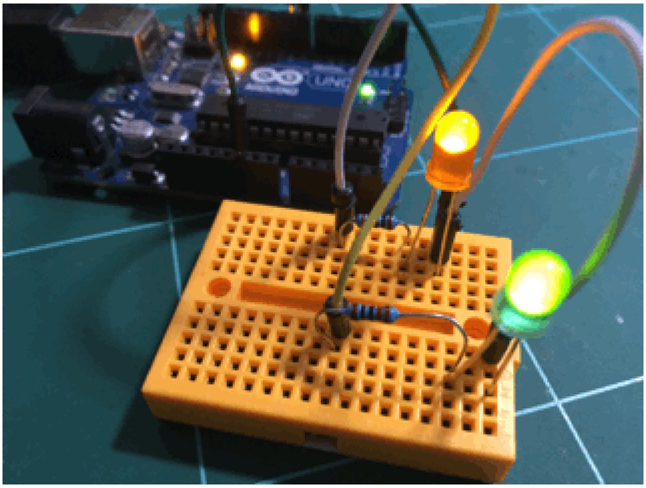
Total ON time for LEDA is 100ms, and for LEDB is 98ms.

Let's also make the total OFF LEDA time 200ms and for LEDB 250ms.

You can copy the sketch from the Gist on Github.

[tcb-script src="https://gist.github.com/futureshocked/d9b84613e0c14230369afd0bade9f5c5.js"][/tcb-script]

Run this sketch on an Arduino with two LED connected, and you will see something like this:



In this example sketch, for each LED (or time-dependent function that we want to implement), we use four variables to create a schedule:

- The total elapsed time for the ON LED state
- The total elapsed time for the OFF LED state
- The last time we made a change to the state of the led.
- The offset.

The pattern is this: if the time we want the LED to be in a specific state has elapsed, move it to the other state. The offset value is used to “push” the ON/OFF cycle forward in time; this is useful if you wish to schedule more than one activities to start at different times.

The result of this time of programming method is known as a “state machine.” A state machine is a program or a machine which can be only in one of several states at any given time. The specific state in which the machine will be depend on rules, which in turn rely solely on past and present conditions. There are state machines that have states that are deterministic, or non-deterministic.

For example, in our sketch, a rule determines that if 5ms have elapsed and LEDA if OFF, then LEDA should become ON.

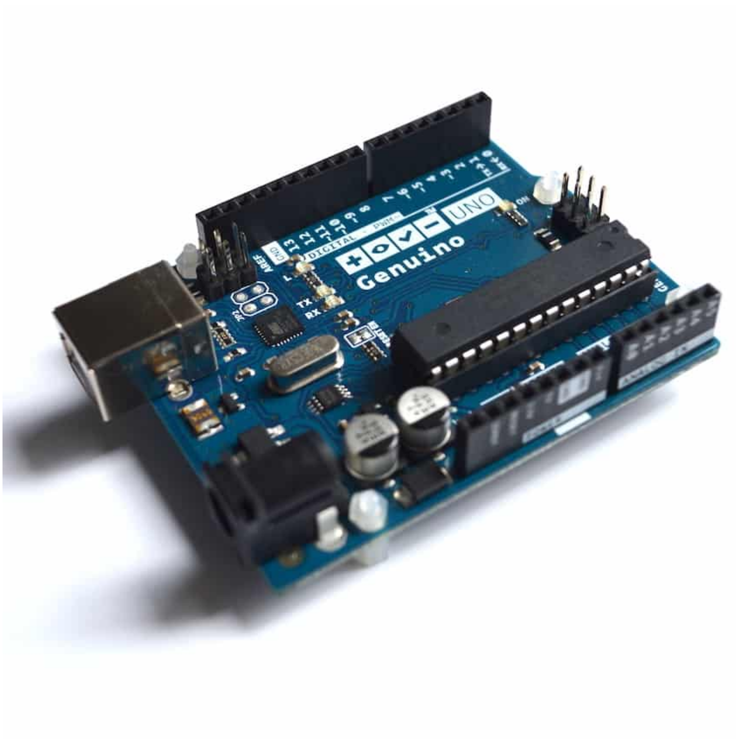
The state of this LED will not change unless another rule is triggered in the future which will take into account the state of the LED and of other variables that describe its environment (in our case, just the millis and the last time that the state of LEDA was changed) to decide if LEDA should be turned OFF.

Using the state machine paradigm, you can implement the efficiency of multitasking on your Arduino.

Indeed, this kind of manual multitasking is not as easy as multitasking is on the desktop. The absence of an operating system means that you, the programmer, have to design the scheduling rules and then implement them in code.

For a small number of states, this is a manageable problem.

What if you have more complicated requirements? Good news: There are ways to abstract multitasking on microcontrollers using libraries! But, this is something for another article in this series.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Learn more

Jump to another article

19. Concurrency with the Scheduler library on the Arduino Due and Zero

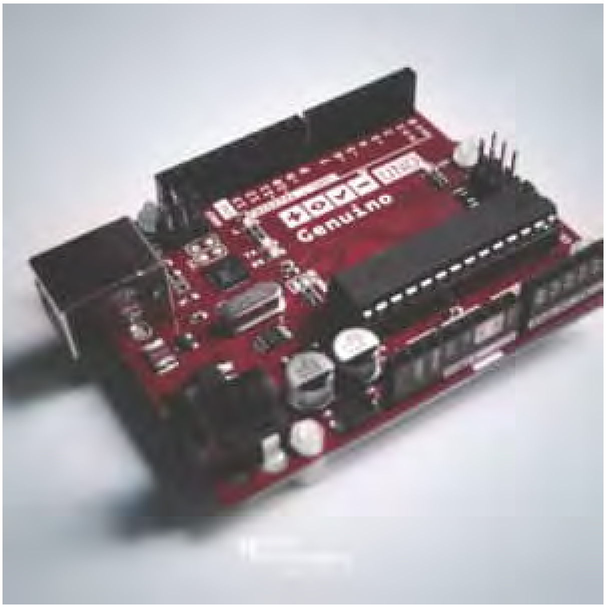
20. Bitshift and bitwise OR operators

21. What is a “static” variable and how to use it

22. Understanding the volatile modifier

23. Optiboot, a free upgrade for your Arduino

24. A real-time OS for the Arduino



Done with the basics? Looking for more advanced topics?

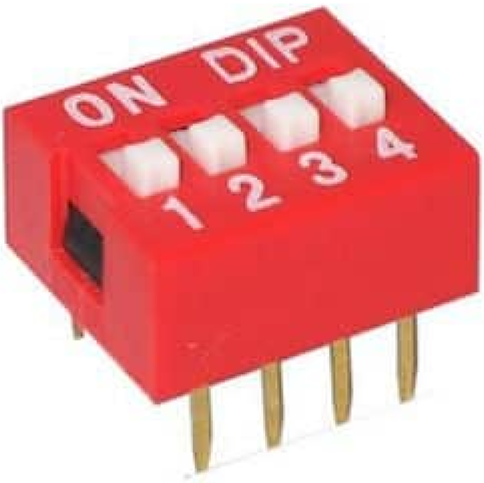
Arduino Step by Step Getting Serious is our comprehensive

18. Boolean arrays

Arduino programming guide series

Boolean arrays

It is a common practice to use arrays to store chars, ints, or double values.But arrays can also store booleans. Here's how.



A boolean data-type is one that can take only two possible values. Usually these values are depicted as "TRUE" or "FALSE". You can also see them as "ON" or "OFF", or "1" or "0".

You may intuitively think that a boolean data-type can be stored with a single bit, instead of a full byte. Let's correct this right now: A boolean data type may be one of two possible values, but it still takes a full byte to store it in

SRAM.

Still, even though boolean data-types will not save an SRAM, they are better suited than alternatives, like int or byte in many applications.

Let's look at an example.

Imagine that you have a set of dip switches connected to the digital pins of an Arduino.

Let's say that your dip switches are configured to produce the code "0101" in digital pins 3, 4, 5, 6.

You can capture this code and store it into a boolean array like this:

```
boolean dip_switch[4];dip_switch[0] = digitalRead(3); // a
boolean data typedip_switch[1] = digitalRead(4); // a boolean
data typedip_switch[2] = digitalRead(5); // a boolean data
typedip_switch[3] = digitalRead(6); // a boolean data type
```

The configuration of the DIP switches is now stored in an array of type “boolean“.

Boolean is a non-standard data type defines in the Arduino language, that is identical to the bool data type. In this example code, you could substitute “boolean” for “bool” without changing the outcome. In fact, the Arduino documentation recommends that you use “bool” instead of “boolean”.

If you want to treat the dip_switch array as a half-byte word, then you can convert it to a decimal. You can do this with this code:

```
byte encodebool(boolean* arr){ byte val = 0; for (int i = 0;
i<4; i++) { val << = = 1; if (arr[i]) val | = 1; } return val; }
```

If *dip_switch* contains “0,1,0,1”, then *encodebool(dip_switch)* will return “5”.

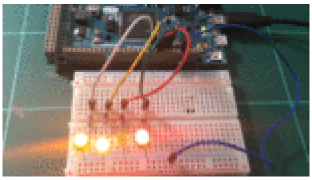
With this simple example, you can use a DIP switch, store the positions of its switches to a boolean array, and convert the half-byte set on the DIP switch to a decimal on demand.

19. Concurrency with the Scheduler library on the Arduino Due and Zero

Arduino programming guide series

Concurrency with the Scheduler library on the Arduino Due and Zero

The Arduino Due and Arduino Zero are far more powerful than the Arduino Uno. They use microcontrollers based on 32-bit ARM technology. With the help of the Scheduler library, you can use them as potent multitasking machines.



As I explained in a separate article on multitasking on the Arduino Uno, it is possible to use simple "if" structures and the millis() function to execute arbitrary functions at specific intervals.

Once the number of concurrent functions increases to more then two or three, this method becomes too complicated. At that point, you can use a more sophisticated approach.

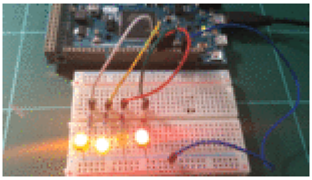
One such approach is implemented with the Scheduler library.

With the Scheduler library, you can set up multiple tasks within your sketch. Each task will be executed concurrently, and independently of each other. Remember that just like with the Arduino Uno, the Arduino Due and Zero are both single-core machines. Therefore, each task will be executed individually even though the effect is that they are all running concurrently.

The effect of concurrency amounts to the ability of the processor to move from one task to the other before it is completed and then to return later to continue where it left off.

Although you only have very basic control over the actual timing of execution, the greatest advantage of this library is its simplicity.

To understand what I mean by this, I have written a simple sketch to implement this functionality:



This is an Arduino Due controlling 4 LEDs. Each LED is toggling on/off on schedule, and each schedule is independent.

```
#include <Scheduler.h>int led_red = 11;int led_green = 10;int led_yellow = 9;int led_orange = 8;void setup() { pinMode(led_red, OUTPUT); pinMode(led_green, OUTPUT);
```

pinMode(led_yellow, OUTPUT); pinMode(led_orange, OUTPUT); Scheduler.startLoop(control_red); Scheduler.startLoop(control_green); Scheduler.startLoop(control_yellow); Scheduler.startLoop(control_orange);}void loop() { yield(); // This is important. With yield, control will be passed to one of the other tasks.}void control_red(){ digitalWrite(led_red, HIGH); delay(300); // Control is passed to one of the other tasks. digitalWrite(led_red, LOW); delay(300); // Control is passed to one of the other tasks.}void control_green(){ digitalWrite(led_green, HIGH); delay(400); // Control is passed to one of the other tasks. digitalWrite(led_green, LOW); delay(400); // Control is passed to one of the other tasks.}void control_yellow(){ digitalWrite(led_yellow, HIGH); delay(500); // Control is passed to one of the other tasks. digitalWrite(led_yellow, LOW); delay(500); // Control is passed to one of the other tasks.}void control_orange(){ digitalWrite(led_orange, HIGH); delay(600); // Control is passed to one of the other tasks. digitalWrite(led_orange, LOW); delay(600); // Control is passed to one of the other tasks.}

First, install the library to your IDE (it does not ship by default).

To set up a task, use the Scheduler's start loop function, and pass the name of the function that implements the task as the parameter. You can have as many of them as you like.

In this example, we set up four tasks. Each one controls an LED and turns it on and off.

In the *loop()* function, there is only one instruction: "*yield()*" this allows the control of the execution to be given to one of the other tasks.
Which one? The scheduler will decide. If you are curious

about the details, you can have a look at the library source code.

The gist is that the Scheduler implements cooperative multitasking. In simple terms, this means that the scheduler depends on the executing task to relinquish control; this is what the yield function does. It tells the scheduler that it is ok for another task to be executed at this point. The library will also use the delay function to "know" when it is ok to transfer control to another task.

For the end user, me and you, this means that to use the Scheduler library we only need to use the yield() and delay() functions to signal to the scheduler that control can be transferred to another task. We can use these functions at any point in the sketch where it makes sense.

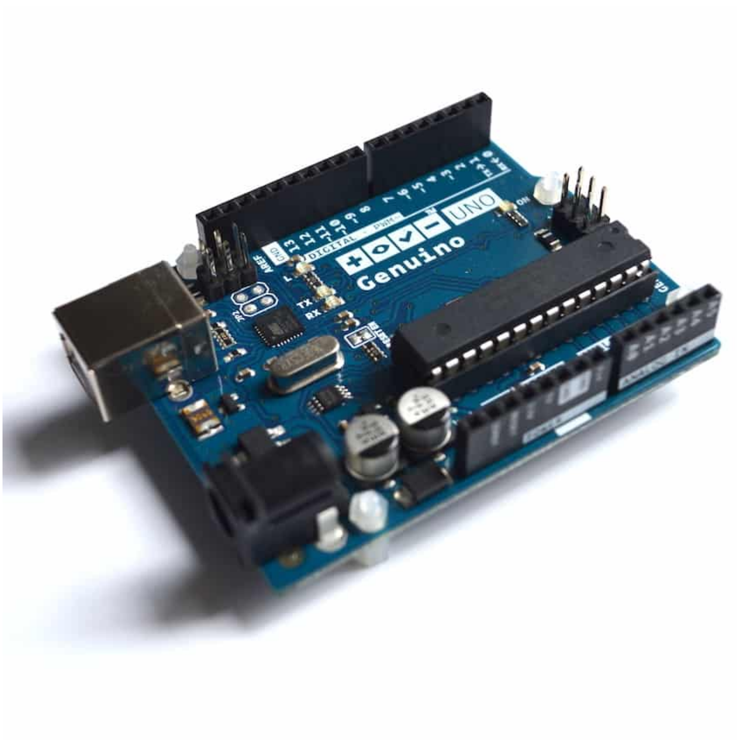
For example, have a look at function *control_orange*. I am making an LED blink, and as usual, I use the delay function to keep the LED at a particular state for, in this case, 600ms. If we were not using the Scheduler library, at this point the complete program would block the execution. Nothing else would happen until this delay was complete.

But in this example, we are using the Scheduler library, so instead of the program blocking, the control is simply transferred to another task.

This way, CPU wasted cycles are minimized, and you get a much more efficient design for minimal additional effort.

You can use the yield function instead of delay, with the same result, as I did in the loop function.

If you have a Due or a Zero around, I encourage you to play with the Scheduler library to become familiar with it. Think about how your next project can use it so you can take advantage of the efficiency it offers.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Learn more

Jump to another article

17. Simple multitasking on the Arduino

18. Boolean arrays

19. Concurrency with the Scheduler library on the Arduino Due and Zero

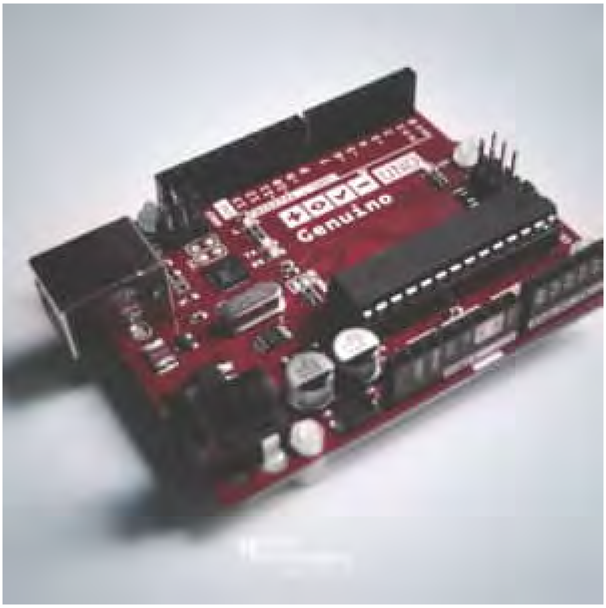
20. Bitshift and bitwise OR operators

21. What is a “static” variable and how to use it

22. Understanding the volatile modifier

23. Optiboot, a free upgrade for your Arduino

24. A real-time OS for the Arduino

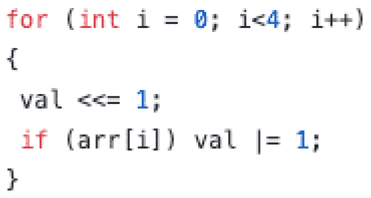


20. Bitshift and bitwise OR operators

Arduino programming guide series

Bitshift and bitwise OR operators

Programming a microcontroller entails much more bit-level manipulation than what is common in general computer programming. Bitwise operators like “<<“, “>>” and “|” are essential. Let’s look at two of the most common bitwise operators, bitshift right and bitwise OR.



```
for (int i = 0; i<4; i++)
{
 val <<= 1;
 if (arr[i]) val |= 1;
}
```

This is a follow up to a separate article (boolean arrays) in this series, where I discussed boolean arrays.

Some of the feedback I received in relation to that article asked for an explanation of what the operators “<<“ (bitshift left) and “|” (bitwise OR) do.

In this article, I will explain the functionality of these operators, drawing from the code in boolean arrays article.

First, I will remind you of the relevant part from the example code from the boolean arrays article.

[tcb-script src="https://gist.github.com/futureshocked/6117638a32dc36e0127c1cfd4d0fe935.js"][/tcb-script]

The symbol "<<" is the binary **bitshift left** operator.

It takes a number like "000**1**000 and shifts (moves) every bit in it to the left.

The number of shifts can be controlled through the parameter given on the right side of the operator. In this example, the parameter is "**1**", so the result of this operation will be "00**1**0000.

When you add the assignment "**=**" operator to the bit-shift operator, so that the two operators together are "**<<=**", then you have the **binary left shift with assignment** operator.

This will take the shifted value and store it in the "**val**" variable, on the left side of the expression.

The exact same, but in reverse, will happen if you use the **bitshift right operator** "**>>**".

Next, "**|**" is the bitwise **OR** operator.

When you add the "**=**" operator, you have this compound operator: "**|=**".

This is the **bitwise inclusive OR with assignment** operator.

In the example code, it works as shorthand for "**val |= 1**".

The full hand notation would be "**val = val | 1**". It is similar to the way that the expression "**val += 1**" is short for "**val =**

val + 1“.

Here’s an example:

If **val** is “000**1**000, then “**val | 1**” would be “000**1**00**1**” (since 1 = 0000001).

Therefore, “**val |= 1**” would result to **val** containing the value “000**1**00**1**”. Perhaps this make more sense if we arrange the OR calculation like this:

val 0 0 0 **1** 0 0 0 1 0 0 0 0 0 0 **1** —————val 0 0 0 **1** 0 0 **1**

In binary OR, 1|1 = 1, 1|0 = 1, 0|1 = 1, and 0|0=0.

When the function starts execution, **val** is initialized to 0. Inside the “for” loop, the first time that the bitshift operator is called has no effect on the value of val. Bitshifting “**0000000**” will still give you “**0000000**“.

But lets say that **arr[0] = 1**.

When the sketch hits the next line, “**if (arr[i]) val |= 1;**“, then val will become “**0000001**“.

And when the block loops back to the “for” instruction in line 4, and the bitshift operation is executed again (**“val <<= 1;”**), then **val** will become “**0000010**“.

And so on.

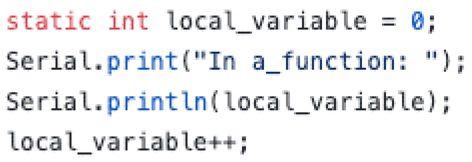
See how essentially the contents of the arr array are copied into the val byte variable, one bit at a time?

21. What is a static varia ble and how to use it

Arduino programming guide series

What is a “static” variable and how to use it

The C language contains special keywords that modify the way a variable behaves. One of them is the keyword “static”. The way it works is not obvious, so I have created a series of experiments to help you understand.



```
static int local_variable = 0;
Serial.print("In a_function: ");
Serial.println(local_variable);
local_variable++;
```

A static variable is used to preserve the value of a variable. When you mark a variable as static, its value is preserved in your program until the program ends.

Think of a static variable as if it was global, with a reduced scope. A global variable has global scope, and it is preserved for as long as the program runs. A static variable has local scope, but is also preserved for as long as the program runs.

You would normally use a static variable in cases where you wish to preserve the value of a variable inside a function.

Normal local variables are destroyed when the function completes and recreated/instantiated when the function is called.

Let's use an example to help understand the behavior of a static variable.

Consider this code, "sample A", which you can get from Github:

[tcb-script src="https://gist.github.com/futureshocked/a5a194d73e9b4fffd0471bc06e294b81.js"][/tcb-script]

Upload the sketch to your Arduino and open the serial monitor. The monitor will show this output:

```
In a_function: 0 In a_function: 0 In a_function: 0 In a_function: 0 In a_function: 0 In a_function: 0 In a_function: 0 In a_function: 0
```

The counter remains "0" because each time the *a_fuction* is called, the local variable *local_variable* is re-declared and re-initialized to "0". Even though in line 16, the value of the local variable is incremented by one, the new value is lost.

Now consider this code, "sample B", which you can get from Github:

[tcb-script src="https://gist.github.com/futureshocked/4797583d5253ff86485e743378430024.js"][/tcb-script]

Upload the sketch to your Arduino and open the serial monitor. The monitor will show this output:

In a_function: 0 In loop: 1 In a_function: 1 In loop: 2 In a_function: 2 In loop: 3 In a_function: 3 In loop: 4 In a_function: 4 In loop: 5 In a_function: 5 In loop: 6 In a_function: 6 In loop: 7 In a_function: 7 In loop: 8 In a_function: 8 In loop: 9 In a_function: 9 In loop: 10

In this variation of the sketch, the variable *local_variable* is made global by defining it and initializing it in line 1. Even though this is a global variable, I decided to keep the original name in order to maintain continuity between the examples.

Because the value of the *local_variable* is incremented inside the *a_function*, and not re-initialized (as it did in the original version of the sketch), the value is not lost. It is preserved during the subsequent calls of the *a_function* function.

Now, consider the next variation of the code, "sample C", which you can get from Github:

[tcb-script src="https://gist.github.com/futureshocked/8c05aca012ac1f4d6e1bba8b772c27c5.js"][/tcb-script]

Upload the sketch to your Arduino and open the serial monitor. The monitor will show this output:

In a_function: 0 In loop: 1 In a_function: 1 In loop: 2 In a_function: 2 In loop: 3 In a_function: 3 In loop: 4 In a_function: 4 In loop: 5 In a_function: 5 In loop: 6 In a_function: 6 In loop: 7 In a_function: 7 In loop: 8 In a_function: 8 In loop: 9 In a_function: 9 In loop: 10

The only difference between the sketches of variation B and C is that in C, the variable *local_variable* is marked as static.

The output of the two sketches is identical.

Finally, consider the last variation of the code, “sample D”, which you can get from Github:

[tcb-script src=”https://gist.github.com/futureshocked/e0220d413fc0fb3f8d77f9ef4003da94.js”][/tcb-script]

Upload the sketch to your Arduino and open the serial monitor. The monitor will show this output:

```
In a_function: 0In a_function: 1In a_function: 2In a_function: 3In a_function: 4In a_function: 5In a_function: 6In a_function: 7In a_function: 8In a_function: 9In a_function: 10
```

Isn’t this interesting?

In Sample D, the *local_variable* variable is marked as static, but this time is declared **inside** the *a_function* function. As per the first experiment, you would expect the the serial monitor will show zeros as the variable is re-initialized in line 13.

But it doesn’t.

Because *local_variable* is marked as static, its value is preserved when the function finishes its execution. It is not re-initialized when the same function is called again.

When will the contents of the static variable dissapear? When the program ends, just like it happens for a global variable.

Let's recap the outcome of these experiments.

In sample A, the monitor printout shows that the variable is always 0. We expected that since the variable is defined and initialized inside the function. Every time the function is called in the loop, the variable is re-defined and initialized to 0.

In sample B, the variable local_variable is global. As expected, both the loop and a_function functions have access to it. Once it initialized, it increments by 1 inside a_function, and the same value is printed to the monitor from inside loop and a_function.

In sample C, we marked local_variable as static, and due to its position, it is still globally accessible. The printout we get is identical to the one we got from sample B. Declaring this global variable as static has no effect on the outcome. The compiler can even ignore this modifier.

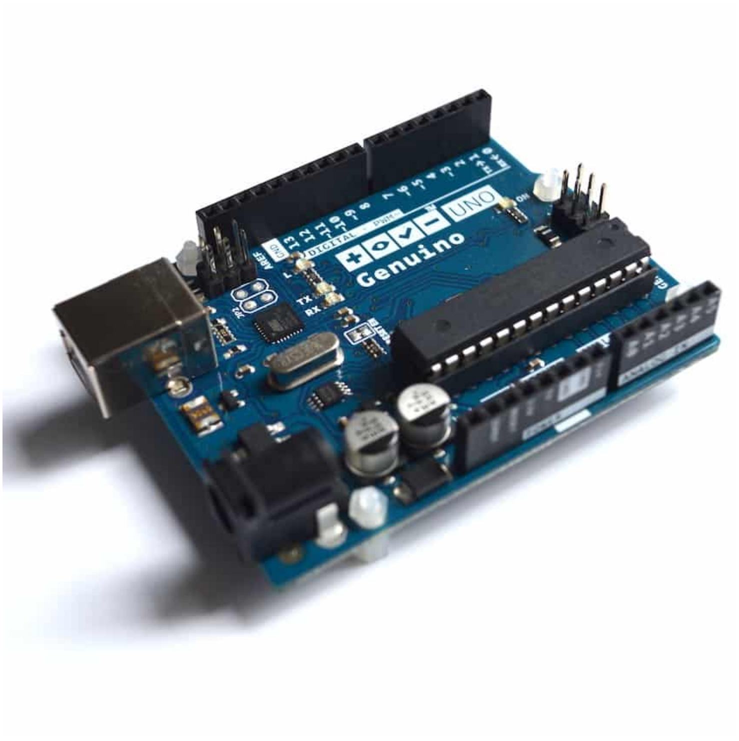
Finally, in sample D, we move the declaration of the variable inside the a_function function. At this point, the sketch is identical to that of sample A. But in sample D, we use the static modifier. The printout from the serial monitor shows that the value of the variable is preserved between calls of a_function.

What is the practical lesson of all this?

1. If you want to preserve the state of a variable across a program, you can create it as a global variable. Marking it as static does not change the fact that the value of the variable will be preserved for the life of the program. However, a global variable can be accessed anywhere in the program, and this could cause defects. In large programs, and because

most Arduino hobbyists don't have access to a debugger, it is best to not use a global variable unless access to it is truly globally required.

2. If you want to preserve the state of a function local variable, mark it as static. You will get all the benefits of the globally declared variable, without the problem described in point 1.



New to the Arduino?

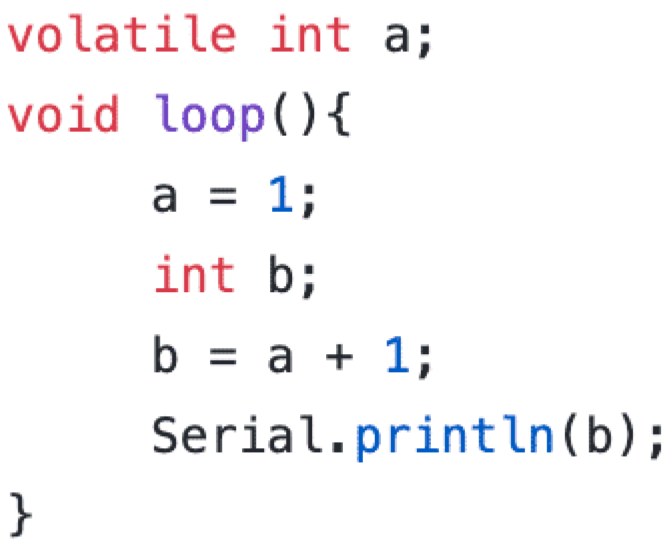
Arduino Step by Step Getting Started is our most popular course for beginners.

22. Understanding the volatile modifier

Arduino programming guide series

Understanding the volatile modifier

Variables can be modified at different parts of a program. Often, it is not possible to predict when a particular variable may be accessed for a read or write operation. This is particularly true when a variable is tied to an interrupt service routine. So, how do we ensure that a variable always contains a correct value? This is where the volatile modifier comes in.



```
volatile int a;
void loop(){
      a = 1;
      int b;
      b = a + 1;
      Serial.println(b);
}
```

A volatile variable is a variable that stores a value that may change at any time from parts of the program that are not related to each other. A volatile variable can be declared in a part of the program, and then be modified out of context.

As such, when the compiler encounters a variable marked as volatile, it does not “optimize” it. It will maintain a single copy of this value in RAM (as opposed to RAM plus CPU a register for faster access).

This way, the programmer sacrifices performance in return for the simplicity of keeping a single copy of a variable, referenced from any part of the program.

Confused?

Let’s drill into this.

Volatile variables are tricky to understand because you first need to have a good understanding of how a CPU works, and how a compiler generates the code that is executed inside the CPU.

In a nutshell, the CPU has a component called “ALU”, short for Arithmetic Logic Unit. It is where arithmetic calculations take place. The CPU knows what to do by looking at an instruction.

An instruction is made up of two things: an instruction code, and the arguments.

For example:

MOV AL, 61h

…is composed of the instruction code “*MOV*” and arguments

“*AL*” and “*61h*”. It results in the value “61” (in hexadecimal) being loaded (moved) into register “AL”. Arguments can be fixed values or memory locations, or other registers.

These instructions are stored in memory. There are three types of memory that a full CPU has direct access to:

- Random Access Memory (RAM).
- Cache.
- Registers.

RAM is plentiful and cheap, though relatively slow.

Registers are few, very expensive, but very, very fast.

And cache is in the middle: less plentiful compared to RAM, more plentiful compared to register. Much faster than RAM, much slower than registers.

The Atmega328p, for example, has SRAM (static RAM, one of several types of RAM), a few registers, but no cache.

Larger CPUs have all three.

A good compiler will try to generate code that is optimized. The optimized code makes efficient use of the available resources, RAM, cache and registers, and fetch-decode-execute cycles.

In applications where there is only one thing happening at a time, like in our Arduino code that does not use interrupts, variable data can be optimized for speed. The compiler will generate code that stores variable data in high-speed registers in addition to the original copy the RAM.

Because the variables are only updated by code in a single thread of execution (such as, in the loop and other functions in the same thread) there is no question as to the validity of a value stored in the register. The value stored in the register is always correct and up-to-date.

On the Arduino, the code can be executed outside the normal path in the form of the interrupt service routine (also known as "interrupt handler").

You can write interrupt routines to execute based on timer events, or external triggers (like a button being pressed).

If the normal path and the interrupt path access the same variable, then it is possible that this variable can be accessed from both places, at any time. This means that our program can never trust that the value it is reading is correct. Perhaps it has been changed by the interrupt routine?

Consider the following example, and imagine that there is an interrupt service routine that operates on the variable at any time (download from Github):

[tcb-script src="https://gist.github.com/futureshocked/e717812bc91959e7edd8c16bff0497b4.js"][/tcb-script]

What is the value of "b" in the serial monitor?

We can't be sure. It can be either "2" or "7".

In the loop, we set "a" to "1". If the interrupt executes before the calculation for "b" happens in the loop, then the value for “a” will be 5 instead of 1. But even if we know that the interrupt has executed is not enough to know what the result of this calculation will be.

We also need to consider that the value for “a” may be stored at different places: RAM, cache (though note the ATMega328p has no cache), or registers.

Which one has the latest correct value? Should we use the copy from the RAM or the copy in the internal register?

I hope this shows that the problem here is uncertainty.

As the designers of a program, we want to remove such uncertainties. We want to write programs that are predictable and deterministic.

The C/C++ language offers the “volatile” modifier for this specific purpose. We can use this modifier to mark a variable as one that can change at any time across different memory accesses, and accross parts of the program that operate at different contexts.

By marking a variable as volatile, we instruct the compiler to **not** optimize for reads and writes for the variable.

The compiler will ensure that the variable value is **always** read from RAM.

Without the volatile modifier, the variable value will be copied from the RAM to a local CPU register the first time it is referenced. After that, it will be read from the CPU local register. Only if this variable copy in the register has been replaced by another variable will the CPU re-read it from the RAM.

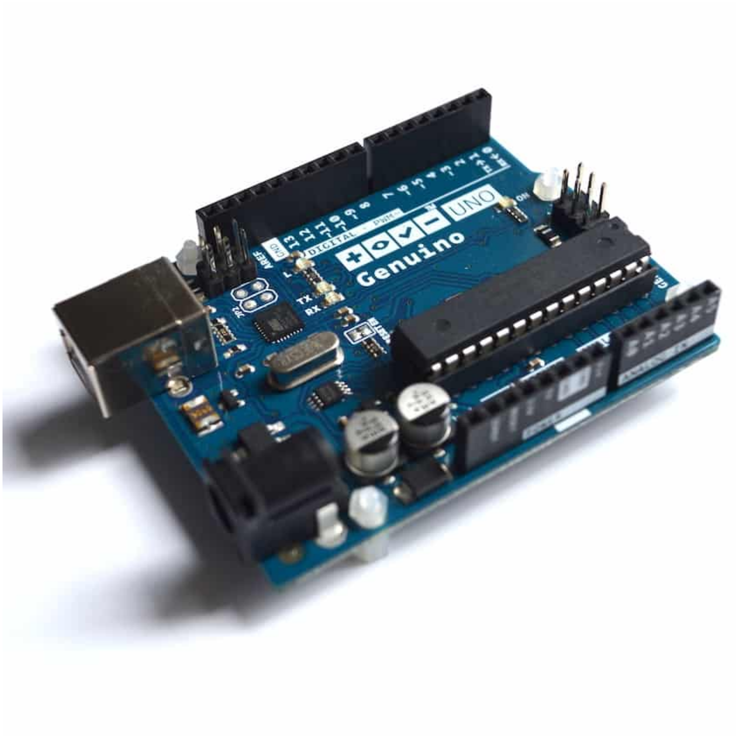
With all this in mind, let's update the example code with volatile (download from Github):

[tcb-script src="https://gist.github.com/futureshocked/53f24f97c3852b77d5cc030a2852810c.js"][/tcb-script]

The only difference between the first and second sketches is that the "a" variable is now marked as "volatile".

That's it.

During execution, the value for this variable will always be fetched from RAM, ensuring that it is current and not stale.

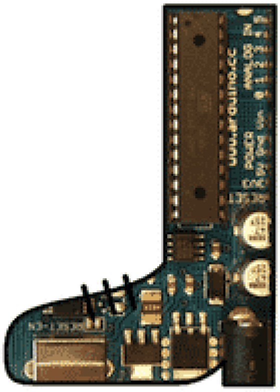


23. Optiboot, a free upgrade for your Arduino

Arduino programming guide series

Optiboot, a free upgrade for your Arduino

Your Arduino's microcontroller contains special software, the bootloader, which makes it very easy to upload new sketches. It is possible to replace the default bootloader with a more efficient variation and hence implement a software upgrade to give your Arduino more space for your sketches and faster uploads.



The Arduino owes much of its ease of use to its bootloader. The bootloader is software that is resident in the Atmega microcontroller and is responsible for making it easy to upload a new sketch from the Arduino IDE to the flash memory of the microcontroller.

The term “resident” means that the bootloader is not overwritten when you upload a new sketch. It remains ready to facilitate the next sketch upload.

Without a bootloader, the process would be somewhat more complicated. You would need to use specialized hardware (such as an ISP programmer) to achieve the same thing. I suspect that many people would give up at that point and never go on to discover the real Arduino magic (getting things done and learning, quickly).

The default bootloader that comes with the Atmega328P on the Arduino is excellent. There are not too many good reasons to change it. But then again, many people don’t need a reason to try something new!

There are alternative bootloaders for the Atmega microcontrollers out there, that promise to improve various aspects of the default bootloader that ships with most Arduinos.

One that delivers a lot of advantages over the default is the Optiboot bootloader.

Advantage 1: size

Optiboot is smaller, by around 1.5KBytes with a total footprint of just 500 bytes.

This means that more flash memory is available for your sketch.

Advantage 2: speed

Optiboot also runs at higher baud rates, which means that it will be faster to upload a new sketch.

This is great for rapid prototyping. A faster sketch upload speeds means that you will be able to iterate through new versions of your sketches faster.

There is a good chance that if you have a newer Arduino Uno, it already has Optiboot installed. If you don’t, you can install it with the help of an AVR ISP programmer.

Although I have searched, I have not been able to find a reliable way to detect the kind of bootloader installed on your Arduino. Jeelabs has a sketch that can detect some older bootloaders but has not been maintained for a while. So, if you want to know what you have, consider uploading your bootloader yourself.

24. A real time OS for the Arduino

Arduino programming guide series

A real-time OS for the Arduino

A real-time operating system is optimized so that processing is completed within tight time bounds, and execution is consistent and predictable. It is the preferred operating system for critical applications. And a version of it works on the Arduino Uno.

In a previous article in this series, I discussed a simple approach for implementing multitasking on the Arduino. This approach is based on the finite state machine paradigm. It is good enough if your program has relatively few states, for example controlling “simultaneously” two or three devices.

However, as an application becomes more complicated, the state machine approach becomes harder to handle and more prone to defects.

When that happens, it is time to turn your attention to something more robust. One such solution is FreeRTOS, or "Free Real-Time Operating System".

FreeRTOS is not a solution specific to the Arduino.

It is software that runs on many microcontrollers and microprocessors. It is a commercially developed product, with an open-source license that allows people to integrate it into their designs, free of charge, for open-source and even commercial closed source projects.

It is recognized as a high-quality, well tested, documented and supported real-time operating system.

Most people are used to general-purpose operating systems, like Windows and Mac OS. These OSs manage the resources of the host computer so that the various processes that are running on top of it get their "fair" share. The OS is responsible for allocating resources (like memory and CPU time). The OS will make a best-effort attempt to allocate resources to a process but will give no guaranty that it will.

A scheduler is a special service within the OS that makes such decisions based on various scheduling algorithms. In general-purpose operating systems, the scheduler tries to be fair to all processes but can make no guarantees.

A real-time OS, on the other hand, is designed so that each process will get the resources it needs in a predictable way.

Critical applications, such as medical, avionics, and industrial automation, depend on predictable patterns in the way that the scheduler allocates resources to a process. Therefore, a general-purpose OS is not suitable, and a real-time OS takes over.

A real-time OS is not meant to work on the type of processors that we find in a personal computer, but usually on microcontrollers used in embedded applications.

Unlike in general-purpose computing, the applications in which a microcontroller is used are very specific: they are only designed to do a particular set of functions, reliably, and predictably. The real-time OS ensures the scheduling, inter-process communication, and synchronization of the processes that make up the firmware running on these microcontrollers.

You can use FreeRTOS on the Arduino, of course!

There is a FreeRTOS library that you must install first (just search for "FreeRTOS" in the Library Manager), and a respectable learning curve to get through. A good place to get started is Maniacbug's Github repository, which contains examples of Arduino sketches that use FreeRTOS, and the FreeRTOS Quick Start Guide page on freertos.org