Introduction

In the past 3 years, I was experimenting with the Arduino platform. It has a lot of features, but a lot of traps. In this tip, I collected a few of my tips & tricks that make the development of Arduino applications more easy.

Speed up I/O Access

On every AVR based Arduino board, the clock speed is 16MHz. Each instruction on the controller completes under 4 cycles, so theoretically, you can do 4 million instructions in one second. However, the I/O speed is much slower. That's because of the `digitalWrite` and `digitalRead` functions. Each time you execute them, a lot of additional code executes also. The additional code is responsible for detecting & mapping the numerical pins to a output port. On every microcontroller, the I/O devices are mapped in groups of 8 pins to ports, which are special registers in the controller.
So the question arises, how slow is it then? In my measurements, the maximal output speed of the following program is around 116-117KHz:

Hide Copy Code

```
void setup()
{
    pinMode(13, OUTPUT);
}

void loop()
{
    digitalWrite(13, LOW);
    digitalWrite(13, HIGH);
}
```

We can improve our program by wrapping the loop code in an infinite loop. It will speed up the execution a litle bit, because the CPU has to do less function calls. The improved loop:

Hide Copy Code

```
void setup()
{
    pinMode(13, OUTPUT);
}

void loop()
{
    while (1)
    {
        digitalWrite(13, LOW);
        digitalWrite(13, HIGH);
        //required if you do serial communication
        if (serialEventRun) serialEventRun();
    }
}
```

In my measurements, the output speed was around 126-127KHz, which is 8.6% faster, but we can improve it. To really top the output speed, we will have to use low level port access. To use

low level poor access, we will have to read the details of the microcontroller used on the board, but it just takes away the beauty of the Arduino platform. So I wrote a library for that purpose, that speeds up I/O access.
It has some limitations. Mainly that it only supports the Uno, Nano, Leonardo, Mega, and Attiny45/85/44/84 models. The code for the library can be found here.

Hide Copy Code

```
void setup()
{
    pinMode(13, OUTPUT);
}

void loop()
{
    while (1)
    {
        //WriteD13 writes Pin 13. D means digital.
        //to write to pin 8 use WriteD8 function.
        WriteD13(HIGH);
        WriteD13(LOW);
        //required if you do serial communication
        if (serialEventRun) serialEventRun();
    }
}
```

It provides inline functions for fast I/O access. The following example produces an output signal around 1Mhz, which is really nice. But, if you put the code in the earlier discussed infinite loop, it gets even faster. With it, you can produce an output signal around< 2,4MHz.

Analog Readings

Analog reading on USB powered Arduino models can produce some weird results, that's because the Analog reference voltage is tied to the logic voltage, which is 5 volts, but if you power your board from a switching power supply or from USB, you can get some serious noise and voltage drop due to the USB cable. So it's posible that your controller is running from 4.5 volts instead of 5 volts.
This is not ideal for precision measurements. To do measurements, you will need to use an external precision voltage reference. Alternatively, you can use the chip's built in voltage reference to measure the current VCC and do some compensated calculations with that.

Hide Copy Code

```
int GetVccMiliVolts()
{
    #if defined(ARDUINO_ARCH_AVR)
    const long int scaleConst = 1156.300 * 1000;
    // Read 1.1V reference against Avcc
    #if defined(__AVR_ATmega32U4__) || defined(__AVR_ATmega1280__) ||
defined(__AVR_ATmega2560__)
        ADMUX = _BV(REFS0) | _BV(MUX4) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
    #elif defined (__AVR_ATtiny24__) || defined(__AVR_ATtiny44__) ||
defined(__AVR_ATtiny84__)
        ADMUX = _BV(MUX5) | _BV(MUX0);
    #elif defined (__AVR_ATtiny25__) || defined(__AVR_ATtiny45__) ||
defined(__AVR_ATtiny85__)
```

```
        ADMUX = _BV(MUX3) | _BV(MUX2);
    #else
        ADMUX = _BV(REFS0) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
    #endif
    delay(2); // Wait for Vref to settle
    ADCSRA |= _BV(ADSC); // Start conversion
    while(bit_is_set(ADCSRA,ADSC)); // measuring
    uint8_t low = ADCL; // must read ADCL first - it then locks ADCH
    uint8_t high = ADCH; // unlocks both
    long int result = (high<<8) | low;
    result = scaleConst / result;
    // Calculate Vcc (in mV); 1125300 = 1.1*1023*1000
    return(int)result; // Vcc in millivolts
    #else
        return -1;
    #endif

}
```

Get Available Free RAM

The following function will return the currently free RAM memory in bytes.

Hide Copy Code

```
unsigned int UnusedRAM()
{
    unsigned int byteCounter = 0;
    byte *byteArray;
    while ( (byteArray = (byte*) malloc (byteCounter * sizeof(byte))) != NULL )
    {
        byteCounter++;
        free(byteArray);
    }
    free(byteArray);
    return byteCounter;
}
```

Use const, PROGMEM & Global Variables

If you have to store large amounts of data in your code (like a bitmap or fornt, or whatever) make the variable `global const` & include the `PROGMEM` statement. This directs the compiler to store the data only in the program memory instead of the RAM.

Hide Copy Code

```
//5120 byte array. Normally it woudn't fit into the RAM,
//but with static storage it gets stored in the program memory
const byte big_array[5120] PROGMEM = {0};
```

Forget the Default IDE

The Arduino IDE is great, but it's not intended for professional software development. So after a time, it becomes a pain to use it, especially if you are writing a program more than a few hundred lines.
Alternatively, you can use Atmel Studio or Visual studio with Visual Micro plugin. The Visual Micro plugin features a debugger, but it is not free. For the debug feature, you will have to pay $17 dollars. Visual Micro can be downloaded from here.
You also can use your favorite text editor to write your programs. Since Arduino 1.5 the Arduino IDE supports building and uploading your program through command line.
The command line syntax is the following:

Hide Copy Code

```
arduino --board (board type description) --port (serial port) --upload (sketch path)

arduino --board (board type description) --port (serial port) --verify (sketch path)
```

The board type description is the following:

Hide Copy Code

```
package:arch:board[:parameters]
```

You can find more information about command line usage in the Arduino IDE manual document, which is located here.

Additional Device Support & Useful Libraries

I've compiled a package of my daily used libraries and device support files. I call this package Arduino Extensions. It can be downloaded from here. Suggestions, pactches & pull requests are appreciated. :)

Inline Your Small Functions

Inline functions are a great way to speed up your program, especially, if you call them in the loop function. By default, the `inline` keyword only suggests the function for inlining. The compiler may decide that it's not worth inlining in the optimization phase.
The GCC manual says that an inline function is as fast as a Macro. As always, there's a catch. If you make a function `inline`, it will make your program code size larger, if you call that function in several places.
To force a function inline, use the `__inline` keyword

Hide Copy Code

```
__inline void SomeFunction()
{
    Serial.println("I'm an inline function");
}
```