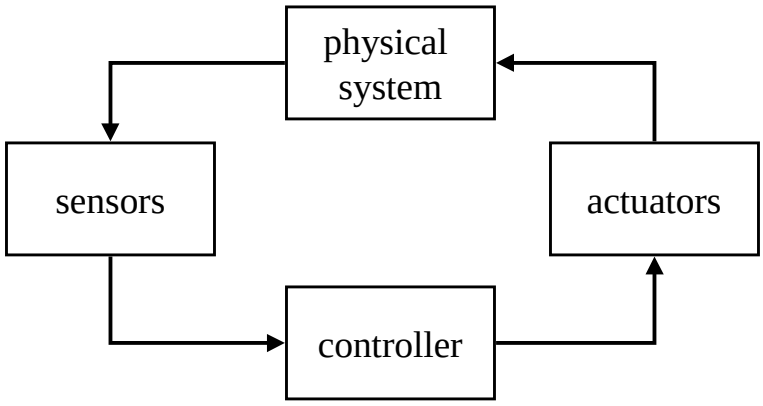
Controlling and reacting to the environment

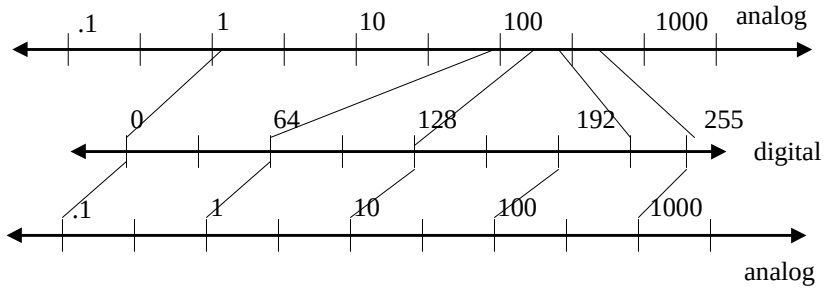
- To control or react to the environment we need to interface the microcontroller to peripheral devices
- Things we want to measure or control
  - light, temperature, sound, pressure, velocity, position
- Sensors
  - switches, photoresistors, photodiodes, phototransistors
- Actuators
  - motors, relays, LEDs,
- Software
- Microcontroller
  - executes software
  - may contain specialized interfaces to sensors and actuators

Typical control system



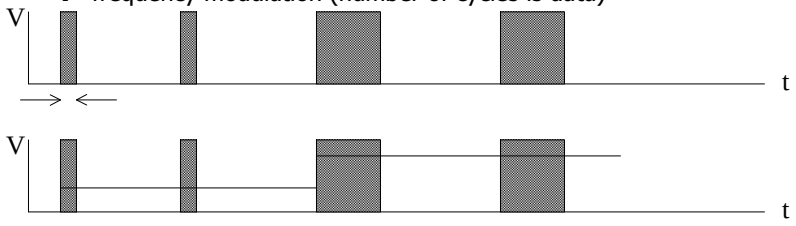
Analog to digital conversion

- Map analog inputs to a range of binary values
  - 8-bit A/D has outputs in range 0-255
- What if we need more information?
  - linear vs. logarithmic mappings
  - larger range of outputs (16-bit a/d)



Digital to analog conversion

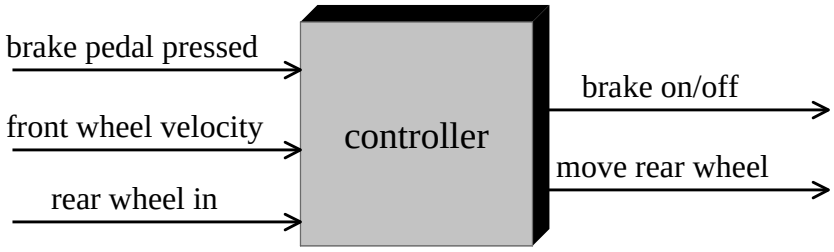
- Map binary values to analog outputs (voltages)
- Most devices have a digital interface
- Time-varying digital signals
  - pulse-code modulation (width of pulse is data)
  - pulse-width modulation (creates an average voltage)
  - frequency modulation (number of cycles is data)



Anti-lock brake system

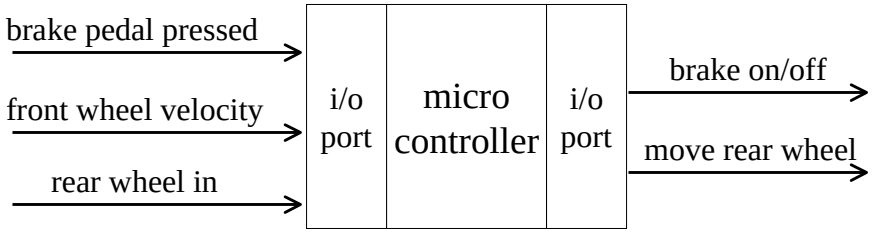
- Rear wheel controller/anti-lock brake system
  - normal operation
    - regulate velocity of rear wheel
  - brake pressed
    - gradually increase amount of breaking
    - if skidding (front wheel is moving much faster than rear wheel)
      - temporarily reduce amount of breaking
- Inputs
  - brake pedal
  - front wheel
  - rear wheel
- Outputs
  - pulse-width modulation rear wheel velocity
  - pulse-width modulation brake on/off

Rear wheel controller/anti-lock brake system



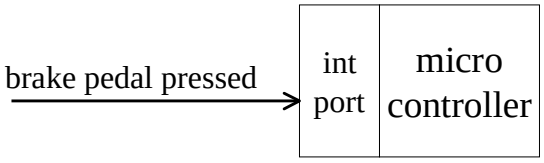
Basic I/O ports (brakes)

- Check if brake pedal pressed
  - brakePressed = peek (brakePedalPort)
- Turn brake on/off
  - poke (brakePort, onOff)
- Move rear wheel
  - poke (rearWheel, onOff)



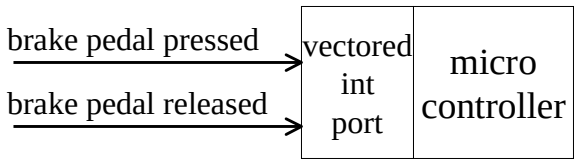
Polling vs. interrupts

- Software must repeatedly check for the brake pedal port
- Use automatic detection capability of processor
- Connect brake pedal to interrupt port or input capture port
  - register interrupt handler
    - poke (InterruptVector, brakePedalHandler)
  - interrupt handler
    - brakePressed = true
    - poke (InterruptReg, ClearInterruptFlag)



Detect if pedal moves

- Check if brake pedal pressed or released
- Connect brake pedal to interrupt port or input capture port
  - register interrupt handler
    - poke (InterruptVector, brakePedalPressedHandler)
    - poke (InterruptVector, brakePedalReleasedHandler)



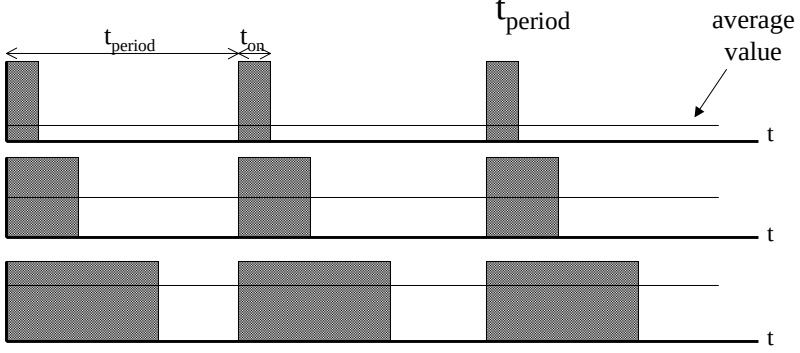
More interrupts

- Periodically check brake pedal
  - set timer interrupt
    - poke (TimerCounter, timeInterval)
    - poke (InterruptVector, brakeHandleHandler)
  - interrupt handler
    - brakePressed = peek (brakePedalPort)
    - poke (InterruptReg, ClearInterruptFlag)
- Problems with this approach?

Pulse-width modulation

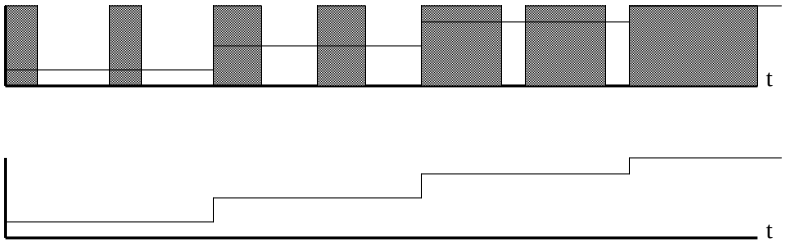
- Main idea: pulse a digital signal to get an average "analog" value
- The longer the pulse, the higher the voltage

$$\text{Pulse-width ratio} = \frac{t_{on}}{t_{period}}$$



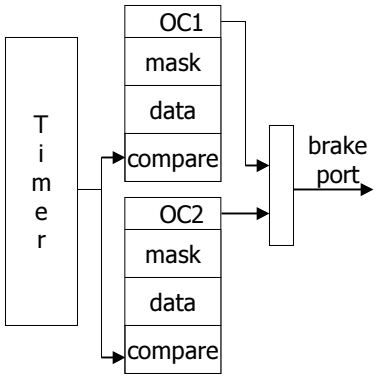
Pulse-width modulation for brakes

- To pump the brakes gradually increase the duty-cycle (t_{on}) until we stop



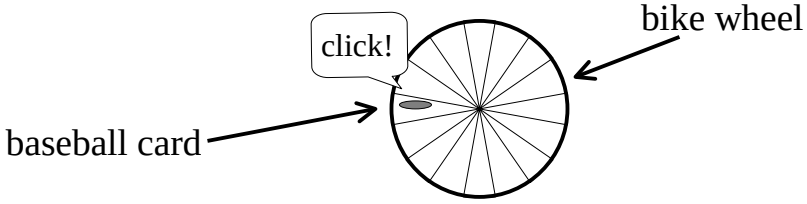
Brake pump setup

- When output compare register OC1 matches timer, set bit high
- Use OC2 to set bit low
- OC1 is attached to the brake port
  - poke (OC1MaskReg, brakePort)
- OC1 turns on the brakes
  - poke (OC1DataReg, brakesOn)
- Set period
  - poke (TOC1Reg, period)
- OC2 turns off the brake
  - poke (TimerControlReg, brakesOff)
- Set duty cycle for brakes
  - poke (TOC2Reg, brakeDutyCycle)



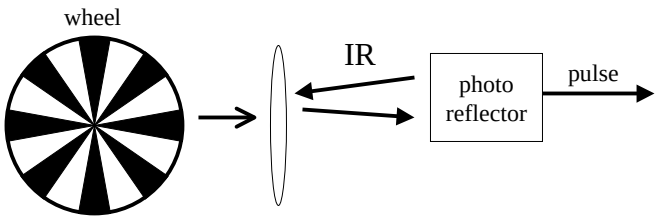
Shaft encoders

- Need to determine the rear wheel velocity
- Use sensor to detect wheel moving
- Determine speed of a bicycle
  - attach baseball card so it pokes through spokes
  - we know number of spokes
  - count clicks per unit time to get velocity
- Baseball card sensor is a shaft encoder



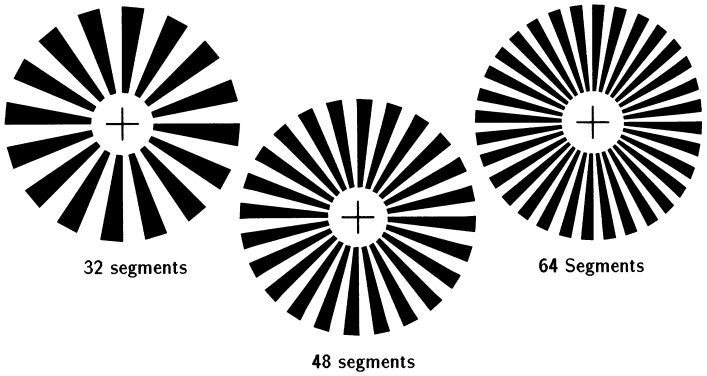
Shaft encoders

- Instead of spokes we'll use black and white segments
- Black segments absorb infrared light, white reflects
- Count pulses instead of clicks



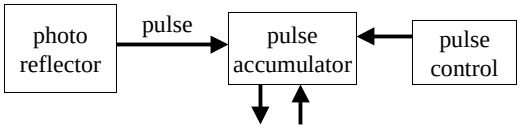
IR reflective patterns

- How many segments should be used?
  - more segments give finer resolution
  - fewer segments require less processing



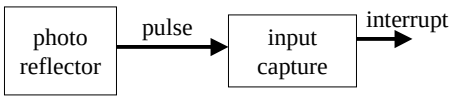
Interfacing shaft encoders

- Two ways
  - pulse accumulator
  - input capture
- Set up pulse accumulator register
  - poke (PulseAccControlReg, risingEdge|enablePulseAcc| rearWheelIn)
  - poke (PulseAccumulatorReg, 0)
- Get velocity and clear count
  - velocity = peek (PulseAccumulatorReg)
  - poke (PulseAccumulatorReg, 0)

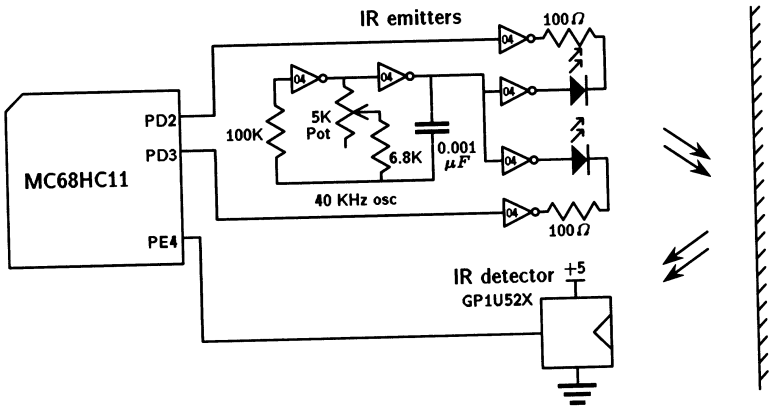


Shaft encoder with input capture

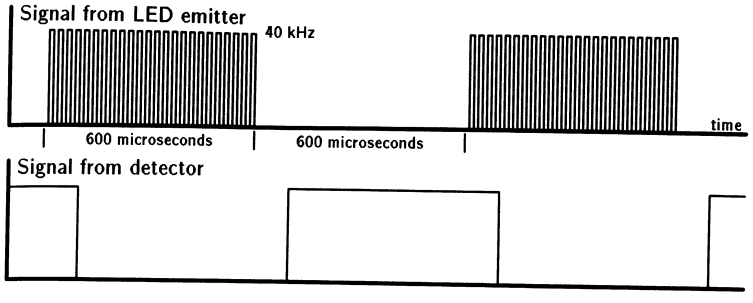
- Generate an interrupt on every pulse
- Set up to interrupt on rising edge pulses on input capture 2
  - poke (InterruptReg, inputCapture2)
  - poke (TimerControl2Reg, (risingEdge << inCapture2BitPosition))
- Register interrupt handler
  - poke (InterruptVector, pulseCounterHandler)
- Interrupt handler
  - pulses = pulses + 1
  - poke (InterruptReg, ClearPulseInterrupt)
- Get velocity and clear count
  - velocity = pulses
  - pulses = 0



IR proximity detector



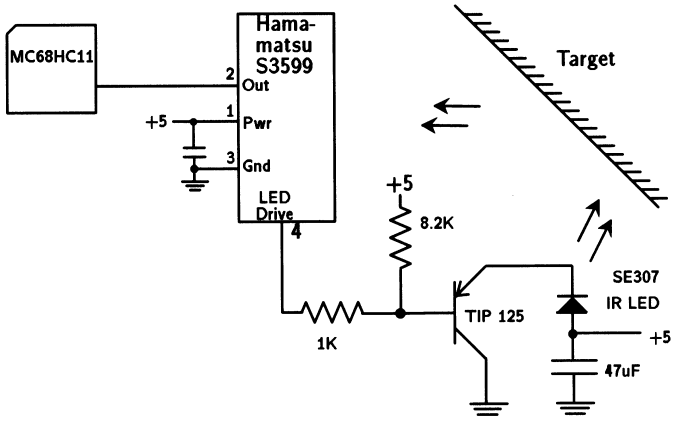
IR frequency modulation



Proximity code

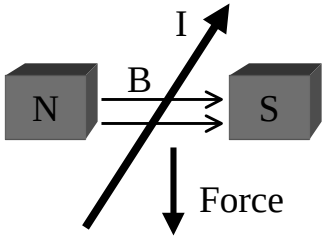
```
void ir_detect() {
        bit_set (portD, emitterPort)        // turn on emitter
        sleep (600us)                       // wait 1667Hz
        val_on = peek (portE)               // read detector
        bit_clear (portD, emitterPort)      // turn off emitter
        sleep (600us)                       // wait
        val_off = peek (portE)              // read while emitter off
        if ((val_off & ~val_on & detectorPort) == detectorPort)
                detectedObstacle  = true;
        else  detectedObstacle   = false;
}
```


Another proximity detector



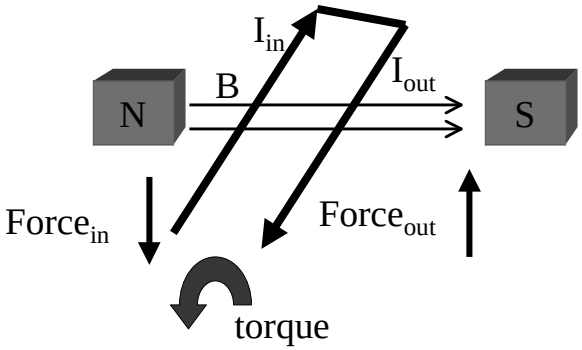
Motor basics

- Current (I) in a magnetic field (B) creates a force perpendicular to I and B
- F = I x B (use right-hand rule)



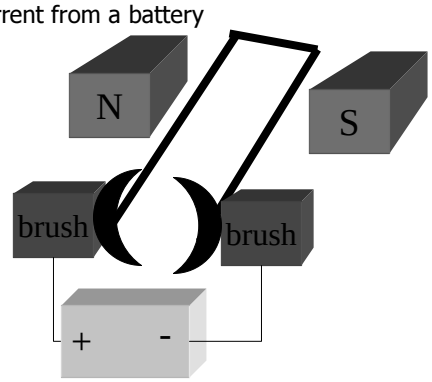
Use loop to create torque

- Loop of current create down force going in and up force coming out



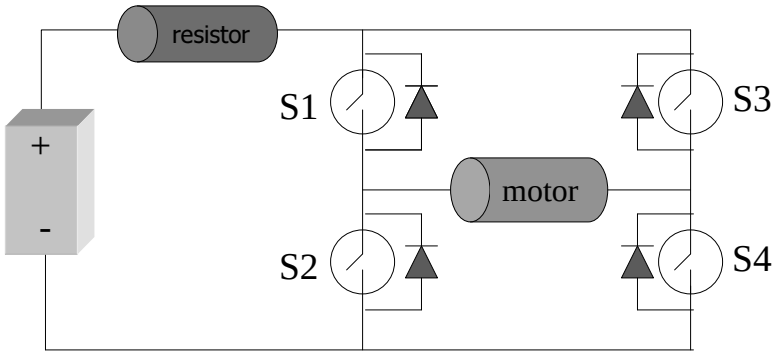
Alternating current creates motion

- Torque forces coil to vertical position
- Reversing direction of current creates motion
- Commutation: the reversal of the order of two quantities
- Brushes create alternating current from a battery

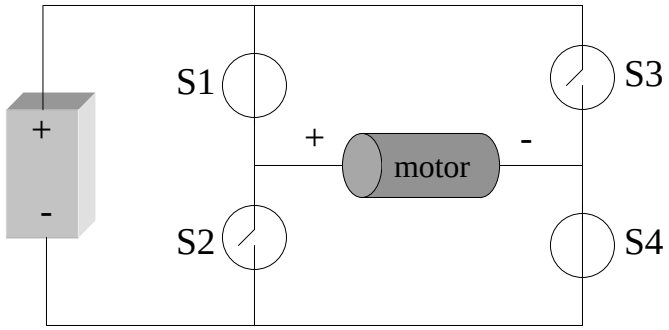


Interfacing motors

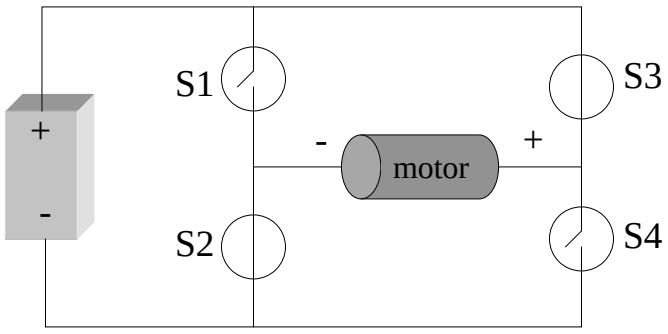
- Motors require more current than the microcontroller's ports can source
- Instead have processor control switches
- Use an H-bridge so that voltage polarity can be changed



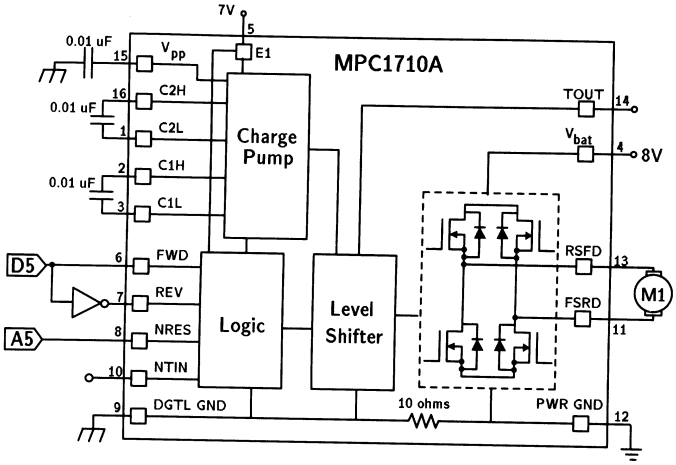
Going forward



Going backward

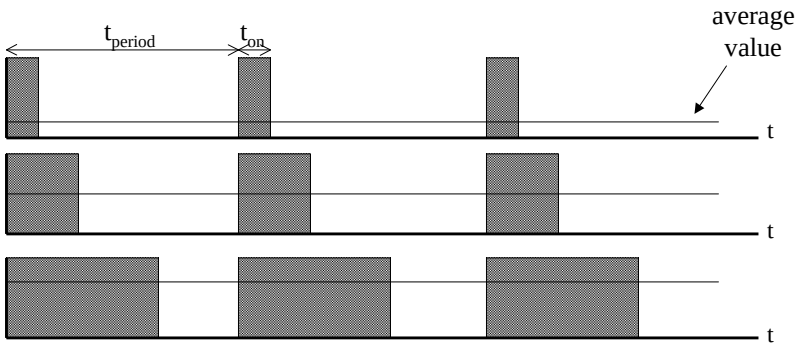


Motor driver chip



PWM to control velocity

- Similar to how we pumped the brakes, generate average voltage
- Larger duty-cycle corresponds to a higher velocity

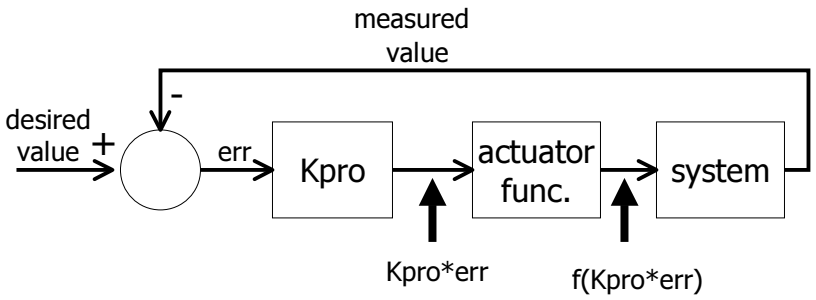


Control algorithms

- Open-loop control
  - give commands
- Closed-loop control
  - give commands
  - check results
  - modify commands
  - check results
  - ...
- Two types of closed-loop control
  - proportional-integral controller (P-I controller)
  - integral controller

P-I controller

- Main idea: scale difference between desired and measured values and use scaled value as input to the actuator function

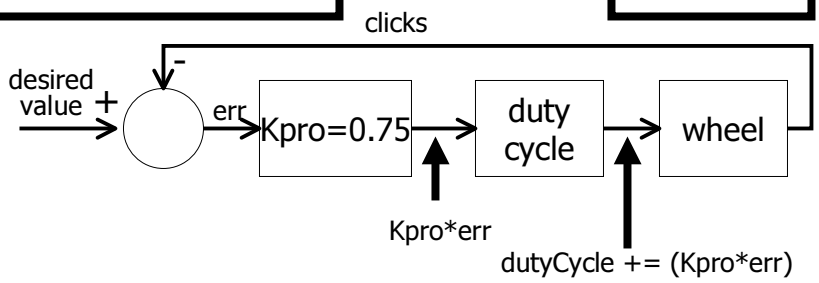


P-I controller example

step	err	duty	clicks
0	0	0	0
1			
2			
3			
4			
5			

Wheel stats

duty	clicks
0	0
25	45
50	90
75	135
100	180

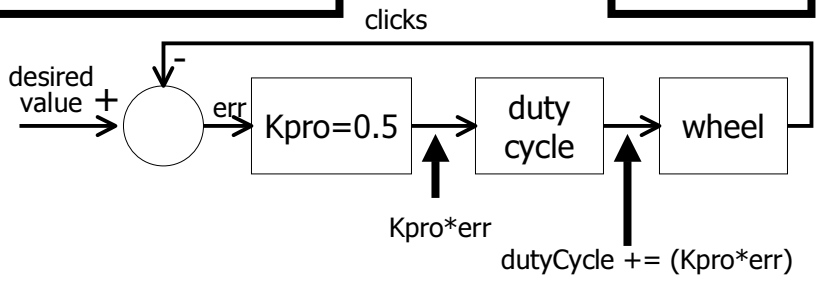


A different Kpro

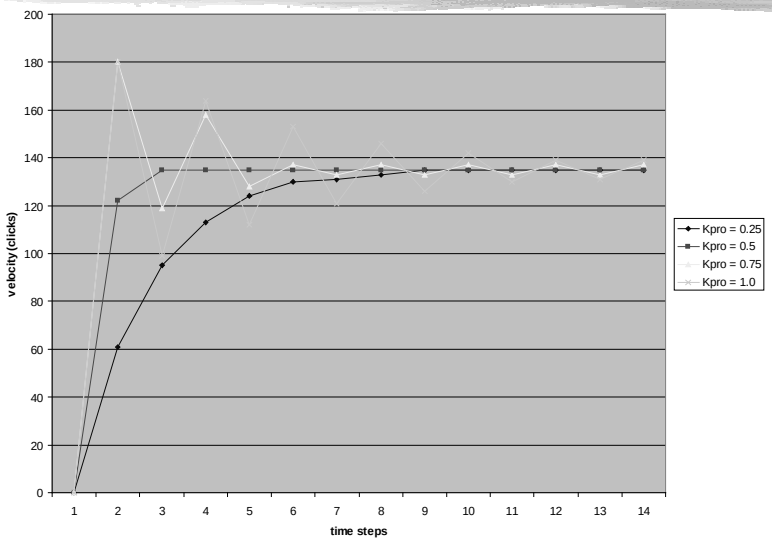
step	err	duty	clicks
0	0	0	0
1			
2			
3			
4			
5			

Wheel stats

duty	clicks
0	0
25	45
50	90
75	135
100	180

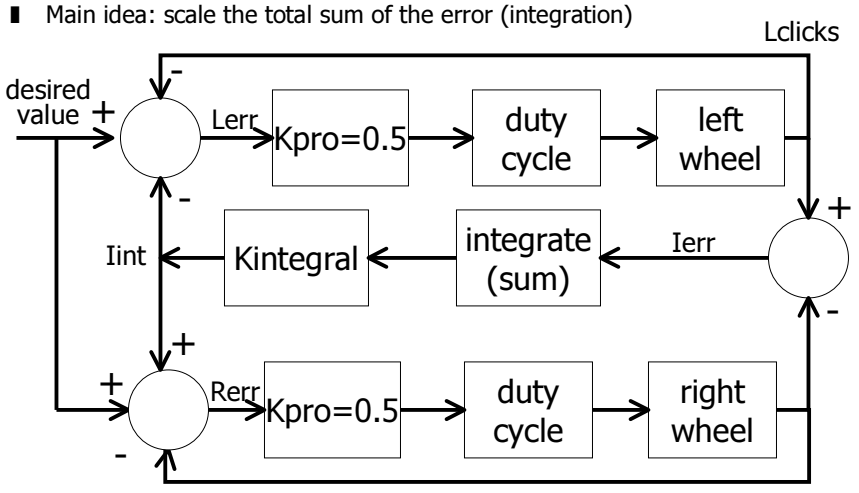


Sensitivity to Kpro



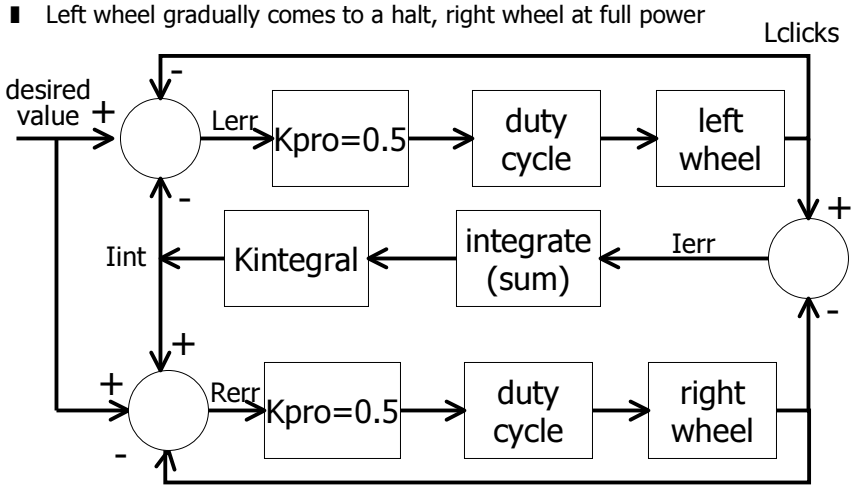
Integral controller

- Main idea: scale the total sum of the error (integration)



Stuck right wheel

- Left wheel gradually comes to a halt, right wheel at full power

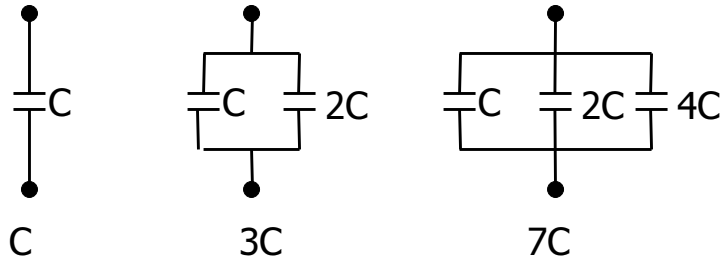


Sensitivity to time

- What happens if we run the control algorithm too slow?
- What happens if we run too fast?
- How is the appropriate period determined?

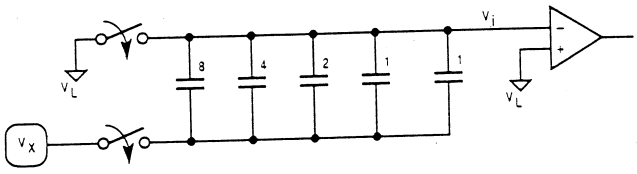
Analog to Digital Conversion

- Use charge-redistribution technique
  - no sample and hold circuitry needed
  - even with perfect circuits quantization error occur
- Basic capacitors
  - sum parallel capacitance



A/D - Sample

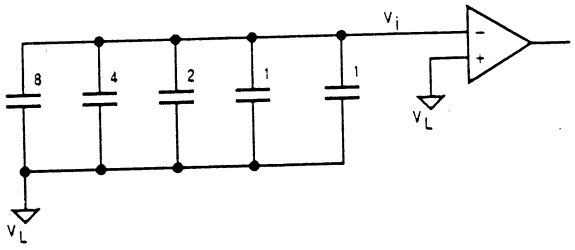
- During the sample time the top plate of all caps switched to V_L
- Bottom plate set to unknown analog input V_X
- Largest cap. corresponds to MSB
- Q = CV
- $Q_S = 16\ (V_X - V_L) = 16V_X$



(a) Sample Mode

A/D - Hold

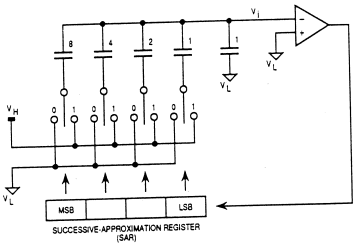
- Hold state by logically controlled analog switches
- Top plates disconnected from V_L
- Bottom plates switched from V_X to V_L
- $Q_H = 16(V_L - V_i) = -16V_i$



(b) Hold Mode

A/D - Approximation

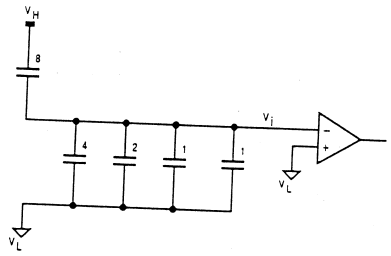
- Conservation of charge $Q_S = Q_H$ so $V_i = -V_X$
- Each cap. switched from V_L to V_H
- Output of comparator determines bottom plate voltage of cap
  - logic one: remain connected to V_H
  - logic zero: return to V_L



(c) Approximation Mode

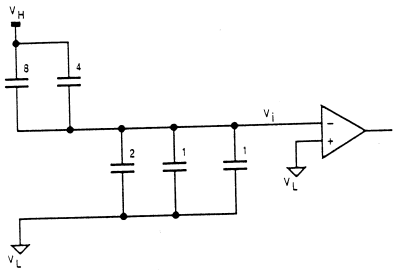
A/D Example - MSB

- Suppose $V_X = 21/32\ V_H$
- $Q_S = 16V_X = 16 * (21/32)\ V_H = 21/2\ V_H$
- $V_i = -V_X = -21/32\ V_H$
- $Q_H = 8\ (V_H - V_i) + 8\ (V_L - V_i) = 8V_H - 16V_i$
- $Q_S = Q_H$ so $21/2\ V_H = 8\ V_H - 16V_i$
- $V_i = -5/32V_H$ and comparator output is logic one



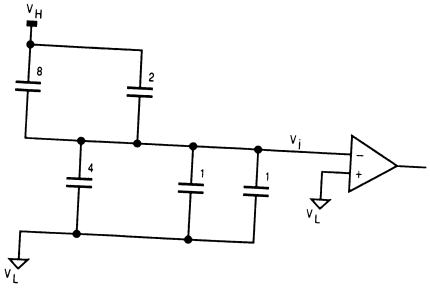
A/D Example - (MSB-1)

- $Q_H = 12\ (V_H - V_i) - 4V_i$
- $Q_S = Q_H$ so $21/2\ V_H = 12V_H - 16V_i$
- $V_i = 3/32\ V_H$
- Output of comparator is logic zero



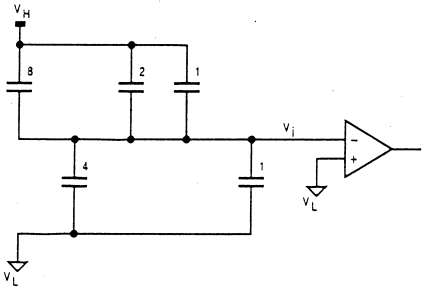
A/D Example - (MSB - 2)

- $Q_H = 10\ (V_H - V_i)\ - 6V_i = 10V_H - 16V_i$
- $21/2\ V_H = 10V_H - 16V_i$
- $V_i = -1/32\ V_H$
- Output comparator is logic one



A/D Example - LSB

- $Q_H = 11\ (V_H - V_i) - 5V_i = 11V_H - 16V_i$
- $21/2\ V_H = 11V_H - 16V_i$
- $V_i = 1/32$
- Output of comparator is logic zero
- Input sample of 21/32 gives result of 1010 or 10/16 = 20/32 or 3% error



A/D Option Control Register

- ADPU - A/D Power Up
  - zero - power to A/D unit off
  - one - power to A/D unit on
    - allow 100us after setting so analog voltages stabilize
- CSEL - Clock Select
  - zero - use E clock
  - one - use internal 1.5MHz RC clock
    - wait 10ms for everything to settle

	7	6	5	4	3	2	1	0	
	ADPU	CSEL	IRQE	DLY	CME	—	CR1	CR0	OPTION $1039
RESET:	0	0	0	1	0	0	0	0	

A/D Control/Status Register (ADCTL)

- User must write to ADCTL to initiate conversion
- CCF - Conversions Complete Flag
  - read-only indicating all four A/D result registers valid
  - bit cleared after ADCTL written
- SCAN - Continuous Scan Control
  - zero - four requested conversions performed once each
  - one - conversions continue in round-robin order
- MULT - Multiple-Channel/Single-Channel Control
  - zero - four consecutive conversions on single channel
  - one - perform conversions on each channel in the group of four
- CD,CC,CB,CA - Channel Selects

	7	6	5	4	3	2	1	0	
	CCF	—	SCAN	MULT	CD	CC	CB	CA	ADCTL $1030
RESET:	0	0	U	U	U	U	U	U	

A/D Channel Assignments

Table 12-1. A/D Channel Assignments

CD	CC	CB	CA	Channel Signal	Result in ADRx if MULT = 1
0	0	0	0	PE0	ADR1
0	0	0	1	PE1	ADR2
0	0	1	0	PE2	ADR3
0	0	1	1	PE3	ADR4
0	1	0	0	PE4*	ADR1
0	1	0	1	PE5*	ADR2
0	1	1	0	PE6*	ADR3
0	1	1	1	PE7*	ADR4
1	0	0	0	Reserved	ADR1
1	0	0	1	Reserved	ADR2
1	0	1	0	Reserved	ADR3
1	0	1	1	Reserved	ADR4
1	1	0	0	V_H**	ADR1
1	1	0	1	V_L**	ADR2
1	1	1	0	1/2 V_H**	ADR3
1	1	1	1	Reserved**	ADR4

*Not available in 48-pin package versions.
**These channels intended for factory testing.

A/D Issues

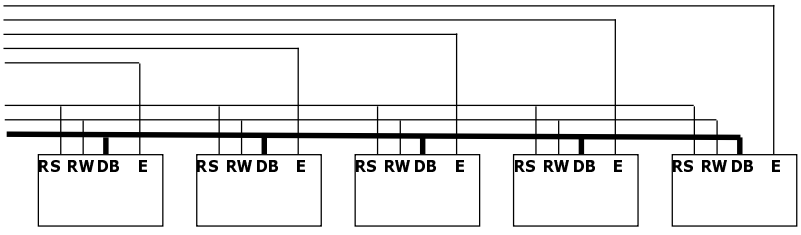
- A series resistor of ~1K and not more than 10K on input
- Capacitors from A/D inputs to V_{RL}
  - prevent errors due to system noise
  - must be properly sized
- External low-pass filter increases noise immunity
  - RC time constant important
  - don't filter important data

I/O Ports

- The are never enough I/O ports
- Techniques for creating more ports
  - port sharing with simple glue logic
  - decoders/multiplexors
  - memory-mapped I/O
  - port expansion units
- Direction of ports is important
  - single direction port easier to implement
  - timing important for bidirectional ports

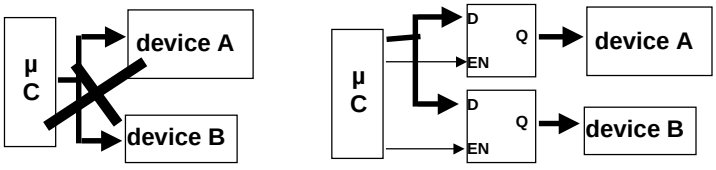
Port Sharing

- If signals all in same direction and have a separate guard signal, then able to share without glue logic
- Example: connect 5 LCD displays to microcontroller
  - can share connections to RS, RW, and DB but not E
  - changes on E affect display – must guarantee only one is activated



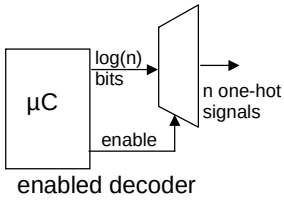
Forced Sharing

- Conflict on device signals (e.g., one signal can affect both)
  - solution is to insert intervening registers that keep signals stable
  - registers require enable signals which now need ports as well

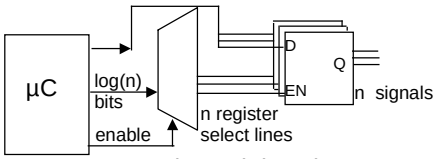


Decoders and Multiplexors

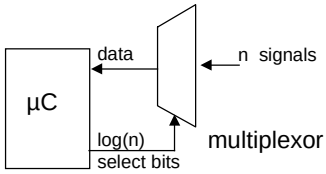
- Encode n single-bit device ports using log n bits of a controller port
  - enabled decoder: one-hot, input-only device ports
  - registered decoder: input-only (but not one-hot) device ports
  - multiplexor: output-only device ports



enabled decoder



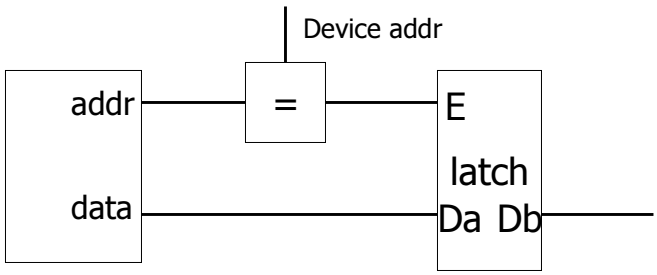
registered decoder



multiplexor

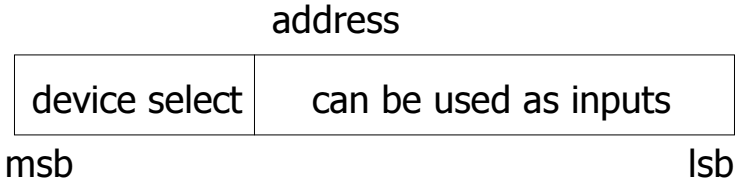
Memory-mapped I/O

- Address bus selects device
- Data bus contains data



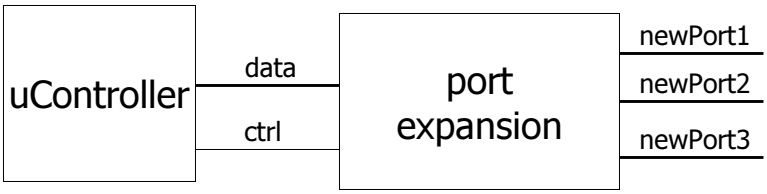
Memory-mapped I/O

- Partition the address space
- Assign memory-mapped locations
- Software
  - loads read from the device
  - stores write to the device
- Can exploit unused bits for device input-only ports



Port Expansion Units

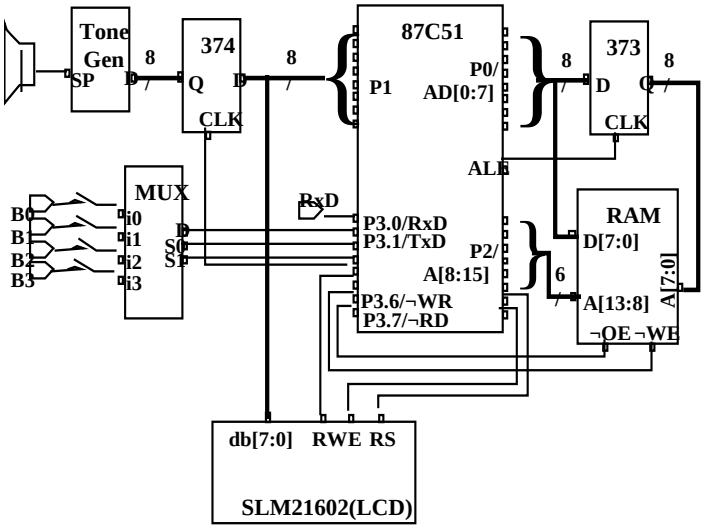
- Problem of port shortage so common port expansion chips exist
- Easily connect to the microprocessor
- Timing may be slightly different



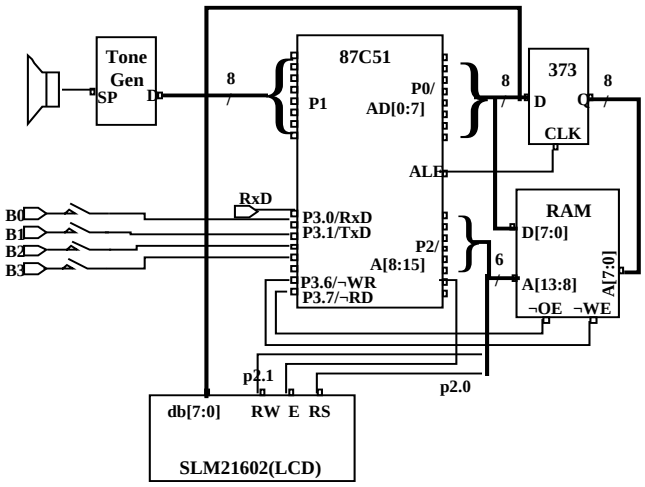
Automatically Connecting Peripherals

- Try to exploit specialized functions (e.g., UART, timers)
- Attempt to connect directly to a device port without adding interface hardware (e.g., registers), try to share registers if possible but beware of unwanted interactions if a signal goes to more than one device
- If out of ports, must force sharing by adding hardware to make a dedicated port sharable (e.g., adding registers and enable signals for the registers)
- If still run out of ports, then most encode signals to increase bandwidth (e.g., use decoders)
- If all else fails, then backup position is memory-mapped I/O, i.e., what we would have done if we had a bare microprocessor

Electronic Phonebook - Human Design

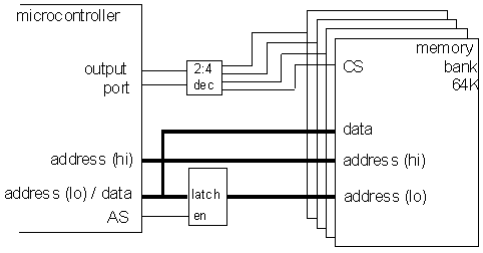


Electronic Phonebook - Automated Connections



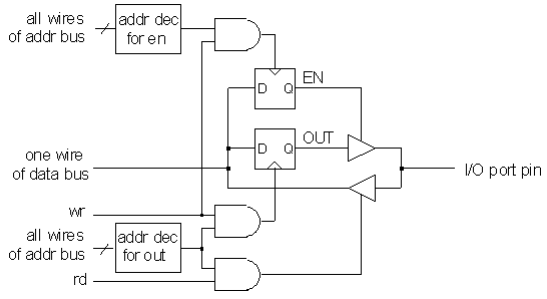
Addressing large address spaces

- Suppose we want to access 256K of RAM with an MC68HC11
- HC11 only has a 16-bit address (64K addresses)
- Solution
  - use memory banks where I/O port selects the bank
  - careful code/data placement



64-bit I/O Port

- Suppose we wanted a 64-bit I/O port
- If EN is true, then we have an output pin
- If EN is false, then we have an input pin



64-bit I/O Port Software

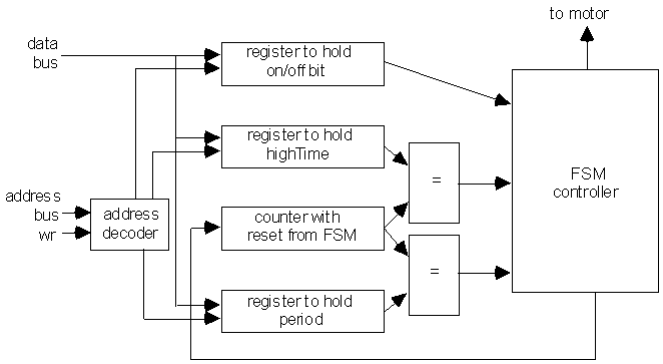
- We need 8 8-bit registers to store/write the 64 bits
- Select the EN addresses to be $0000 to $0007
- Select OUT addresses to be $0010 to $0017
- Read 15th bit
  - load value at address $0011 (2nd set of OUT regs)
  - logical AND accumulator with 0x80
  - MSB of acc is 15th bit
- Write the 47th bit
  - read OUT register at $0015
  - set MSB to desired value
  - store ACC in $0015
  - load EN register at $0005
  - set bit to output
  - store value back to $0005

Design considerations for adding parallel units

- Single processor
  - over-burdened - not enough remaining cycles to compute
  - raw performance - process just not powerful enough
  - timing/complexity - communication/computation difficult in sw
- System complexity
  - increase: more chips usually means more complicated/expensive
  - decrease: abstractions simplify design
- Power
  - increase
    - more chips/wires possibly more power
  - decrease
    - high performance chip vs. many low performance
    - "sleep" non-active components

External PWM Unit

- Design a system to control many digital motors (not enough output compare pins)
- Solution: design a PWM unit



External PWM FSM Controller

```
if (onOff == OFF)
    nextState = MotorLow
    reset counter
else if (period NOT Expired)
    nextState = MotorLow
else if (period Expired)
    nextState = MotorHigh
    reset counter
```

Motor Low State

```
if (onOff == OFF)
    nextState = MotorLow
else if (highTime Expired)
    nextState = MotorLow
else if (highTime NOT Expired)
    nextState = MotorHigh
```

Motor High State

External PWM Software

```
// in initialization code
Write off to onOff register

// do some stuff

// set up PWM
Repeat for each motor
      Write highTime and period registers

// turn motors on
Repeat for each motor
      Write on to the onOFF register

// more stuff
```