Microcontrollers

Class 1: Serial and Digital I/O

March 7, 2011

Outline

Outline

Outline

Quick Tour of the Board

Pins, Ports, and Their Registers

Boolean Operations

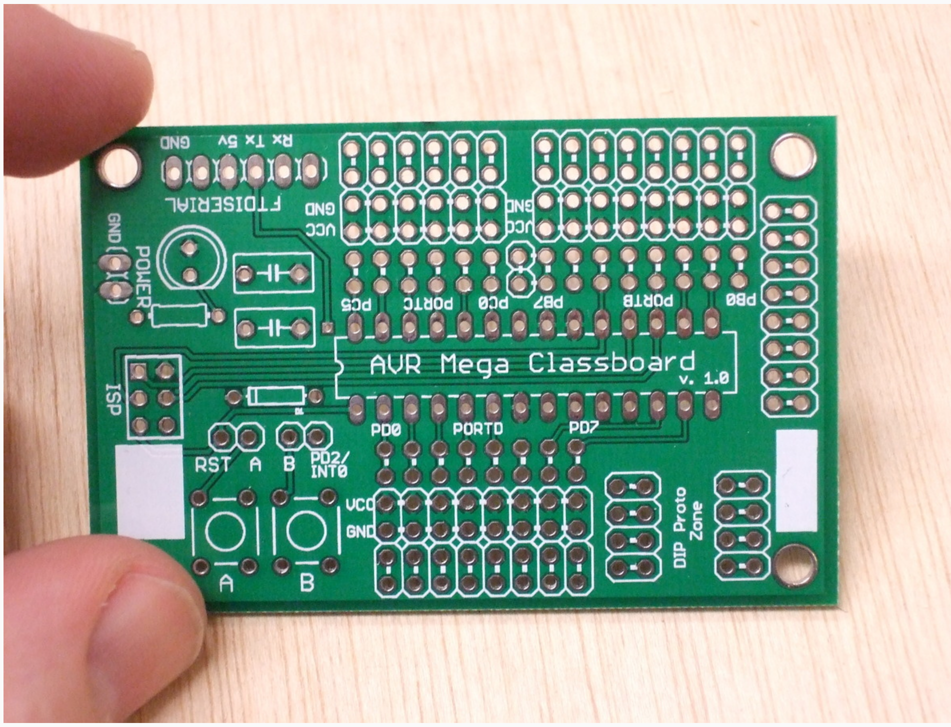
Cylon Eyes

Digital Input and Testing Particular Pin States

Debouncing

Serial Communications

See http://wiki.hacdc.org/index.php/Avr2011_kit



Outline

- Quick Tour of the Board
- Pins, Ports, and Their Registers
- Boolean Operations
- Cylon Eyes
- Digital Input and Testing Particular Pin States
- Debouncing
- Serial Communications

Hello World Example

Blinkies!

- Last class, showed an example that turned a pin on and off
- Sections of the C code:
  preamble – includes and defines
  function definitions (didn't have any)
  main function (chip initialization and endless loop)
- The main loop twiddled a bit back and forth in a memory register, and that made Vcc and GND volts appear on a particular pin.
- But let's flesh that all out a little more...

Registers

Special memory locations

- Usually we think of memory as being a place to store info
- In micros, some special memory regions change the way the chip behaves: Registers
- DDRx register from initialization of LED blinking demo
- Writing a "one" to a bit in the DDRx register sets up a corresponding pin for output
- There's a similar mapping from the PORTx register to the output of the pins: writing a 1 to a bit in PORTx sets the corresponding pin at Vcc, 0 to GND
- When the pins are configured for input, the PIN registers read 0 if a low voltage is present on its pin, and 1 for high

Addressing the Pins

Writing bits to registers

- So, say we're working on PORTB, and we want to set pin PB2 and PB6 to 5v (to light up some LEDs)
- Write a 1 in the 2nd and 6th slots in the PORTB register
- Write it in binary directly: `PORTB = 0b01000100;`
- Write it using its decimal value: `PORTB = 68;`
- Write it in hex: `PORTB = 0x44;`
- Write it using a bit-value macro:
  `PORTB = _BV(2) | _BV(6);`

Outline

The Math

Bit-shift Operators

- <<: Left shift
- >>: Right shift

Binary logic

- & : AND
- |: OR
- ^: XOR
- ~: NOT

Bit-shifting

Left shift: <<

- Very handy: say you want a 1 in the pin-3 place: 00001000
- Start with 1: 00000001
- Shift it over 3:
  `1 << 3` = 00001000
- Or using the pin-name macros: `1 << PD3`
- `#define _BV(bit) (1 << (bit))`

Right shift: >>

- Start with 12: 00001100
- Shift 2: `12 >> 2` = 00000011
- Note that right-shift is like dividing by 2^n: handy
- (Similarly, left-shift is like multiplying by 2^n)

Using Shifts

Practical examples:

- `PORTD = (1 << 3);`
- `PORTD = (1 << PD3);`
- `PORTD = (1 << (1+2));`
- `j = 3; PORTD = (1 << j);`
- `j = 3; PORTD = _BV(j);`

Set Two Pins

Addition:

- Say we want PB3 and PB4 both on
- Add them together?
- `PORTB = _BV(PB3) + _BV(PDB);` will work
- After all:
$$\begin{array}{rl} & 00001000 \\ + & 00010000 \\ = & 00011000 \end{array}$$
- Works if you're just setting the port using `PORT = something;`
- But you never see bitwise addition. Why?

Turning Pins On

Why addition won't cut it

- What if you don't know (or care) what LEDs are already on, but you want to turn on PD3?
- `PORTD = PORTD + _BV(PD3);`?
- If PD3 is already on: $\begin{array}{rl} & 00000100 \\ + & 00000100 \\ = & 00001000 \end{array}$ Ouch!
- Could be worse: $\begin{array}{rl} & 01111100 \\ + & 00000100 \\ = & 10000000 \end{array}$
- We need an OR statement

Turning More Pins On

The OR statement: |

- `0b01000010 = _BV(1) | _BV(6)`, so it's as good as addition
- OR turns on a bit if this bit *or* that bit is on
- `PORTD = PORTD | _BV(PD3);`
- If PD3 is not on: $\begin{array}{rl} & 11000000 \\ | & 00000100 \\ = & 11000100 \end{array}$
- If PD3 is already on: $\begin{array}{rl} & 11000100 \\ | & 00000100 \\ = & 11000100 \end{array}$ Yay!
- Turn on PD3, PD4, PD5?
  `PORTD = PORTD | (_BV(PD3) | _BV(PD4) | _BV(PD5));`
- And here's a nice shorthand: $X = X + Y \quad \rightarrow \quad X += Y$
  `PORTD |= _BV(PD3) | _BV(PD4) | _BV(PD5);`

Turning Pins Off

The AND statement: &

* AND turns on a bit if this bit and that bit are *both* on

- ```
    11111110
  & 10001101
  = 10001100
  ```
- Can think of AND as masking out the the off bits
- Use it to turn off PD3: `PORTD &= 11110111?`
- `PORTD &= (_BV(PD7) | _BV(PD6) | ... skip PD3 ... | _BV(PD0) );`
- Works, but it sucks.

Turning Pins Off II

The NOT statement: ~

- There must be a better way to make 11110111
- `_BV(PD3)` = 00001000
- `~_BV(PD3)` = 11110111
- Turn off PD3: `PORTD &= ~_BV(PD3);`
- Turn off PD3 and PD4:
  `PORTD &= ~(_BV(PD3) | _BV(PD4));`
  Careful with those parentheses!
- Also remember this in terms of the fundamentals:
  ```
  PORTD &= ~((1 << PD3) | (1 << PD4));
  PORTD &= ~((1 << 3) | (1 << 4));
  ```
  ```

Toggling a Pin

The XOR statement: ^

- A lot of the time, it's handy to be able to toggle a bit
- `PORTD ^= _BV(PD3);`
-
```
     11111001
 ^   00001000
 =   11110001
```
-
```
     11110001
 ^   00001000
 =   11111001
```

Outline

One Last Part...

...then Cylon Eyes

- So we know how to turn on bits, and how to turn them off
- How do we make cylon eyes?
- Start with light 0 on.
  Turn off the 0th, turn on the 1st, pause
  turn off the 1st, turn on the 2nd, pause
  etc
- `PORTD &= ~_BV(PD0); PORTD |= _BV(PD1); delay`
  `PORTD &= ~_BV(PD1); PORTD |= _BV(PD2); delay` etc.
- `{PORTD &= ~_BV(i) ; PORTD |= _BV(i+1); delay }`
- And make `i` range from 0 to 7 and back again
  (being very careful about endpoints)

Basic Looping

The For loop

- `for(i=0; i < 7; i = i + 1){...}`
- Repeats the block in parentheses a bunch of times.
- First time, `i` $= 0$.
- Then it checks if `i` < 7.
  If not, it skips the block and moves on.
  If so, it executes the next command and then the block.
- So in our case, it executes the block with `i`$= 0, 1, 2, 3, 4, 5, 6$
  and then is done.
- `for(i=7; i > 0; i = i - 1){...}` and a different block
  will bring it back down
- `i`$= 7, 6, 5, 4, 3, 2, 1$

Digital Output: Summary

Configure, Write, Done

- So at this point, we're all set for doing all sorts of cool stuff with digital output
- First, set up the DDR for output (on pins of your choosing) by writing a 1 to the relevant bit
- The set the PORT register to set pins high or low, depending
- Loop, repeat

Outline

Initializing for Input

It almost seems too easy...

- To initialize for *output* set bit to one
  DDRx = _BV(whatever)
- For *input*, want to set the bit to zero instead.
- But zero is the default value. Done!

Initialization for Input

One wrinkle: Initialize a pullup resistor

- A pullup resistor ties the input pin to 5v (internally) when it's not pulled low from outside
- Often want a pullup with input
- Why? Simplest input circuit is a switch from pin to ground
- AVR's PORTn does double-duty.
  In output mode, controls output.
  In input mode, selects the pullup
- So often set PORTn to one to enable the pullup:
  `PORTB |= _BV(PB3);`

Reading the Input

Reading the input register

- Input values in the PINx register
- Can read them like `readIn = PINB;`
- `readIn` will contain an 8-bit number, each bit corresponding to the voltage state of all 8 of its pins.

Reading one pin: the most common case

- `PIND & _BV(PD3);`
- If PD3 has more than 1.25v on it, we'll get 00001000
- If PD3 has less than 1.25v on it, we'll get 00000000
- Test of pin state: `if((PIND & _BV(PD3)) == 0){...}` or `if(!(PIND & _BV(PD3))){...}`
- See *simplePushbutton.c*

Outline

The Real World

Switching Noise

- In reality, switches make/break contact a bunch of times as you press it
- Two pieces of metal touching, bending, with different resistance all over
- If you're trying to make a per-button-press device, this can cause troubles
- Symptom: Get multiple presses for what you thought was a single press
- Solution: Debouncing

Debouncing

Many Approaches

- Delay I: turn on after a short delay after first button press
- Delay II: wait short period of time after first press, test again if it's still pressed
- Integrate: test N times in a row, with a delay between, decide the button is pressed if more than M hits
- There are many others. There was even a Hackaday competition recently for favorite debounce algorithms (`http://hackaday.com/2010/11/09/debounce-code-one-post-to-rule-them-all/`)
- `http://www.ganssle.com/debouncing.htm`
- I'll send code around for you to experiment with
- Advertisement for Hardware Timers!

When To Debounce?

To Debounce

- When you're counting events
- When you need to know how long the button is held down
- When it's not really a button, but an analog voltage, and it spends a bunch of time in the dreaded 0.8v - 1.5v range

Or Not to Debounce

- When all you care about is on/off, don't mind the bounce
- When other parts of the code act as a delay

Outline

Simple Serial

The easiest way to get rich debugging info

- The microcontroller really comes into its own as an *interface*
- The USART serial port (and a USB serial cable) is the easiest way to get data to and from your computer
- AVR has a built-in hardware serial machine, all you have to do is load its buffer up
- This is your first include of a non-standard file: #include "USART88.h"
- If you're curious how I wrote them, the USART serial section of the datasheet is a good place to start. Dive in!
- ... or just look at examples and monkey it.

Using USART88.h

What Do the Functions Do?

- `#define BAUDRATE 9600`
- `initUART()`:
  uses BAUDRATE to set up the baud rate
  then some bits in the USART config register are set for stop bits and parity
- `transmitByte()`:
  wait for the USART busy flag to become unset
  load the data into the transmit buffer register
  walk away, letting the hardware serial do the rest
- `receiveByte()`:
  once initialized, the hardware USART is always receiving
  wait for the USART received-data flag to be set
  return the data

Serial Interfacing

For the Big Computer

- Screen: for terminal emulation
  `screen /dev/ttyUSB0` or even `screen /dev/ttyUSB0 9600`
- Python: pyserial `http://pyserial.sourceforge.net/` for everything else

Serial Ideas

Things I Have Done With USART Serial

- Control 4x4x4 LED cube from my desktop
- Simple menu system for a logging accelerometer
- GPS datalogger
- Parallax RFID readers
- Hook up 2 AVRs (radio, IR LED, wires)
- Debug, debug, debug!

The End

◂ Outline