Beyond the Arduino:

Programming AVR Microcontrollers in C

Elliot Williams

elliot@littlehacks.org
github: hexagon5un

March 18, 2014

Outline

Outline

AVR C for Arduinisti

The Familiar

- Control external devices by connecting them to AVR pins
- Communicate with your desktop/laptop over serial
- PWM (alias `analogWrite()`),
  ADC (alias `analogRead()`),
  initialization / event loop structure

The New

- Different toolchain to program chip
- Learn a few microcontroller-C programming idioms
- Learn about, configure, eventually master the internal hardware peripherals

C?

Isn't C a Dinosaur?

- Yeah.

So why learn microcontroller-style C then?

- Speed: your code can run *much* faster
- Flexibility: make the chip do what you want/need
- Responsiveness: do many things at once, respond instantly
- Portability: C is available for every(?) CPU
- Portability II: your code will work on all AVRs
- Curiosity: just to learn more about microcontrollers

Outline

What is a Microcontroller?

It's a whole computer on a chip:

- Write programs in various languages (C, assembly, BASIC)
- CPU (1-20MHz)
- Dynamic memory (SRAM)
- Non-volatile memory (Flash ROM and EEPROM)

But it's a *very* little computer:

- 8-bit words
- Not much memory
  (1-32 KB program space, couple KB SRAM)
- No operating system
- Low-level input/output
- = halfway between a "component" and a "computer"

What can it do?

Cool Stuff

- Super-fancy Blinkers, POV toys
- Robots / Quadcopters / 3D Printers
- Dataloggers (GPS, Energy monitors)
- USB Devices
- Interface between real world and computer world
  – glue logic in many devices
- But how??

It's All in the Pins

Output

- Digital Output: Apply 0V or 5V to any pin.
  – Light up LEDs, flip switches, spin motors
  – Digital communication: UART, SPI, I2C, USB
- Pulse-width Modulation (PWM) for fake analog:
  Arduino's `analogWrite()`
  Flip the digital output on and off quickly.
  Percentage of time high/low determines average voltage

It's All in the Pins

Input

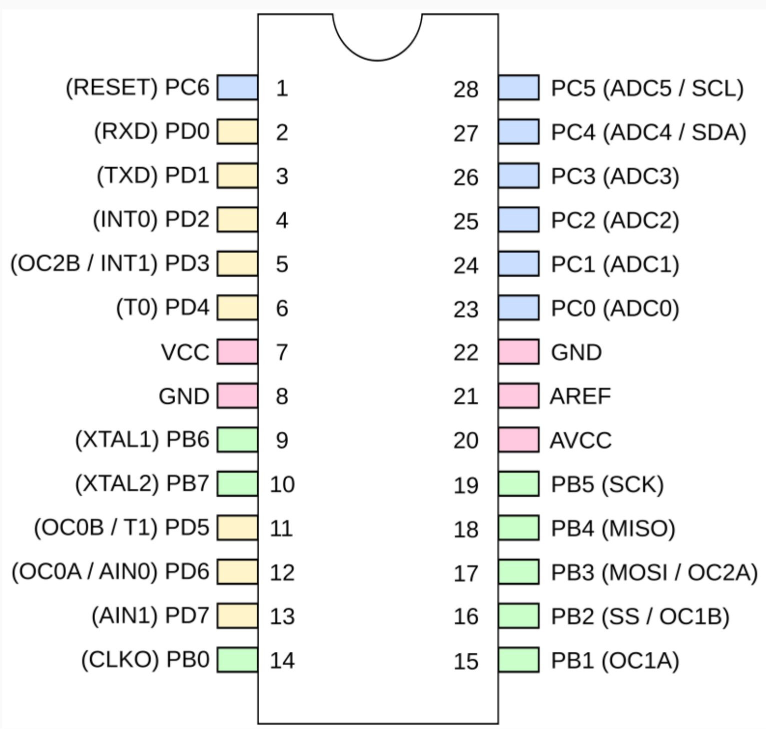
- Input: Read voltage levels applied to pins
  – Digital (pushbuttons, threshold sensors)
  and of course receiving digital communications
  – Analog-to-Digital Conversion (light levels, audio waveforms)
- Input Modes:
  – "Hi-Z": effectively disconnected from the circuit
  – Internal pullup resistor: test if anything connected

On-board Hardware Peripherals

Making Your Life Easier

- Most everything you can do with an AVR *can* be done just by toggling pins
  (ADC is the big exception)
- But writing code to do send and receive serial data is miserable, and would tie up the CPU
- Robot: monitor the serial line, control (via PWM) motor speed, blink some LEDs, and be continually ready to turn off its killer laser when it detects a person.
- Doing multiple things at once – offload tasks to onboard hardware peripherals.
- Learn the hardware.

The ATMega xx8



The Arduino Rant

Short Version

- The Arduino is an nice development kit
- It has everything you need
- It has more than you need sometimes
- The Wiring/Arduino "language" is very simple to learn
- Too simple. You gloss over a lot of the good stuff.
- Ash: "Working with Arduino is like knitting in boxing gloves."
- Forty years of microcontroller development has been aimed at making it easier for you to realize your projects

Outline

The Basic Workflow

The Lifecycle of AVR Programming

- Write code in C (using whatever you want)
- Cross-compile for the chip $\rightarrow$
  the AVR machine-code version of your code
- Transfer the code to the chip:
  – Hardware programmer to talk to the chip
  – Software to run the programmer
- Get feedback and debug until it works

Getting Firmware into the Microcontroller

What you'll Need to Download or Buy

- Cross-compiler: GNU `avr-gcc` and associated software tools
- Hardware programmer (or a previously installed bootloader)
- `AVRDUDE`: knows how to run many hardware programmers
- Usually a `Makefile` to compile and flash for you in one step
- USB-serial converter: "printf" debugging feedback

Software

Packages

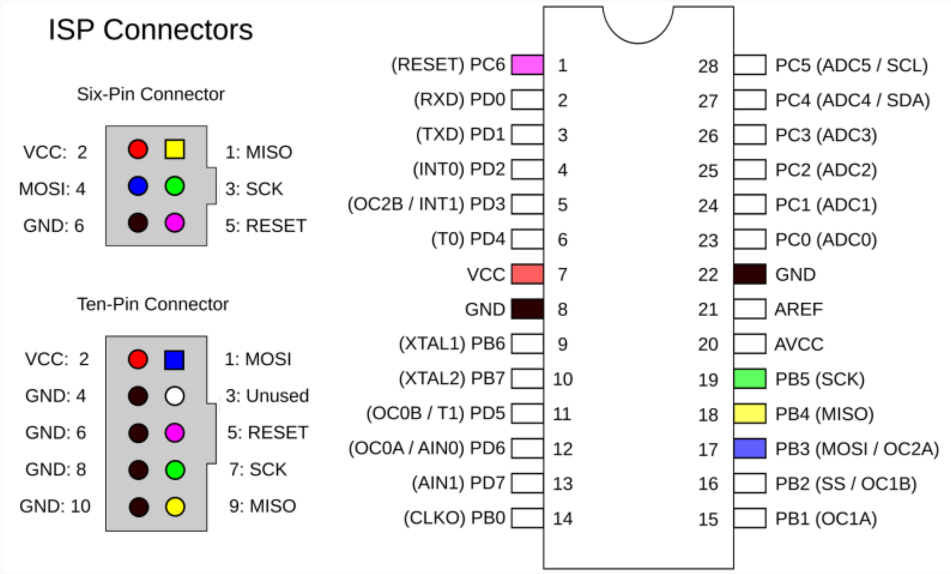
- Linux: `sudo apt-get install avrdude binutils-avr avr-libc gcc-avr`
- Windows: WinAVR, or Atmel's AVR Studio
- Mac: CrossPack (optionally XCode)

Programming Hookup

On Breadboard:

- AVR uses the SPI interface for In-System Programming (ISP) (Yeah. SPI for ISP. Thanks for the confusing acronyms.)
- Bottom line is that you need to hook up four signal wires, plus power and ground.
- `SCK` (serial clock)
  `MISO` (master-in, slave-out)
  `MOSI` (master-out, slave-in)
  `RESET` (tell the AVR to enter programming mode)
- `VCC`, `GND`

Programming Hookup

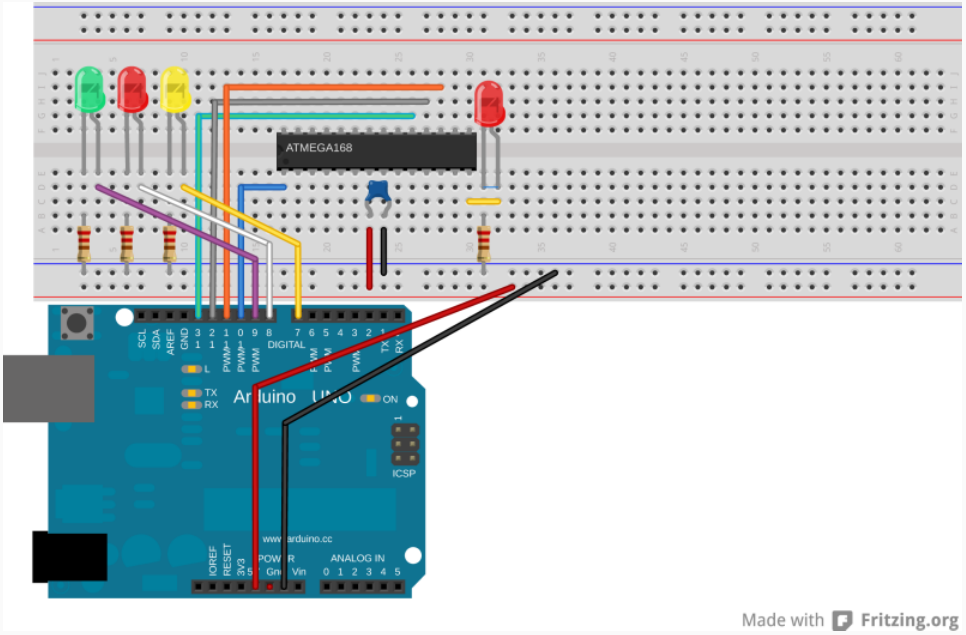


Arduino as Hardware Programmer

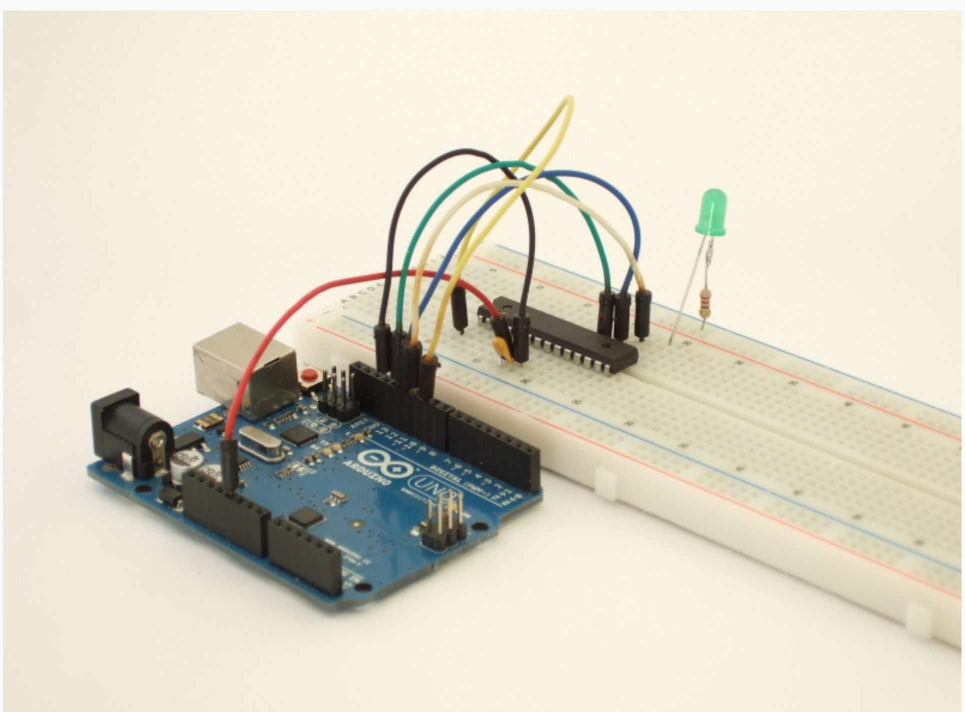
AVRs Programming AVRs

- In the `Files...Examples` menu, you'll find `Arduino ISP`
- Flash that in.
- Now you're ready to talk to the AVR *through* the Arduino
- `avrdude -p atmega168 -c avrisp -b 19200 -P /dev/ttyACM0 -nv`
- If that works, you'll see a lot of details about the chip on your breadboard
- If it fails, re-check connections

Arduino ISP Hookup



Arduino ISP Hookup



Other Programmers

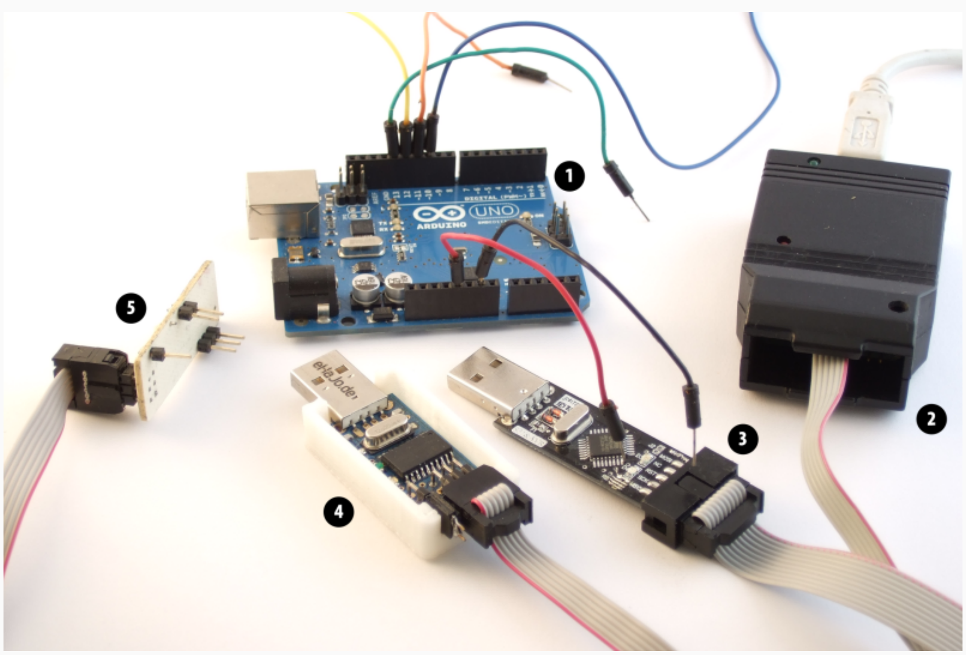
DIY

- Once you've got a working Arduino AVR programmer, you can make your own permanent ISP.
- Search "VUSBTiny" for a truly minimal design
- Also see USBTiny and USBasp projects (DIY Versions.)
- Parallel port connector and 5 wires (DAPA)

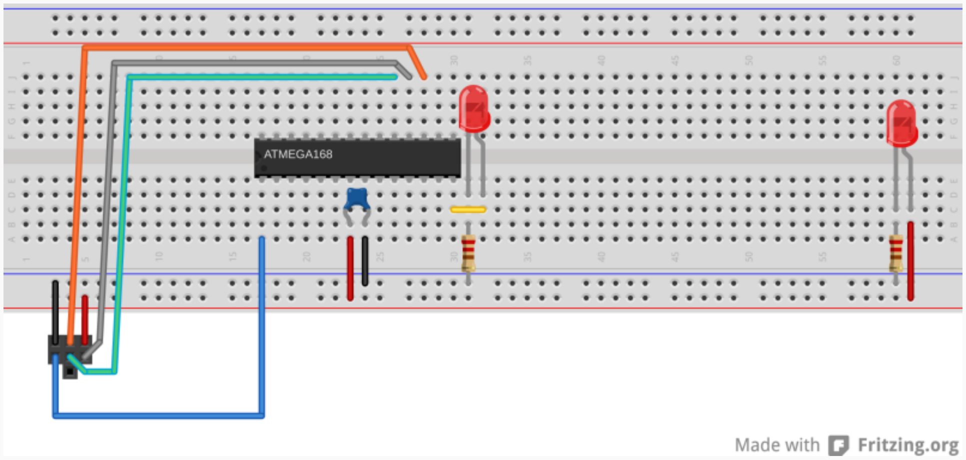
Or Go Pro

- AVRISP MkII (Atmel's Own. Very robust.)
- LadyAda, Sparkfun, etc sell USBTiny kits
- Bus Pirate (SPI mode)
- USBasp-based designs available for $5 from the far east warning: some of these are electrically fragile

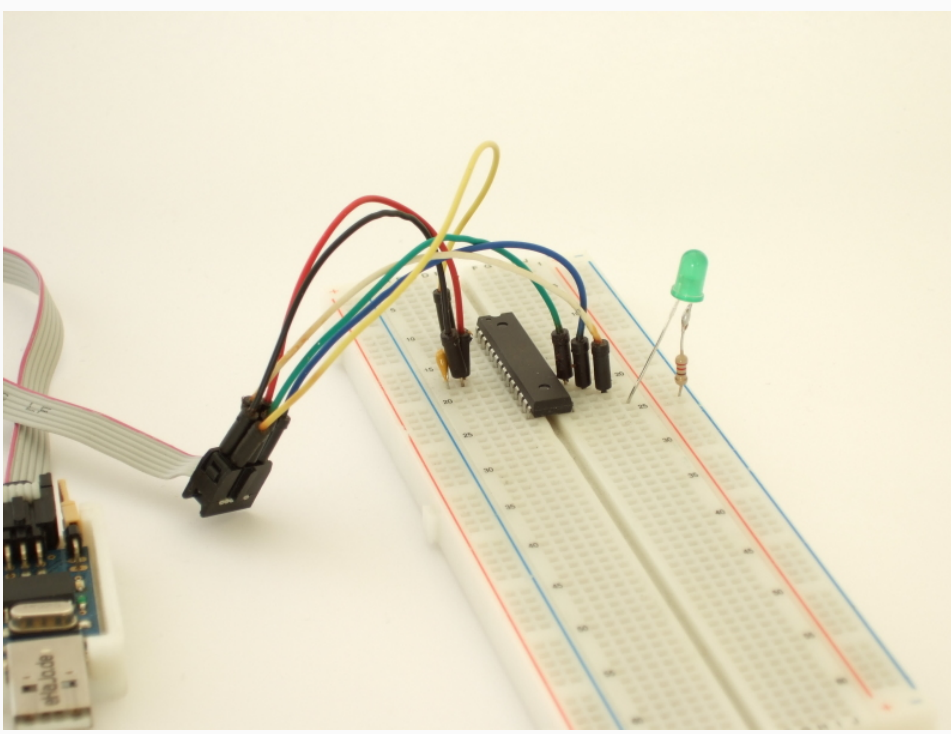
ISP Options



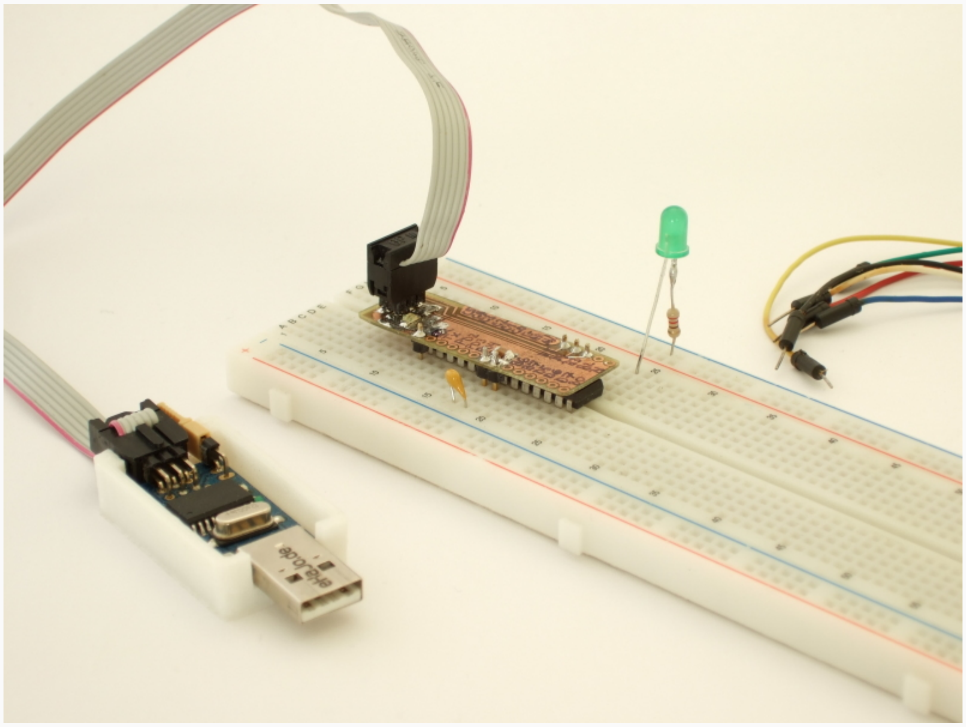
Simplest ISP Hookup



Simplest ISP Hookup



Something a Little More Refined

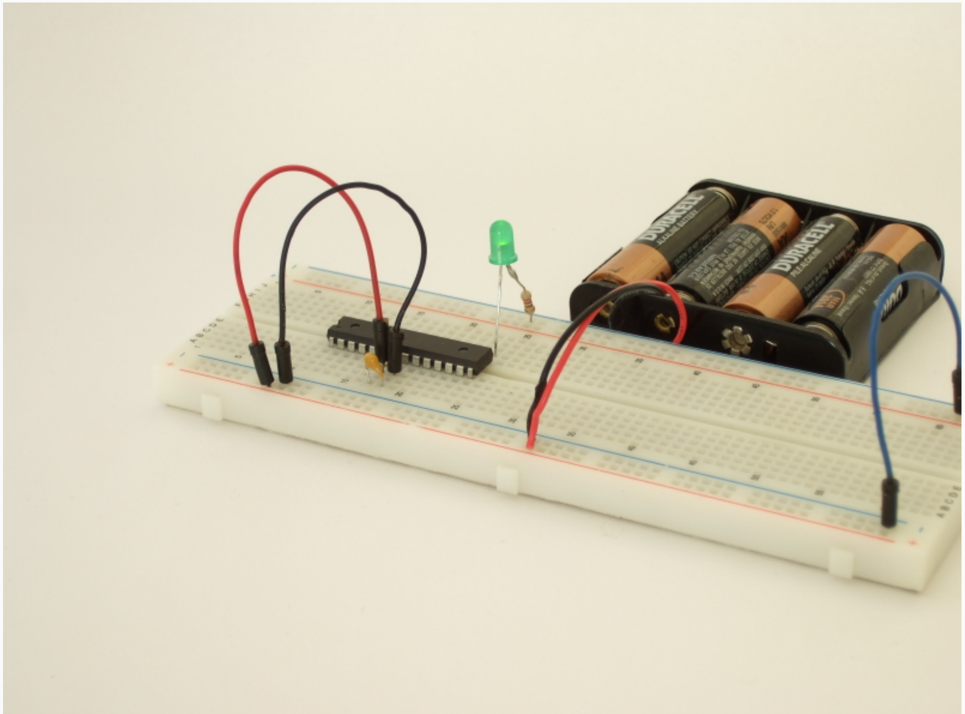


Blinky LED Demo

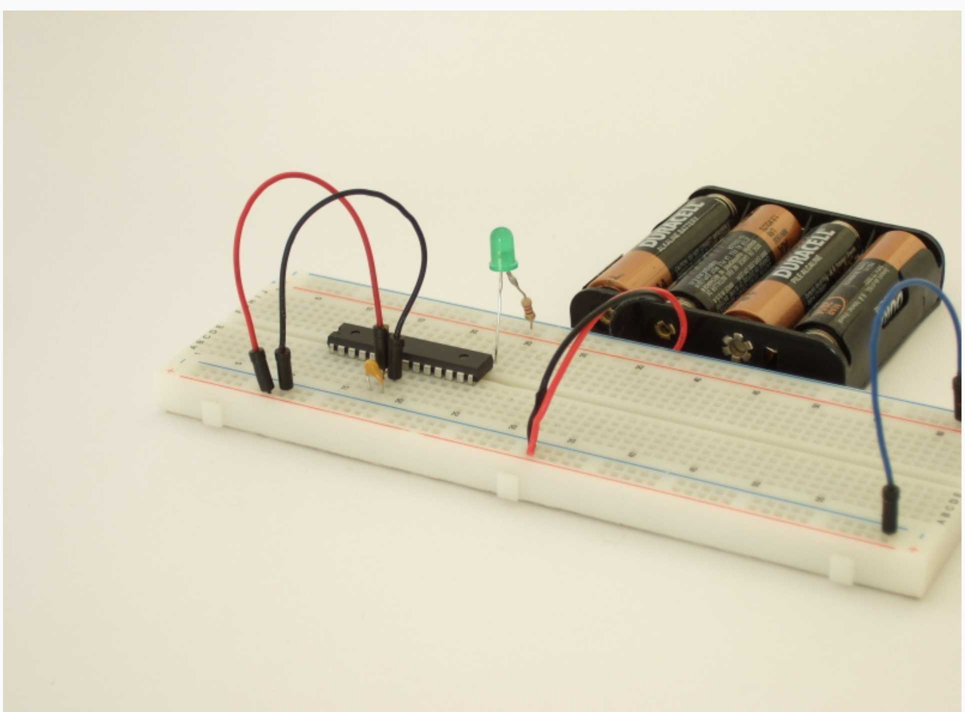
... and Flash

- First thing to do is tweak your Makefile so that it matches your setup:
  – target chip (MCU)
  – programmer
  – serial port and speed if necessary (programmer options)
- Make sure you're opening up the Makefile in the same directory as your project
- Open up a terminal window in the project directory
- Run `make flash` or `make program` and you're off to the races

LED On



LED Off



Flashing the Chip Manually

Just to be sure we know what's going on:

- Type `make` to create the AVR machine-code file
  check for errors, heed warnings
  your code needs to compile successfully first before you upload
- `avrdude -p atmega168 -c usbtiny -U blinkLED.hex`
- `avrdude -p atmega168 -c avrisp -b 19200 -P /dev/ttyACM0 -U blinkLED.hex`

AVRDUDE Options

What you need to know

- `-p chip`: What chip type are you trying to program?
- `-c programmer`: What programmer are you using?
- `-U hexfile`: Which file to upload?
- And some optional options:
- It's good to be able to test these out by hand
  you'll want to personalize these values in your Makefile

blinkLED.c

```
                                                       /* Blinker Demo */

// ------- Preamble -------- //
#include <avr/io.h>                        /* Defines pins, ports, etc */
#include <util/delay.h>                     /* Functions to waste time */

int main(void) {

  // -------- Inits --------- //
  DDRB = 0b00000010;           /* Data Direction Register B:
                                   writing a one to the bit
                                   enables output. */
  // ------ Event loop ------ //
  while (1) {

    PORTB = 0b00000010;            /* Turn on one LED bit/pin in PORTB */
    _delay_ms(1000);                                           /* wait */

    PORTB = 0b00000000;          /* Turn off all B pins, including LED */
    _delay_ms(1000);                                           /* wait */

  }                                                  /* End event loop */
  return (0);                              /* This line is never reached */
}
```

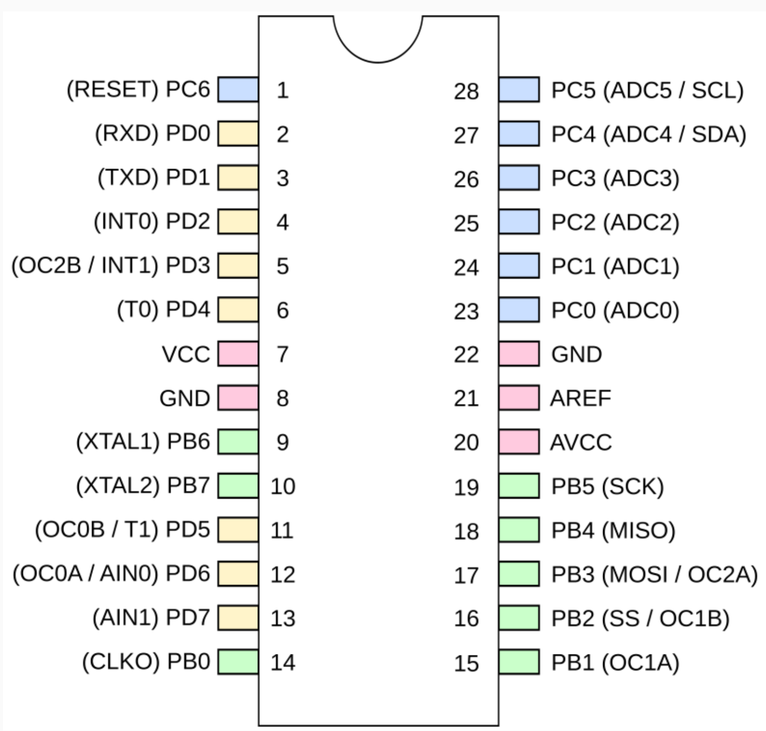
Outline

Peripherals

Useful Built-in Hardware

- Timers: AVR ATMegas have three internal timer/counters, useful for counting, timing, and scheduling events
- Timers also make PWM easy: Arduino "analog" pins
- Interrupts: Internally- or externally-triggered
  Run code whenever an event happens
- Serial I/O: built-in hardware for
  USART, SPI, I2C serial protocols
- ADC: Convert analog voltage to digital numbers
- EEPROM Memory: Read/write memory that
  doesn't get lost when the power goes out

AVR Pinout Diagram



Peripherals

... are Awesome

- The ADC, serial, and timer/counter peripherals run independently of the CPU
- All of the peripherals can trigger interrups
  your code doesn't have to wait for incoming serial data, but can instead be interrupted only when a new byte comes in
- Clever use of these features enable your chip to do many things at once
- Each of these peripheral devices are very flexible
- (Arduino hides a lot of this from you)
- (erm... I mean, does a lot of this for you)

Peripherals

... Require Configuration

- You have a bunch of configuration to do
- I/O – select input or output mode,
  hi-z or pullup if input
- PWM: base timer clock speed, set PWM mode
  toggle pin?
  trigger interrupt?

The Datasheet is Your Friend

or Maybe Frenemy

- Datasheet for ATMega48/88/168 is *660 pages long*
- Encyclopedia, not novel
- Page one and two are a really good read
- After that, skip to the chapters you need
- Read chapter intro, try to understand the block diagram
- Now you're ready to configure the Registers

Registers

The Secret to Control

- *Registers* ("special function registers") are fixed memory locations with side-effects
- Read and write just like a normal variable
- Each register byte is bits – think of each bit a switch
- Each switch has a side-effect,
  depends on which register, which bit
- In `blinkLed.c`, we wrote `PORTB = 0b00000010;`
- Setting this register's value flips the number 1 bit,
  turns on PB1

Intro to Hardware Configuration

Input/Output Pins

- Don't usually think of them as being "hardware peripherals" but even the I/O pins need configuration
- Arduinisti are used to calling `pinMode()` to get this done
- In C, write directly to the special function register that controls the pin's data direction, the Data Direction Register (DDR).
- Code: `DDRB = 0b00000010` sets pin one in PORTB (PB1) into output mode.
- Warning: The AVR hardware (and C) starts counting at 0
- Everything the chip can do is configured by setting and clearing bits in registers

Configuration Example

Bits in Registers

- Let's set up PWM on PB1 (OC1A) to run at around 1KHz with no CPU involvement
- We need to configure three things:
  – choose a Timer clock source
  – set up PWM mode ("Fast PWM, 10-bit mode") – enable automatic output on PB1
- So let's have a look at the register description and see how it works

Timer 1 Register Descriptions

16.11 Register Description

16.11.1 TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 16-4. Waveform Generation Mode Bit Description[1]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX

Segue to Idiomatic C

All that binary is miserable

- To set 10-bit Fast PWM mode, we need to set bits WGM10, WGM11, and WGM12
- So we could look up the bits in the register, find out that WGM10 is bit zero
  and WGM11 is bit one. We then assign `TCCR1A = 3;`, which is the sum of the two bits in binary.
- Now we also need to enable output on PB1, so we look up that bit: `COM1A0`.
- Oh man, it's bit number six. How many was that again in binary? Plus the three from before?
- There must be a better way!

Outline

Quick Overview

The Toolchain

Peripherals and their Configuration

C for Microcontrollers

Wrap-Up and Resources

Microcontroller Idioms

C is Not C

- A lot of microcontroller programming is necessarily low-level we've got no OS here, and we're flipping bits
- Microcontroller C `can be` ANSI C, but you're going to use a different part of it than you're used to
- C has provisions for doing bit-wise manipulations, bit shifts, bitwise `AND` and `OR`, etc.
- Enter Bit Twiddling: – bit-shifting – bit-masking – bitwise logical operations

blinkLED.c Again

```c
                                                    /* Blinker Demo */

// ------- Preamble -------- //
#include <avr/io.h>                         /* Defines pins, ports, etc */
#include <util/delay.h>                      /* Functions to waste time */

int main(void) {

  // -------- Inits --------- //
  DDRB = 0b00000010;           /* Data Direction Register B:
                                  writing a one to the bit
                                  enables output. */
  // ------ Event loop ------ //
  while (1) {

    PORTB = 0b00000010;            /* Turn on one LED bit/pin in PORTB */
    _delay_ms(1000);                                           /* wait */

    PORTB = 0b00000000;          /* Turn off all B pins, including LED */
    _delay_ms(1000);                                           /* wait */

  }                                                  /* End event loop */
  return (0);                              /* This line is never reached */
}
```

Idiomatic AVR C

```c
// ------- Preamble -------- //
#include <avr/io.h>                         /* Defines pins, ports, etc */
#include <util/delay.h>                      /* Functions to waste time */

#define LED       PB1
#define LED_PORT  PORTB
#define LED_DDR   DDRB

int main(void) {
  // -------- Inits --------- //
  LED_DDR |= (1 << LED);                           /* Enable output on LED */

  // ------ Event loop ------ //
  while (1) {

    LED_PORT |= (1 << LED);                             /* Turn on LED pin */
    _delay_ms(1000);                                           /* wait */

    LED_PORT &= ~(1 << LED);     /* Turn off all B pins, including LED */
    _delay_ms(1000);                                           /* wait */

  }                                                  /* End event loop */
  return (0);                              /* This line is never reached */
}
```

Bit-Shifting

(1 << PB1)

- What's going on?
- 1 in binary is `0b0000001`
- `<<` is the left bitshift operator
- Starting with 1, shifting over:
```
0b00000001 = (1 << 0)
0b00000010 = (1 << 1)
0b00000100 = (1 << 2)
0b00001000 = (1 << 3)
```
- `#include io.h` at the top of the code includes the following definitions:
```
#define PB0 0
#define PB1 1
```
etc.

Bitwise Logic:

Turning on Multiple Bits with OR

- We want to turn on two LEDs: `PB1, PB7`
- Bitwise `OR`: |
```
0b00000010 = (1 << PB1)
0b10000000 = (1 << PB7)
0b10000010 = (1 << PB1) | (1 << PB7)
```
- Bitwise `OR` applies the logical `OR` function down the columns

Bitwise Logic:

Toggling bits with `XOR`

- Bitwise `XOR`: `^`
- $\rightarrow$ 1 if and only if two bits differ
  $\rightarrow$ 0 if both bits are 1 or both are 0
- ```
   0b00001111
  ^0b00000010
   0b00001101
  ```
- ```
   0b11110000
  ^0b00000010
   0b11110010
  ```

Bitwise Logic:

Negating bits with `NOT`

- Bitwise `NOT`: `~`
- `~0b00001111` $\rightarrow$ `0b11110000`
- Easy!

Bitwise Logic:

Clearing bits with NOT and AND

- Bitwise `AND`: `&`
- → 1 if *both* bits are 1
  → 0 otherwise
  makes it nice for zeroing things out (bit-mask)
- ```
   0b00001111
  &0b11111101
   0b00001101
  ```
- And using `NOT` is a convenient way to create
  `0b11111101 = ~(1 << PB1)`
- So to turn off the bit corresponding to `PB1`:
  `PORTB = PORTB & ~(1 << PB1)`
  or `PORTB &= ~(1 << PB1)`

Bitwise Logic:

Summary

- Pshwew!
- Set bit with `OR`:
  `PORTB |= (1 << PB1);`
- Toggle bit with `XOR`:
  `PORTB ^= (1 << PB1);`
- Clear bit with `AND` and `NOT`:
  `PORTB &= ~(1 << PB1);`
  ```

Application to Configuring a Register

There was a point to all this...

- Remember we had these bit names from the datasheet
- We wanted a way to set/clear bits in registers to control various peripherals
- Now we can set/clear them by name:
  `DDRB |= (1 << PB1);` sets the “PB1”th bit in `DDRB`
- And we can set multiple bits:
  ```
  TCCR1A |=
  ( (1 << WGM10) | (1 << WGM11) | (1 << COM1A1) );
  ```
- And if we wanted to clear the `COM` bit without changing the mode bits:
  `TCCR1A &= ~(1 << COM1A1);`

Make Your Life Easier

That was hard!

- It was. You'll get used to it.
- Alternative:
  ```
  #define BV(x)           (1 << x)
  #define setBit(P,B)     (P |= BV(B))
  #define clearBit(P,B)   (P &= ~BV(B))
  #define toggleBit(P,B)  (P ^= BV(B))
  ```
- Macro versions are clearer, easier to read
  but other people will use the basic form so you need to understand it

blinkLED.c the Way I'd Probably Write It

```c
// ------- Preamble -------- //
#include <avr/io.h>                          /* Defines pins, ports, etc */
#include <util/delay.h>                       /* Functions to waste time */

#define LED               PB0
#define LED_PORT          PORTB
#define LED_DDR           DDRB
#define BV(x)             (1 << x)
#define setBit(P,B)       P |= BV(B)
#define clearBit(P,B)     P &= ~BV(B)
#define toggleBit(P,B)    P ^= BV(B)

int main(void) {
  // -------- Inits --------- //
  setBit(LED_DDR, LED);                               /* set LED for output */
  // ------ Event loop ------ //
  while (1) {
    toggleBit(LED_PORT, LED);
    _delay_ms(1000);
  }                                                       /* End event loop */
  return (0);                                 /* This line is never reached */
}
```

Outline

Quick Overview

The Toolchain

Peripherals and their Configuration

C for Microcontrollers

Wrap-Up and Resources

Summary

Covered a Lot of Ground

- The AVR-GCC programming toolchain
- Million-mile overview to the important AVR peripherals and why they matter
- Rudimentary bit-twiddling and how it works with configuration registers
- In truth – this is all of the hard stuff!

What's Left?

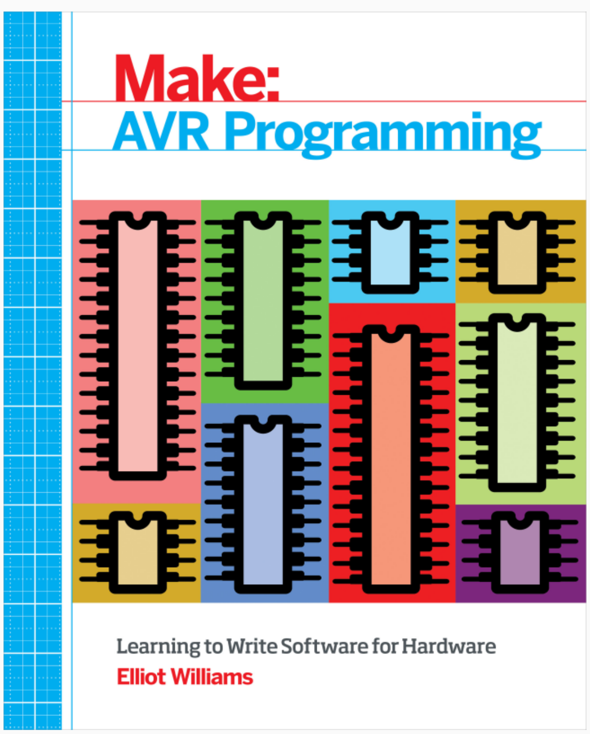
- The specifics of peripherals: how they work, how they help
- Setting up event priorities with interrupts
- Watchdog timer, sleep modes for power saving, etc.
- Man, this chip does a lot

Resources

- My AVR Site: `www.littlehacks.org` (new stuff added weekly these days)
- Old Material from AVR classes I've taught: `wiki.hacdc.org/index.php/AVR_Microcontroller_Class_2011`
- Bruce Land's Cornell University Engineering Course (for the serious down-low)
- Hackaday, Make Blog, Sparkfun, LadyAda for inspiration
- VUSBTiny project (build your own minimal programmer)

Questions

Oh yeah, I wrote a book



The End

◂ Outline