EENG 496 (462?)
Embedded Real-Time Systems

Module 1:
Course Overview and Introduction

Introduction

- **What this class covers:**
  - Embedded systems in general
  - A review of some low-level circuits, sensors, and actuators that embedded processors often interface to
  - Intro to feedback control
  - Intro to priority-driven multitasking, real-time scheduling
- **What this class is NOT about:**
  - Operating Systems beyond programmer-controlled scheduling
  - Assembly language programming
  - Networking and IOT (sorry)
- **This class assumes:**
  - C Programming
  - Basic knowledge of Analog and Digital Circuits
  - Calculus
  - Fourier analysis may prove to be helpful

Course Outline

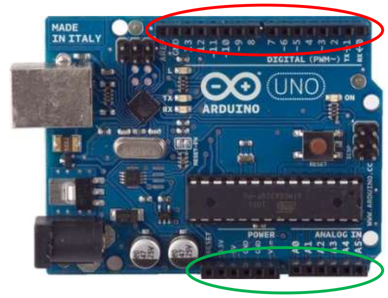
- **Topics (see syllabus for approx schedule):**
  - Introduction to Embedded Systems
  - Interface Electronics
  - Digital to Analog and Analog to Digital Conversion
  - Multithreading Kernels
  - Thread Synchronization, Semaphores, and Interrupt Service Routines
  - Feedback Control and the PID algorithm
  - Real-Time Scheduling (Rate/Deadline Monotonic)
- **Lab and Homework Rules and Timing**
  - see syllabus

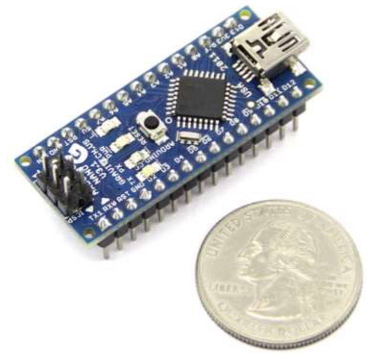
Introduction

- **Text**
  - None required
  - Detailed lecture notes will be provided on the course Canvas site
- **Compilers and Operating Systems**
  - We will be making extensive use of the Arduino development environment, which is free
  - Available under Windows, Mac OS X, and Linux
  - Instructor will be using the Windows version
  - Arduino devices are programmed using mostly a C subset of C++
  - C++ is used for library development
  - The main() routine is provided for you (more on that later)
- **Other**
  - Your course fee pays for a parts kit that you keep

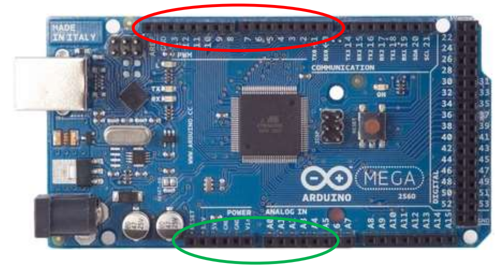
Arduino

- **Open-source development**
- **Uno / Nano**
  - Flash: 32 kbyte
  - SRAM: **2 kbyte**
  - 8-bit ATmega328, 16 MHz

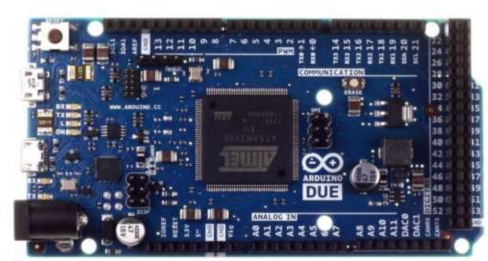




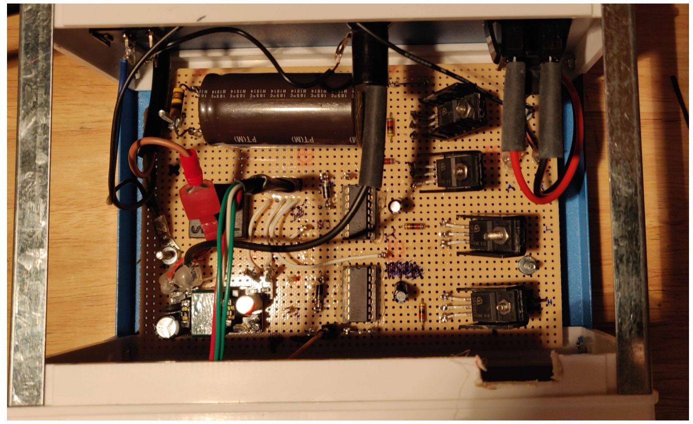
- **Mega 2560**
  - Flash: 256 kbyte
  - SRAM: **8 kbyte**
  - 8-bit ATmega2560, 16 MHz

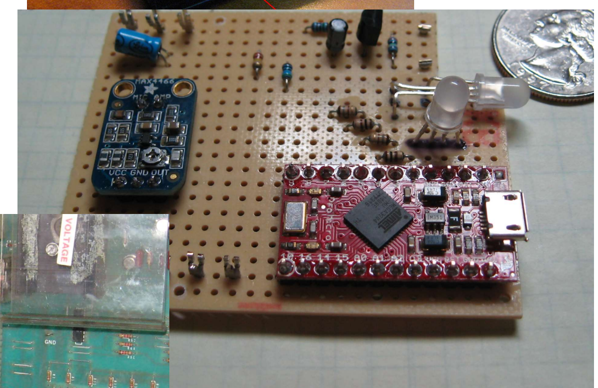


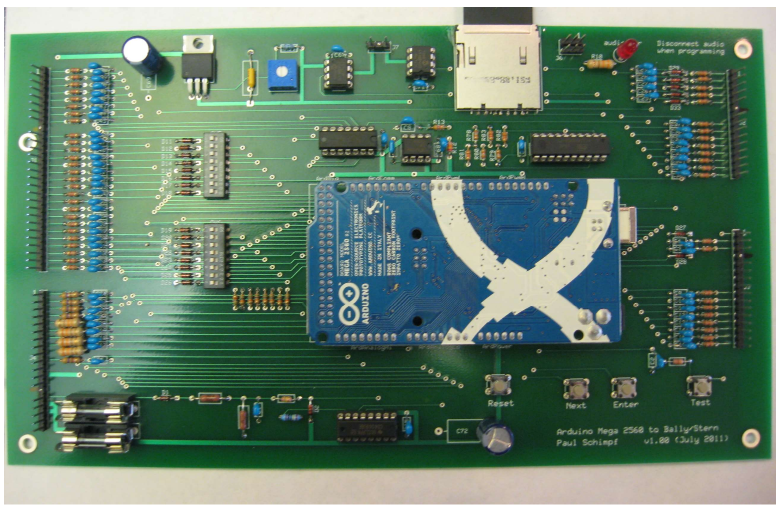
- **Due**
  - Flash: 512k kbyte
  - SRAM: **96 kbyte**
  - 32-bit ARM Cortex-M3, **84 MHz, 3.3V I/O**
  - x 1.24 DMIPS/MHZ = 104 DMIPS (equivalent to first gen Raspberry Pi)

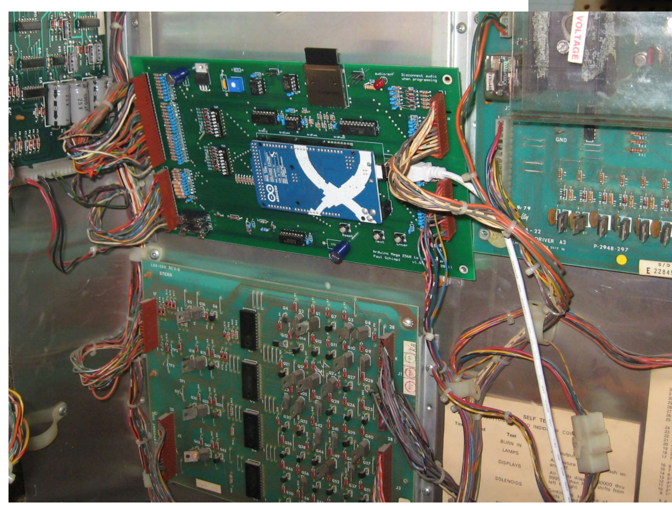


Not Just for Tinkering









- **AVR microcontrollers use a modified Harvard (separate instruction and data buses) RISC architecture**

More About Arduino

- **Arduino is based on a microcontroller, as opposed to a microprocessor**
- **Advantages**
  - **built-in analog input on some pins (ADC)**
  - **built-in (hardware) for "analog" output on some pins**
  - **"analog" output is actually pulse-width-modulation (PWM) with the timing done in hardware (as opposed to software PWM on a microprocessor)**
  - **built-in <u>*hardware*</u> timers, which are handy for many things, such as PWM and generating periodic interrupts**
  - **built-in Flash (persistent program) and SRAM (data) and EEPROM (configuration data) memory**
- **Disadvantages**
  - **small SRAM space (for variables) if not augmented**
  - **towards the lower end of processing power**

ESP8266, ESP32

- Similar to Arduino in that they provide Digital, Serial, SPI, and I2C input/output
- Most importantly, they add built-in WiFi communication capability (802.11 b/g/n)
- ESP32 adds Bluetooth, a second CPU core, additional GPIO, faster clock, more SRAM, hardware PWM, touch and temp sensors
- Supported by the Arduino IDE
- 32-bit processor, 3.3V
- No hardware-based PWM (analog output)
- ESP8266 has only one analog input
- Good choice for IOT devices

ESP8266, ESP32

Specifications	ESP8266	ESP32
MCU	Xtensa® Single-Core 32-bit L106	Xtensa® Dual-Core 32-bit LX6 600 DMIPS
802.11 b/g/n Wi-Fi	Yes, HT20	Yes, HT40
Bluetooth	None	Bluetooth 4.2 and below
Typical Frequency	80 MHz	160 MHz
SRAM	160 kBytes	512 kBytes
Flash	SPI Flash , up to 16 MBytes	SPI Flash , up to 16 MBytes
GPIO	17	36
Hardware / Software PWM	None / 8 Channels	1 / 16 Channels
SPI / I2C / I2S / UART	2/1/2/2	4/2/2/2
ADC	10-bit	12-bit
CAN	None	1
Ethernet MAC Interface	None	1
Touch Sensor	None	Yes
Temperature Sensor	None	Yes
Working Temperature	- 40℃ ~ 125℃	- 40℃ ~ 125℃

Raspberry Pi

- Processor speed is significantly higher than 8-bit Arduinos but comparable to 32-bit Arduino (Due)
- Microprocessor (vs. Microcontroller)
  - no *hardware-based* timers, no hardware-based PWM, no built-in ADC
- External vs. on-board RAM
  - much more memory, but it is DRAM instead of SRAM (DRAM is slower)
- OS vs Bare (can be good or bad)
  - runs Linux, whereas Arduino runs no OS
  - Arduino expansion is done without device drivers, strictly via C++ libraries
- Dev Environment is on-board
  - whereas Arduino uses a cross-compiler dev environment
  - i.e., you log into a Pi, and download to an Arduino
- Default Dev Language is Python
  - which is interpreted rather than compiled, and is (still) quite slow
  - offers impressive optimized Math libraries
  - C/C++/Java can be used on the Pi, which speeds things up …

Some Speed Comparisons

Sieve of Eratosthenes

Primes from 2 to 7800, 600 passes, time in seconds

Machine/Language	C99 -O3	C99	Java	Python 2.7	Python 3.7
Desktop (9th gen Core i5)	0.004	0.024	0.013		0.48
Laptop (4th gen Core i7)	0.004	0.032	0.014	1.3	0.51
Raspberry Pi 4 (1.5 GHz ARM Cortex-A72)	0.14	0.14	0.035		1.3
Raspberry Pi 1 (700 MHz ARM11)	0.36	0.36	N/A		30
Arduino Due (84 Mhz ARM Cortex-M3)		1.4			
Arduino Mega2560 (16 Mhz ATmega2560)		7.5			

Observations ***(for this problem)***

Arduino Mega (and Uno) are much faster than the Pi 1 running Python
but over an order of magnitude slower than the Pi 1 running C

Arduino Due is comparable to the Pi 4 running Python
and over an order of magnitude faster than the Pi 1 running Python
but an order of magnitude slower than the Pi 4 running C
and roughly half an order slower than the Pi 1 running C

Unoptimized C is an order of magnitude faster than Python 3 on the Pi and on Intel

Optimized C is two orders of magnitude faster than Python 3 on Intel
but not on the Pi, where the C optimizer does nothing (ARM compiler)

Java is faster than unoptimized C, but slower than optimized C (this has been observed elsewhere)

Which should I use?

- Lots of advice out there, here's my take ...
  - If you need a lot of memory (presently >96 kbytes) then you want a Raspberry Pi
  - If you need it to be tiny, you probably want an Arduino
  - If you need tiny with Wifi, you probably want an ESP
  - If you need specialized I/O (e.g. analog), you'll want to consider an Arduino
  - If you need OS services, you'll want a Raspberry Pi
  - If you need sophisticated Math libraries you'll want a Pi
  - Needing speed might mean a Raspberry Pi, but not necessarily (and you may want to avoid Python)
  - Multi-tasking (with or without threads) can be done on either, but on Arduino that will be a custom library, whereas you can use standard Linux pthreads on the Pi
  - If you need hard real-time guarantees, Pi may be OK, but you don’t have complete control and Linux can get in the way

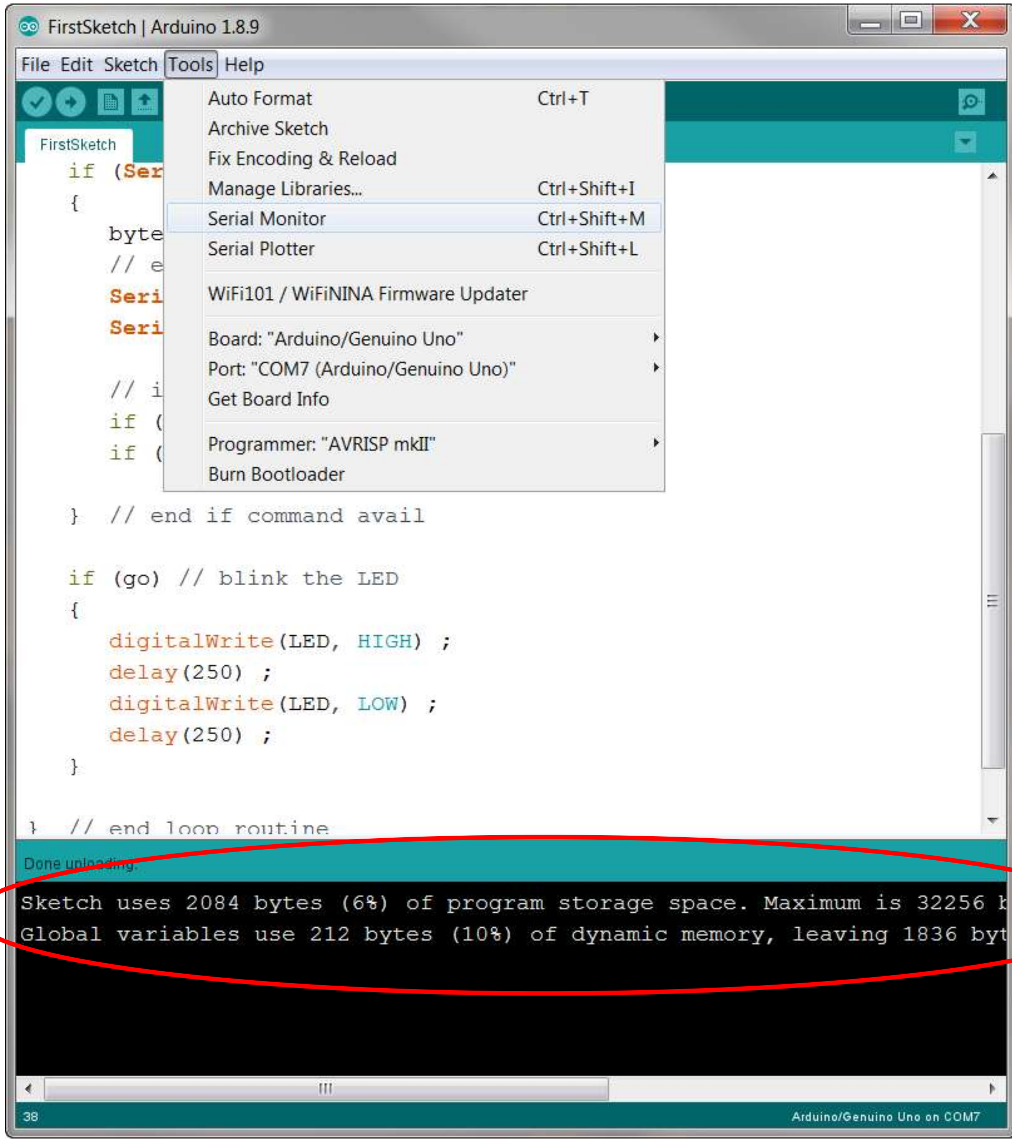
Getting Started w/ Arduino

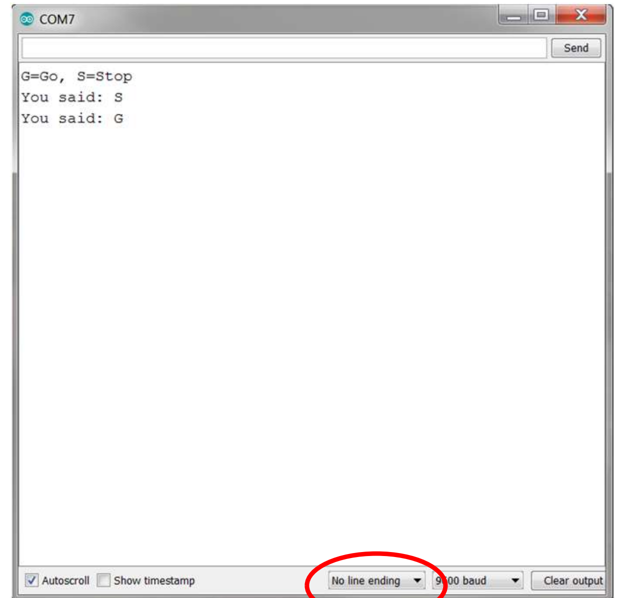
- try this "sketch":

```
// why might you prefer this over
// const int LED=3 ; ??
#define LED 13

void setup() {
   pinMode(LED, OUTPUT) ;
   Serial.begin(9600) ;
   Serial.println("G=Go, S=Stop") ;
}

void loop() {
   char bytein ;

   // is static necessary here?
   // what is the alternative?
   static boolean go=true ;

   // check for char command
   if (Serial.available()>0) {
      bytein = Serial.read() ;
```

```
      // echo what you got:
      Serial.print("You said: ");
      Serial.println(bytein) ;

      // if valid cmd, do it
      if (bytein=='S') go=false ;
      if (bytein=='G') go=true ;

   }  // end if command avail

   if (go) {
      // blink the LED, which is
      // connected to pin 13
      digitalWrite(LED, HIGH) ;
      delay(250) ;
      digitalWrite(LED, LOW) ;
      delay(250) ;
   }

}  // end loop routine
```

Getting Started w/ Arduino





- **Open the serial monitor**
  - **set "No line ending"**
- **Type S <Enter>**
  - **the command should be echoed back to you**
  - **the LED should stop flashing**
- **Type G <Enter>**
  - **the LED should start flashing**

Code Notes

- **The Arduino programming language is C**

  (skinny C++ would be more precise)

- **The main() function is written for you**
  - ***logically*, it works like this ...**

```
void main() {
    setup()
    while(1) {
        loop() ;
    }
}
```

  - **setup() is called when the code initializes, which is after a download or a hardware or software reset (opening the serial monitor performs a software reset)**
  - **then loop() is called, in an infinite loop, as fast as possible, which depends on the processor speed and how much code you put into it**

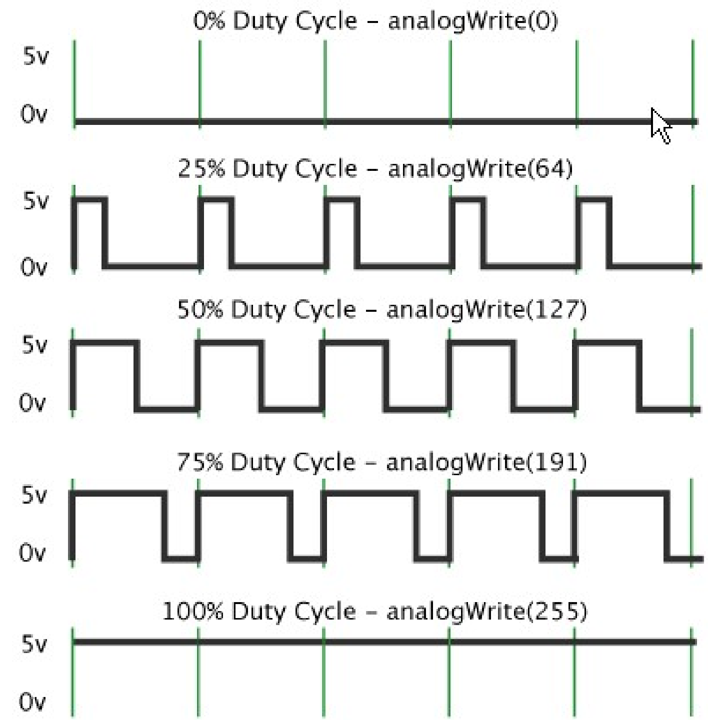
Code Notes

```
void loop() {
   ...
   if (go) // blink the LED
   {
      digitalWrite(LED, HIGH) ;
      delay(250) ;
      digitalWrite(LED, LOW) ;
      delay(250) ;
   }

}  // end loop routine
```

- **This loop() includes a half second (500 msec) of hard-coded delay in order to control the LED blink rate**
- **During the delay() calls, no other code can run, except code in interrupt service routines**
- **We could use a hardware-based timer to blink the LED instead, *with or without* code (but let's not get ahead of ourselves)**

analogWrite == PWM

- **Many μControllers produce "analog" outputs in the form of a Pulse-Width Modulated bitstream**
  - instead of an analog voltage, this is a single-bit square wave where the duty-cycle represents the analog value
  - such signals are easy to generate using timer / counter circuits
  - this is how analog outputs work on the Atmega (Arduino)
  - surprisingly, such outputs are often directly usable in this form, for example:
  - motor speed control
  - sound output (a class D amplifier outputs PWM)

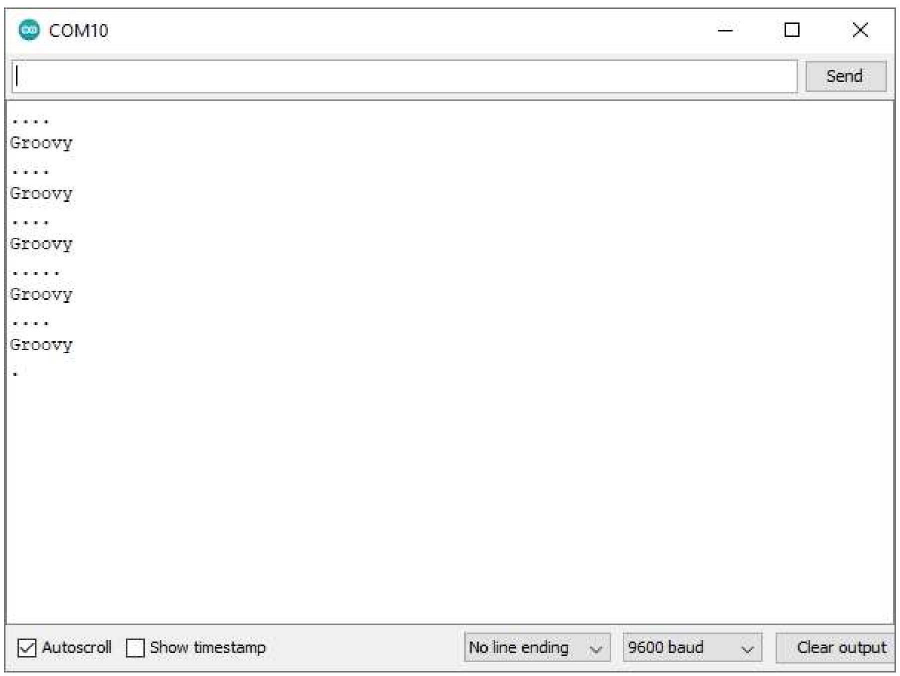


Built-in Timers

- The Uno has 3 hardware timers
  - the Mega has 6
- These can be used for various things, such as
  - automatically changing output pin state
  - firing interrupts
  - Timer0 is used by the delay(), millis(), and micros() functions
  - analogWrite() uses various Timers depending on the pin being written to (including Timer0 !!!)
  - the Tone generation and Servo libraries use Timers
  - Timers can be controlled directly via processor registers that are made available to the C programming level as reserved variable names
  - there are also libraries that make it much easier (e.g., the TimerOne, TimerThree, TimerFour libraries)

Timer Interrupt Driven Blink

- **Allows us to do something else simultaneously**
  - is the same as being in parallel?
  - we'll have much more to say about interrupts later on
- **Note the static variable again**




```
#include <TimerOne.h>
#define LED 13

void setup() {
  Serial.begin(9600) ;
  pinMode(LED, OUTPUT) ;
  Timer1.initialize(500000) ;
  Timer1.attachInterrupt(isr) ;
}

void loop() {
   Serial.println("\nGroovy") ;
   delay(2200) ;
}

void isr() {
  static boolean state=false ;
  state = !state;
  digitalWrite(LED, state) ;
  Serial.print(".") ;
}
```

Output pins can also be read

- In this case providing a clever way to provide a boolean "state"
  - as opposed to using a global or static variable (as in the preceding example)
  - some may frown upon this practice

```
#include <TimerOne.h>
#define LED 13

void setup() {
  Serial.begin(9600) ;
  pinMode(LED, OUTPUT) ;
  Timer1.initialize(500000) ;
  Timer1.attachInterrupt(isr) ;
}

void loop() {
   Serial.println("\nGroovy") ;
   delay(2200) ;
}

void isr() {
  digitalWrite(LED, !digitalRead(LED)) ;
  Serial.print(".") ;
}
```

And Finally

- **We can let a Timer manipulate the pin directly**
  - **no blink code required at all!**
- **This is not the only way to get PWM output**
  - **analogWrite will do that as well**
  - **(more on that later)**

```
#include <TimerThree.h>

void setup() {
  Serial.begin(9600) ;
  // 1 sec period
  Timer3.initialize(1000000) ;
  // 50% duty cycle on pin 5
  Timer3.pwm(5, 512) ;
}

void loop() {
   Serial.println("\nGroovy") ;
   delay(2200) ;
}
```