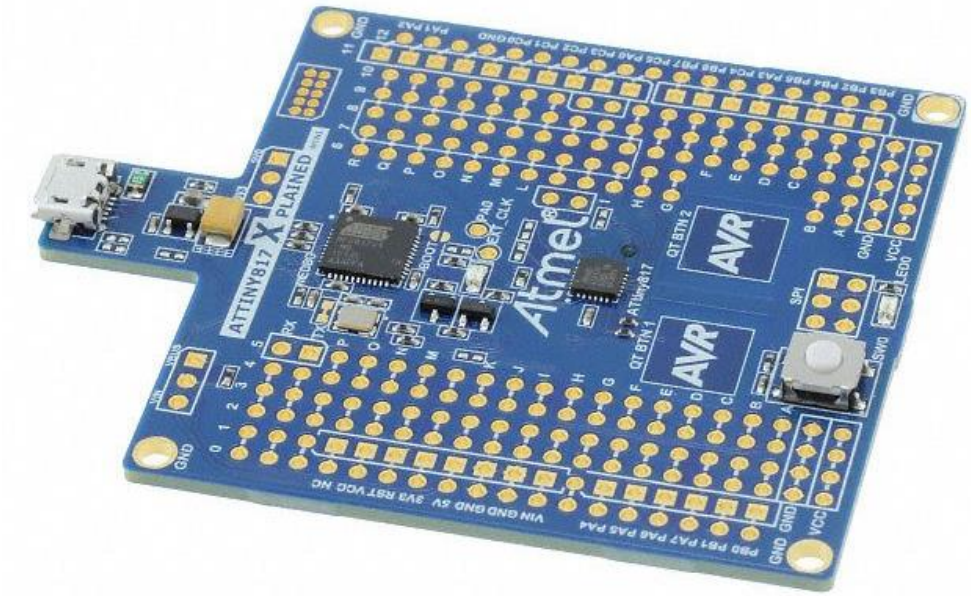
THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING



COMPSYS201 - 2018

Fundamentals of Computer Engineering

COURSEBOOK – Vol. 2

Part-2 Microcontroller (uC) based embedded systems (ES) Lecture material

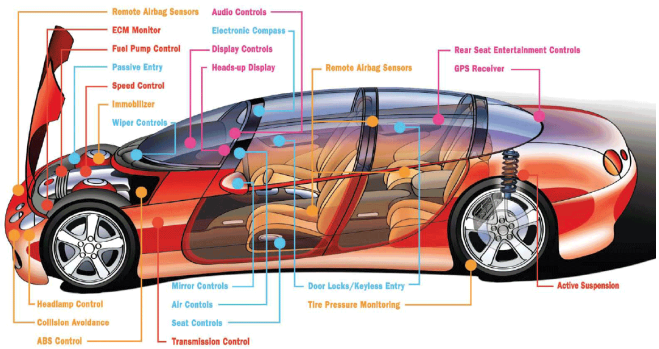
Contents

2.1.Introduction

Embedded systems are at the centre of the exponential increase of smart devices and systems in the world around us. ES's are increasingly becoming a feature of this period of history such that it is now known as 'Industry 4.0' or IoT - the internet of things. Common ES's have at their heart microcontrollers (other digital controllers include FPGAs, ASICs, microprocessors,…). ES's are ubiquitous (everywhere - but so much so that we are generally unaware of them) they automate, balance, maximise, minimise, regulate and control our everyday lives better than we can ourselves! A washing machine measures the dirtiness and weight of clothes and determines for us the amount of water and the length of the washing cycle, the output emissions of a car are monitored and the ES controls the ratio of fuel, air and exhaust gas injected into the engine making decisions on our behalf about fuel consumption, efficiency and pollution. Electricity controllers link to payment systems and switch our power on and off as we pay (or not) our power bills. Very soon out toilet roll holders will tweet us when the roll has run out!

This part of the course is an introduction to the programming of uC's and how as part of ES's they sense and control the environment. Because ES's come in so many varied forms, with so many varied features and have so many varied roles, finding a single definition is not easy. Instead a focus on the core characteristics of ES is needed: their automatic and reactive operation in real-time, the close interrelationship between hardware and software and the unique tools that engineers rely on to develop them. ES's are vital to all three specialisations within the department because you will undoubtedly come across the ubiquitous ES at several different stages in your careers and you must be aware of the important constraints that they have and their 'fit' within the purposes of the systems they are embedded within.

www.asbbank.co.nz



4. Today's vehicles feature many more MCUs, which control numerous functions. *(Courtesy of Microchip Technology)*

2.2.Learning outcomes:

Student learning goal: develop a viable mental model of a microcontroller based Embedded System
To achieve this goal you will need to begin to:

LO 1: understand the interrelatedness of hardware and software in ES's

- explain how program commands work that link hardware and software
- with reference to a datasheet, write code that manipulates/checks single bits in registers

LO 2: understand the ES as an automaton

- explain the common C code variants used to make an ES into an automaton
- discuss ES's with regard to unattended consistent and reliable operation
- explain how an ES can operate with the illusion of concurrency
- explain state - the interrelatedness of the past, present and future conditions of an ES

LO 3: understand the ES as reactive and responsive to its environment

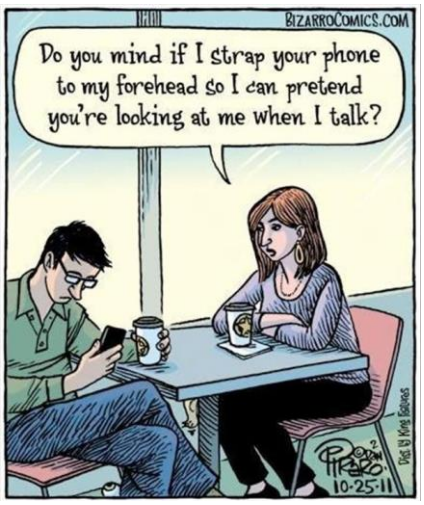
- explain polling and its timing limitations
- explain contact bounce issues with physical switches and software de-bounce code
- explain how internal uC timers are used to make an ES responsive and reactive
- setup a uC timer in software to make an uC responsive to an environmental event
- explain how uC external interrupts are used to make an ES reactive
- setup external uC interrupts in software
- describe the significant characteristics of an internal ADC
- describe the use of a sensor in a voltage divider circuit
- identify issues relating to dynamic range and quantization with analog sensors
- explain issues of an ES's responsiveness with regard to polling, blocking and interrupts.
- describe a software state machine in terms of states, actions, conditions and transitions
- explain deterministic and non-deterministic behaviour
- turn a state machine model into C code, using while / switch

LO 4: understand the importance of transparent software practices for an ES

- apply a formal approach to writing program code through use of a template, macros and naming conventions for variables and functions that makes code transparent
- write comments in code that explain function rather than textual features

LO 5: understand how 'fitness-for-purpose' guides uC choice for an ES

- list specifications of an uC that would inform choosing a uC for an ES
- describe the different memories of the microcontroller and their uses
- explain uses of different variable types and their limitations
- describe implications of over/under flow of variables
- interface an LED to a UC, taking into account the DC characteristics of each



LO 6: understand the various roles of an IDE

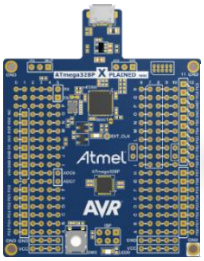
- describe the pre-processor, compiler and programming processes
- compile and upload compiled code to a microcontroller and test it

L O7: develop as an engineering student by responding positively to new, challenging and complex (multidimensional) tasks

2.3.Atmel and the Xplained mini

Atmel is one of many brands of microcontrollers. Other brands include: Texas, Cypress, Renasas, ST, TI, National, NXP, Freescale…

- Atmel make several different ranges of uC's - AVR8, UC3(32 bit) and ARM
- The 8 bit AVR's includes several sub types: ATtiny, ATmega, ATxmega, AT90
- ATMEL have about _____ of the world market share of 8 bit uC's –they sell in excess of _____ per year at an average cost of around _____ each
- Each sub type has many different variants e.g. ATtiny45, ATmega2560, _____
- When learning about uC's you will often start with a development board or kit.
- The Atmel Xplained Mini development board has an ATmega328P – the board costs < _____
- But you can get an equivalent Arduino Uno Board from aliexpress.com for about _____



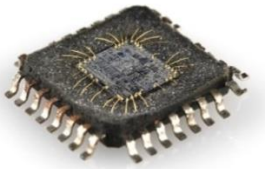
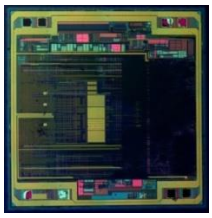
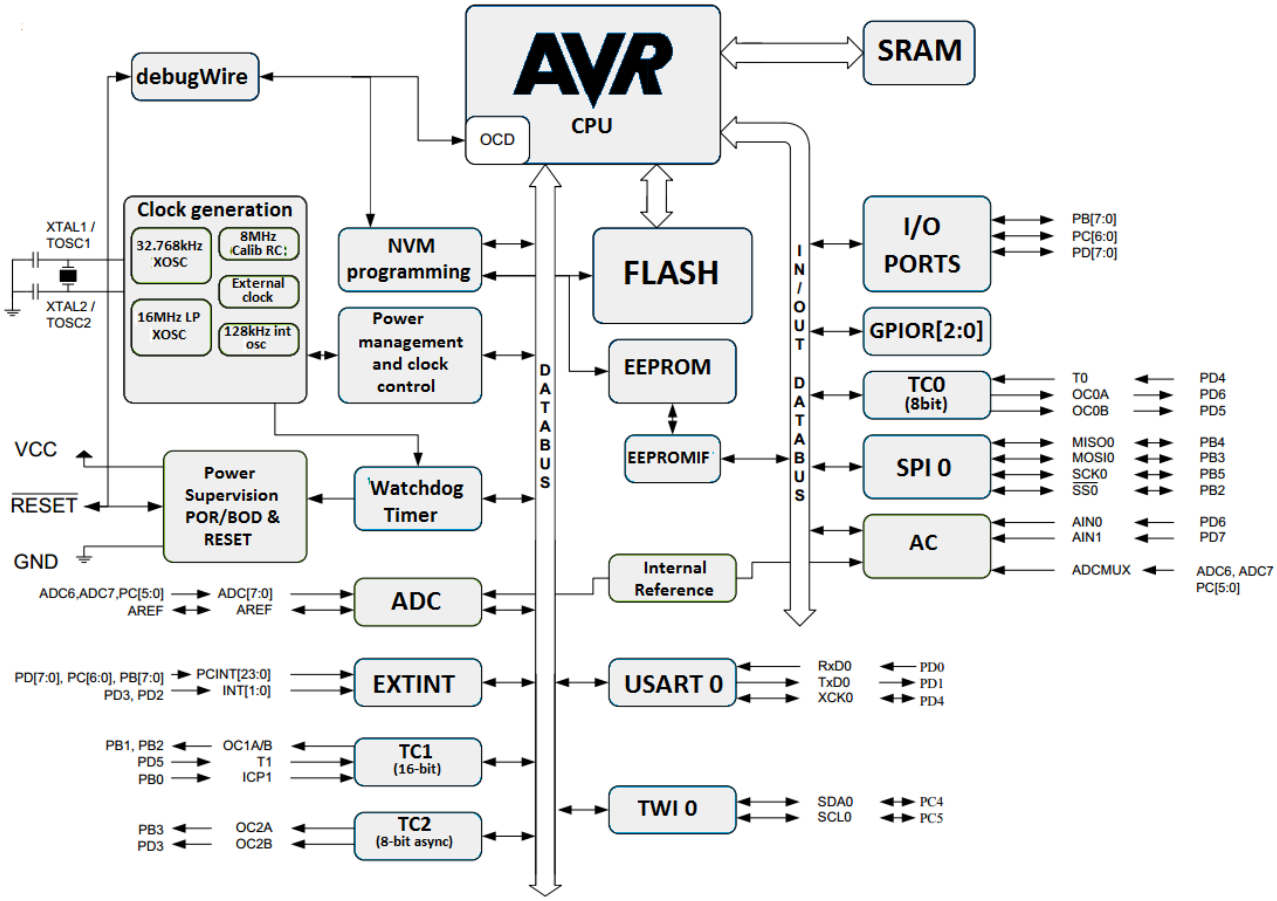
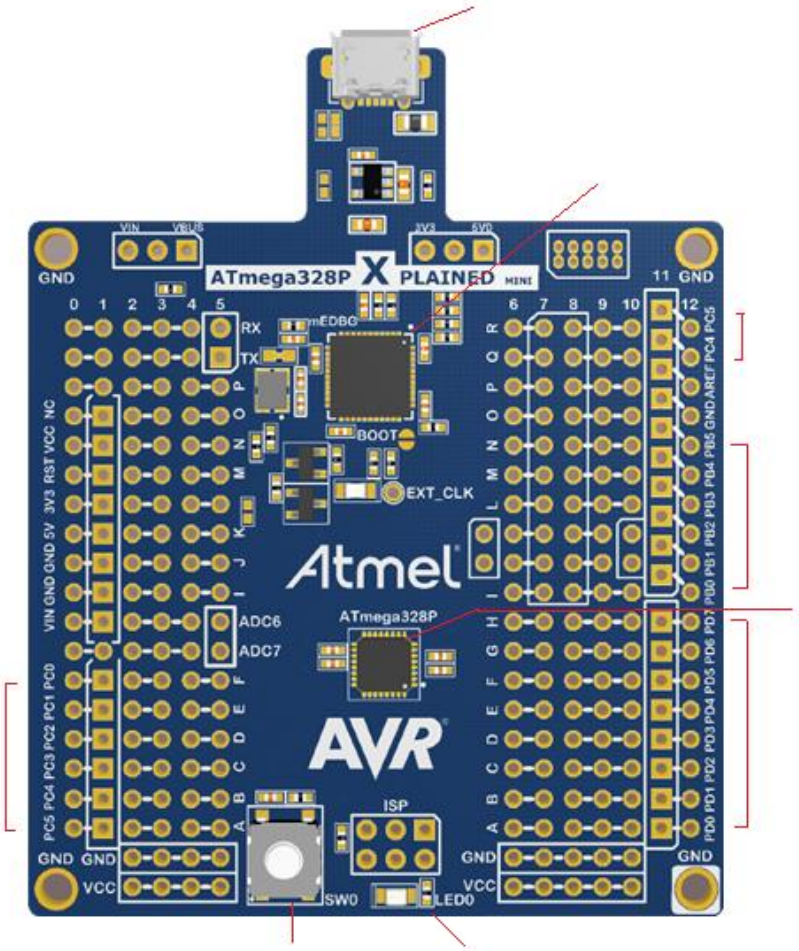




Figure 4-1. Block Diagram



The ATmega328P Xplained Mini (development board)



List specifications of a uC that would inform choosing a uC for an ES.

PORTS – collections of I/O (input/output) pins – places we can connect:

I/P devices: _____

O/P devices: _____

ATMega328P has 3 ports: PORT___ PORT___ PORT___

Ports each have _____ and each pin on a port is named _____ Because of the number of physical pins on the ATmega328P package and the way the Xplained mini board is made we do not have access to all 8 pins on each port

- e.g. PORT___ has: ____ ____ ____ ____ ____ ____ ____ ____
- e.g. PORT___ has: ____ ____ ____ ____ ____ ____ ____ ____
- e.g. PORT___ has: ____ ____ ____ ____ ____ ____ ____ ____

We can connect either input or output devices to a pin to sense or control the real world. An initial aspect of using a pin is to set up which direction it will be – input – to read an external input device or as an output to control an external output device.

Datasheet: http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf

What is the difference between an FPGA and a uC (microcontroller)?

What is the difference between VHDL and C?

An FPGA is a collection of logic blocks (circuits) within a single package. VHDL is not written into an FPGA, VHDL is the language used to configure the arrangement of signals between the logic blocks and which and where I/O pins connect to the logic blocks; in this way VHDL tells an FPGA what it will become**.** The name Field Programmable Gate Array describes its functionality quite well it is array of gates (logic blocks) which is programmable (changeable) in the field (away from the factory). Importantly it is possible to program the arrangement of logic within the FPGA so that different functions can operate concurrently and so an FPGA is able to do multiple tasks independently and in parallel.

A uC is a single computer condensed into one chip (integrated circuit), it is a highly complex logic circuit whose hardware is unchangeable. The programs for a uC are sequences of instructions written in C (or other programming languages); these are compiled into machine code (and stored in .hex files). Machine code is written into the microcontroller's Flash (program memory). In this way programs tell the fixed hardware of a uC what it will do. Importantly the different parts of a C program running in a uC operate sequentially**.**

2.4.C review

Take time to review the information below from the course notes from ENGGEN 131

page 5: the difference between compiled and interpreted
page 6: code blocks, array indexes and logical operators, the last paragraph
page 9: the diagram – we will compare this to the process used in ES development
page 10 preprocessor (#include, #define), compiler
page:11 the main() function
page 18: overflow
page 20: the 'warning'
page 21: Data
page 24: unary operators
page 25: special assignment and operator precedence
page 31: basic arrays
page 32: booleans
page 33/34: the if selection statement
page 36: loops, while(){}, for(){}, do{}while()
page 52: functions
page 53: void
page 54: return
page 55: calling a function
page 58: function prototypes
page 60: variable scope
page 63: parameters behave like local variables

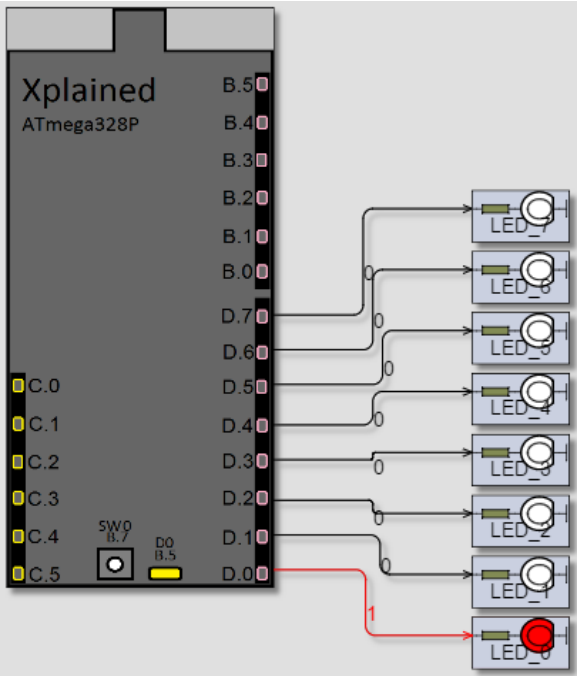
2.5. LO: Understand interrelatedness of hardware and software in ES's

A core ES characteristic is how ES developers work fluidly between hardware and software.

describe how program commands work that link hardware and software

```
/***** Project Header *****/
// Project Name: Cylon 8
/***** Hardware defines *****/
//make sure this matches your oscillator
#define F_CPU 16000000//crystal
/***** Includes *****/
#include <avr/io.h> //_____________________________________________
#include <util/delay.h>
/***** User macros *****/
#define TIME_DELAY 100 //_____________________________________
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    // Initially make all micro pins outputs
    DDRB = 0xff; //setup all pins as outputs
    DDRC = 0xff;
    DDRD = 0xff;
    /***** Loop code *****/
    while (1) {
        PORTD = 0b00000001;
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000011;
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000111;
        _delay_ms(TIME_DELAY);
        PORTD = 0b00001110;
        _delay_ms(TIME_DELAY);
        PORTD = 0b00011100;
        _delay_ms(TIME_DELAY);
        PORTD = 0b00111000;
        _delay_ms(TIME_DELAY);
        PORTD = 0b01110000;
        _delay_ms(TIME_DELAY);
        PORTD = 0b11100000;
```

```
    } //end while(1)
} //end of main
```



Software that controls hardware

```
DDRD = 0xff;
```

make all 8 pins of portD outputs

```
PORTD = 0b00000001;
```

force bit 0 high ***AND*** all other bits low of register PORTD.

```
_delay_ms(TIME_DELAY);
```

uC's carry out instructions at rates much faster than environments can recognise or respond to, programs can make use of simple delays to make the ES meet the demands of the environment. This code blocks or pauses the execution of the program briefly so that the people have time to see the LED pattern.

```
_delay_ms(TIME_DELAY);
```

If these lines of code were removed what would the Cylon look like?

2.6.LO Understand the ES as an automaton

A core ES understanding is seeing the ES as an automaton - an automatic machine that carries out instructions without human intervention. An ES begins running on power up, follows a set of predetermined instructions without human intervention and will do so until the power is removed. In contrast the software run on a PC is transactional in nature, it requires a human to decide which path the software will take.

The way an ES becomes an automaton is via this simple program operation

Explain the common C code used to make an ES into an automaton

```
int main(void) { //c programs begin executing at the main function

    while (1) {
        …
        …
        …
    }

}               //C programs finish at end of the main function
```

2.7. LO: Understand the importance of transparent software practices

Apply a formal approach to approach to writing program code

```
/***** Project Header *****/
// Project Name: Cylon 14
// Author: Bill
// Date: 21/02/2014  11:51
// Version: 1
// This code…

/***** Hardware defines *****/
//make sure this matches your oscillator setting
#define F_CPU 16000000//crystal
/***** Includes *****/
#include <avr/io.h>
#include <util/delay.h>
/***** User macros *****/
#define TIME_DELAY 100 //
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    // Initially make all micro pins outputs
    DDRB = 0xff; //setup all pins as output
    DDRC = 0xff;
    DDRD = 0xff;
    /***** Loop code *****/
    while (1) {
        PORTD = 0b00000001; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000011; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000111; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00001110; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00011100;
        _delay_ms(TIME_DELAY);
        PORTD = 0b00111000;
        _delay_ms(TIME_DELAY);
        PORTD = 0b01110000;
        _delay_ms(TIME_DELAY);
        PORTD = 0b11100000;
        …

    } //end while(1)
} //end of main
```

Key structural elements in the code

- **Template**
  - Structures your code
  - into different sections
- **Title block**
  - Name
  - Date
  - Author
  - Version
- **Macros**
  - #define TIME_DELAY 100
  - no magic numbers
  - CAPITALS
- **Comments**
  - describe function not syntax
- **Layout**
  - indentation
    -
  - syntax highlighting
    -
- **Why are these points so critical?**
  - engineers are not haphazard in what they do, professionalism in documentation is essential , most organisations will have formal methods of structuring their planning and documentation.

2.8. State

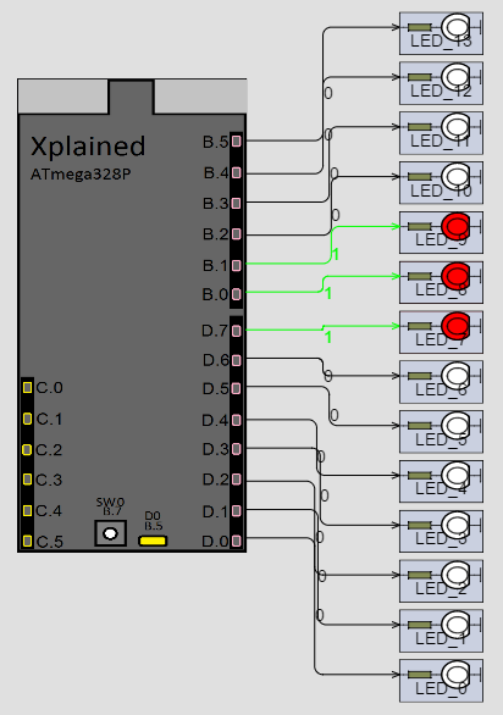
Explain state - the interrelatedness of the past, present and future conditions of an ES

A core understanding of the ES as an automaton is that it is a sequential device; instructions are carried out one after another in a sequence. This has a significant side effect; the concept of state. The future state (condition) of the system is entirely dependent upon the current state and what has happened previously. In contrast an application on a PC does not have state, as what happens at any point in time in a PC is not dependent upon what happened previously and does not impact on what will happen in the future.

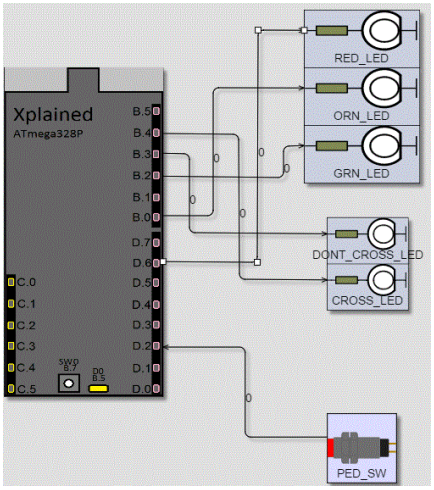
In this exercise the concept of state can be seen when…

The Cylon14

```
while (1) {
    PORTD = 0b00000001; //turn on the first LED
    _delay_ms(TIME_DELAY);
    PORTD = 0b00000011; //turn on TWO LEDs
    _delay_ms(TIME_DELAY);
    PORTD = 0b00000111; //turn on THREE LEDs
    _delay_ms(TIME_DELAY);
    PORTD = 0b00001110; //shift pattern by one
    _delay_ms(TIME_DELAY);
    PORTD = 0b00011100;
    _delay_ms(TIME_DELAY);
    PORTD = 0b00111000;
    _delay_ms(TIME_DELAY);
    PORTD = 0b01110000;
    _delay_ms(TIME_DELAY);
    PORTD = 0b11100000;
```



2.9.Describe the ES as reactive and responsive to its environment



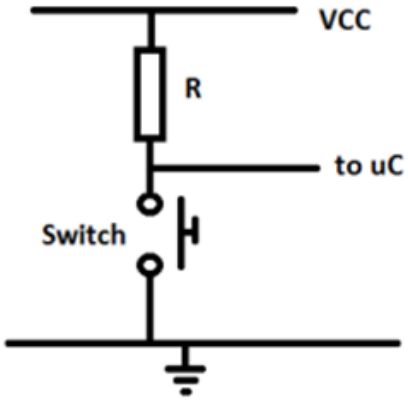

```
/***** Hardware macros *****/
//Hardware macros for outputs
1.   #define SET_RED_LED   PORTD |= (1<<PD6)
2.   #define CLR_RED_LED   PORTD &= ~(1<<PD6)
3.   #define SET_ORN_LED   PORTB |= (1<<PB0)
4.   #define CLR_ORN_LED   PORTB &= ~(1<<PB0)
5.   #define SET_GRN_LED   PORTB |= (1<<PB2)
6.   #define CLR_GRN_LED   PORTB &= ~(1<<PB2)
7.   #define SET_DONT_CROSS_LED   PORTB |= (1<<PB3)
8.   #define CLR_DONT_CROSS_LED   PORTB &= ~(1<<PB3)
9.   #define SET_CROSS_LED   PORTB |= (1<<PB4)
10.  #define CLR_CROSS_LED   PORTB &= ~(1<<PB4)
11.  //Hardware macros for inputs
12.  #define PB_SW_1_IS_LOW   ~PIND & (1<<PD2)
13.  /***** User macros *****/
14.  #define ORANGE_DELAY 3000
15.  #define RED_DELAY 500
16.  #define PED_DELAY 8500
17.  /***** Main function *****/
18.  int main(void) {
19.      // initially make all micro pins outputs
20.      DDRB = 0xff; //set as outputs
21.      DDRC = 0xff; //set as outputs
22.      DDRD = 0xff; //set as outputs
23.      // make these pins inputs
24.      DDRD &= ~(1 << PD2);
25.      /***** Run once code goes here *****/
26.      //setup state of all outputs
27.      //do not assume they are off
28.      SET_GRN_LED;
29.      CLR_ORN_LED;
30.      CLR_RED_LED;
31.      SET_DONT_CROSS_LED;
32.      CLR_CROSS_LED;
33.      /***** Loop code *****/
34.      while (1) {
35.          if(PB_SW_1_IS_LOW){
36.              SET_ORN_LED;
37.              CLR_GRN_LED;
38.              _delay_ms(ORANGE_DELAY);
39.              SET_RED_LED;
40.              CLR_ORN_LED;
41.              _delay_ms(RED_DELAY);
42.              SET_CROSS_LED;
43.              CLR_DONT_CROSS_LED;
44.              _delay_ms(PED_DELAY);
45.              CLR_CROSS_LED;
46.              SET_DONT_CROSS_LED;
47.              SET_GRN_LED;
48.              CLR_RED_LED;
49.          }
50.      } //end while(1)
51.  }
```

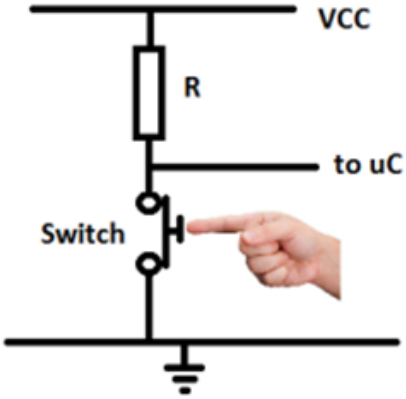
Which **three** lines of code are part of making the ES responsive to its environment?

The pullup resistor is an essential part of an input circuit. When the switch is:

- **not pressed**,
- **pressed**,

The uC can then be programmed to react differently to whether the switch is open or closed





Without a pullup resistor when the switch is open the input pin would be _____

Checking to see if a pin is low

```
#define SWITCH_IS_LOW          ~PIND & (1<<PD2)

if ( SWITCH_IS_LOW ){

}
```

OR checking to see if a pin is high

```
#define SWITCH_IS_HIGH         PIND & (1<<D2)

if ( SWITCH_IS_HIGH ){

}
```

Draw a pulldown circuit

Apply a formal approach to writing program code

At the core of professionalism in programming is **transparency**. Code must be readabel, logicamal, understoodable and maintenanceable. In programs indentation and macros are used to structure code to make it transparable!! There is no place for obfuscation in programming, do not be an obfuscator!

In this code some of the lines use macros, some don't, which line of code is wrong?

```
//pedestrian crossing program
/***** Hardware macros *****/
//Hardware macros for outputs
#define SET_RED_LED   PORTD |= (1<<PD6)
#define CLR_RED_LED   PORTD &= ~(1<<PD6)
#define SET_ORN_LED   PORTB |= (1<<PB0)
#define CLR_ORN_LED   PORTB &= ~(1<<PB0)
…
…
1.  while (1) {
2.  if(PB_SW_1_IS_LOW){
3.  SET_ORN_LED;
4.  PORTB &= ~(1<<PB2);
5.  _delay_ms(3000);
6.  SET_RED_LED;
7.  PORTB &= ~(1<<PB0);
8.  _delay_ms(500);
9.  PORTB |= (1<<PB3);;
10. CLR_DONT_CROSS_LED;
11. _delay_ms(8500);
12. PORTB &= ~(1<<PB4);
13. PORTB |= (1<<PB3);
14. PORTB |= (1<<PB2);
15. CLR_RED_LED;
16. }
17. }
```

Discuss why transparency might (or might not) be more important for ES software than a PC application

2.10. Die Program (dice - plural, die - singular)

Apply a formal approach to writing program code

```
/***** Prototypes for Functions *****/
void one();
void two();
void three();
void four();
void five();
void six();
/***** Declare & initialise global variables *****/
uint8_t count=0;
/***** Main function *****/
int main(void) {
    /***** Loop code *****/
    while (1) {
        count++;
       if (TACT_SW_1_IS_LOW) {
            switch (count) {
                case 1:
                    one();
                    break;
                case 2:
                    two();
                    break;
                …
                …
            }
            //delay a bit
        }
    } //end while(1)
} //end of main
/***** Functions *****/
void one(){//turn on centre LED4
    PORTB=0b
    PORTD=0b
}
void two(){
    PORTB=0b
    PORTD=0b
}
void three(){
    PORTB=0b
    PORTD=0b
}
void four(){
    PORTB=0b
    PORTD=0b
}
void five(){
    PORTB=0b
    PORTD=0b
}
void six(){
    PORTB=0b
    PORTD=0b
}
```

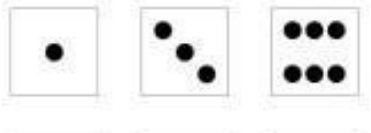
Making code transparent with functions

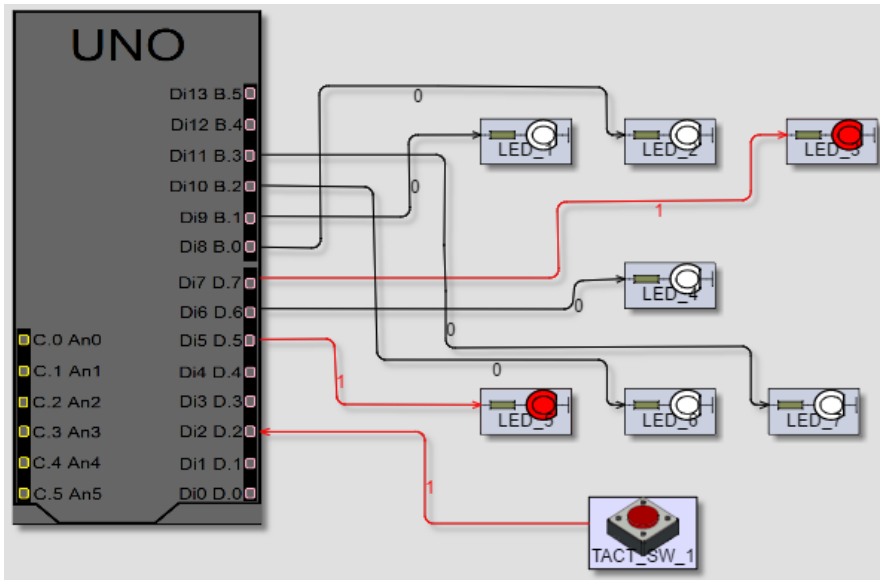
Functions have a single purpose

Functions have descriptive names

Functions fit in 1 page

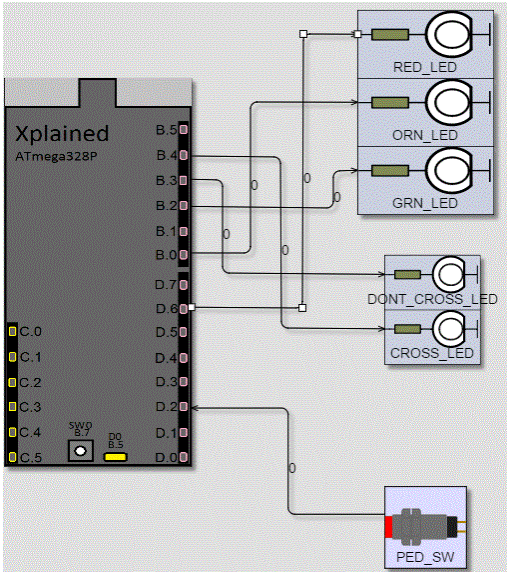
Use function prototypes





The use of one(), two() etc. as function names makes sense in this trivial program, however if the code become more complex, these might need more explanatory names. What would you name them to make them as fully transparent as possible?

2.11. Macros & bit masking



It is not common to control all pins of a port at one time, we would normally set and clear pins individually.

In the pedestrian lights controller in this diagram we will want to control different lights at different times of the sequence so we must learn how to control each pin of the port on its own. To control the GRN_LED we would follow the process of:

- reading all 8 bits of PORTB,
- change bit 2 (either high or low) while keeping all the other 7 bits unchanged
- writing the value back into PORTB.

We do this using an understanding of Boolean identities

1. X or 1 = 1 we can force a bit high
2. X or 0 = X we can keep a bit unchanged
3. X and 0 =0 we can force a bit low
4. X and 1 = X we can keep a bit from changing

To turn the LED on we need to make bit 2 '1' (high) and keep the other bits unchanged – identities 1 and 2 will help us. **PORTB = PORTB | 0b00000100; //turn GRN_LED on**

Here we are forcing bit 2 to be high, it does not matter what value it has to start with, the effect will be PORTB output B.2 will go high and the LED will come on. The other bits are 'OR' with 0 so they remain unchanged

To turn the LED off we need to make bit 2 '0' (low) and keep the other bits unchanged – identities 3 and 4 will help us. **PORTB = PORTB & 0b11111011; //turn GRN_LED off**

Here we are forcing bit 2 low, it does not matter what value it was to start with the effect will be that PORTB output B.2 will go low turning off the LED. The other bits are 'ANDED' with 1 so remain unchanged.

2.11.1. Macros

While the statements above work perfectly, for code transparency reasons we must not go 'littering' our code with statements like those above it makes it totally unreadable and awful to debug. We learn early on to write macros at the top of your program that keep all the hardware control in one place.

- #define SET_GRN_LED PORTB|=(1<<B2) //macro to turn GRN_LED on
- #define CLR_GRN_LED PORTB&=~(1<<B2) //macro to turn GRN_LED on

Throughout our code we use the commands SET_GRN_LED /CLR_GRN_LED that way if we ever moved the GRN_LED to another pin we only need to change it in one place in the code and that is easily identifiable as they are all in one place.

2.11.2. **unpacking macros**

The Anatomy of an output macro

- pre-processor replaces the macro
- read the port register
- change only the required bit(s)
- write the new value to the port register

Output macros

```
SET_RED_LED;
```

```
CLR_RED_LED;
```

What would this command do?

```
PORTD ^= (1<<PD6);
```

The anatomy of an input decision?

- pre-processor replaces the macro
- read the pin register
- isolate the required bit(s)
- write the new value to the port register

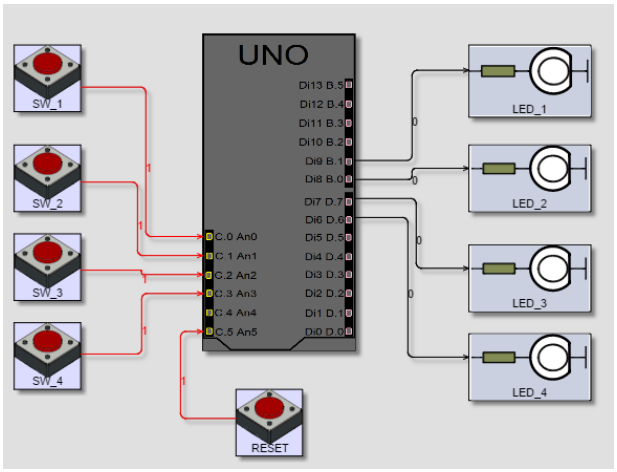
```
if ( PED_SW_IS_LOW ) {      }
```

```
if (PED_SW_IS_HIGH) {       }
```

2.12. Polling

Describe how an ES operates with the illusion of concurrency

The uC often has to read multiple inputs, it is however a sequential device (not like an FPGA) so can only do one thing at a time. This means that the program in a uC has to **poll** or regularly check each input. Polling however must be at a rate that is fast enough not to miss any input event. When polling inputs the rate of polling relates to how fast the environment changes input states. If we poll switch inputs faster than the environment can change them,then no environmental change will ever be missed and we will have the illusion of concurrency. In this example polling is demonstrated by sequentially checking four contestants switches. The illusion of concurency is demonstrated by the rate of polling being so high that it appears no contestant has an unfair advantage over another. Note polling refers not just to reading switches but to the concept of regularly checking something.



How does a quiz game controller function? Write down here what the uC should do. We call this an algorithm, sometimes drawn in diagram form or written as plain language or written in pseudo-code. Because there is an interrelationship between the contestant buttons we need to think about logic that will manage the interaction correctly.

```
while (1) {

}
```

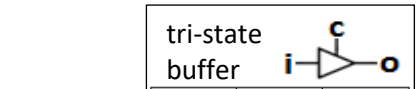
2.12.1. How can a single uC pin be both an input and an output?

Describe how program commands work that link hardware and software

Each pin has three single bit registers or latches associated with it,

- PORTx.n e.g.
- PINx.n e.g.
- DDRx.n e.g.

tri-state buffer

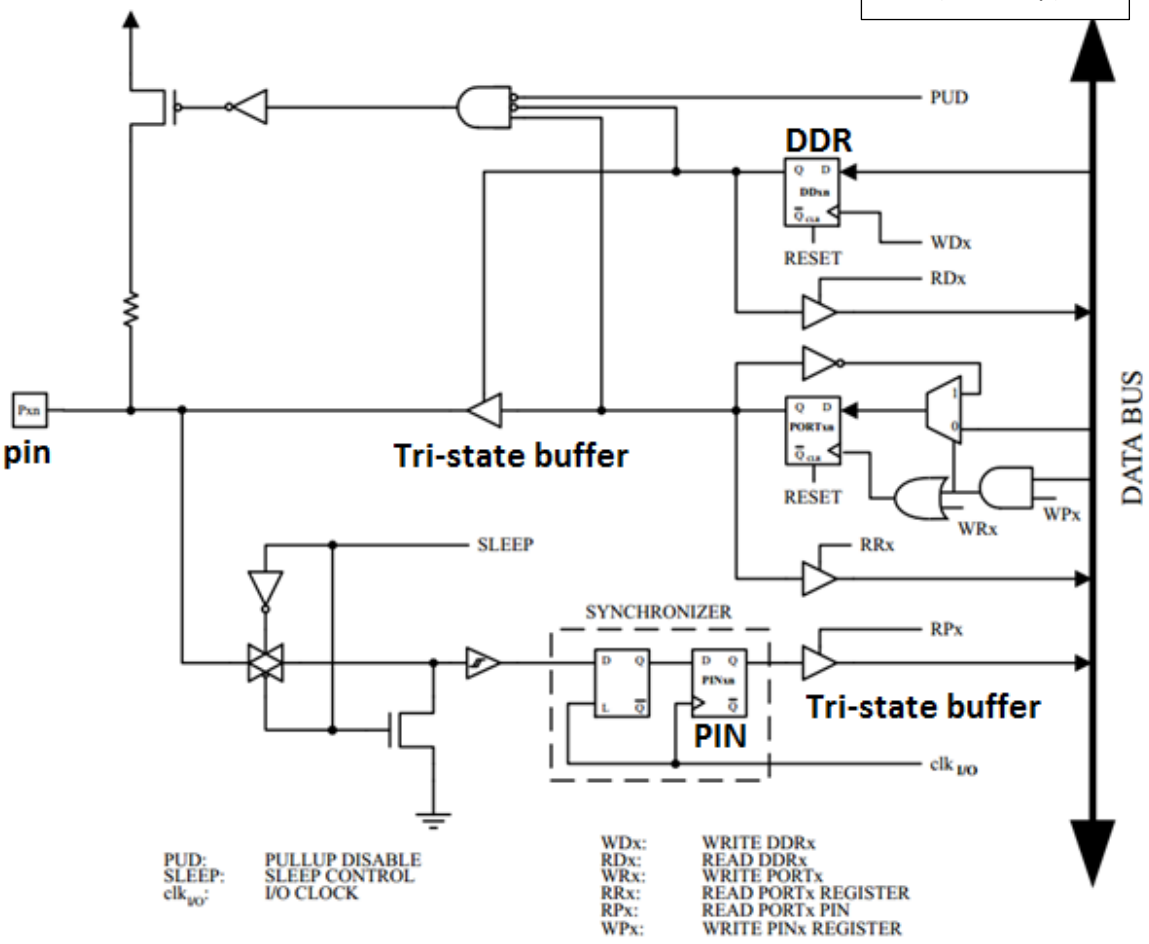


C	In	Out
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

Hi-Z (like a tap)

How a µC pin works as an INPUT

Figure 18-2. General Digital I/O(1)

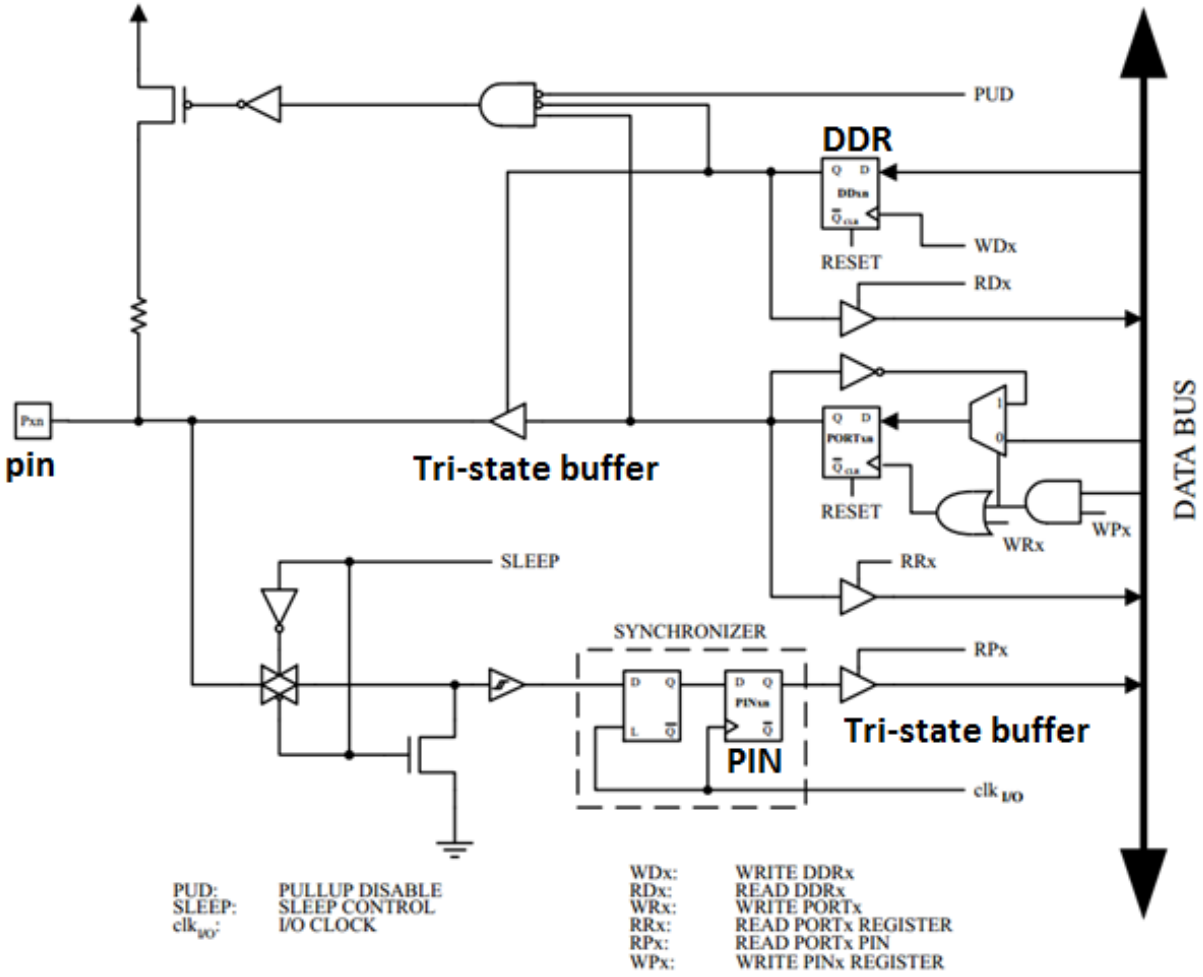


The initial power-up state of all pins is as inputs- the DDR (data direction register) output is 0. This means the tri-state buffer has a high Z output (it's like a tap is off) and signals from the PORT register cannot go through to the pin. The signal (1 or 0) from the physical pin comes through to the PIN register. When the uC encounters a command to read the pin e.g if (SW_1_IS_LOW) {} or more precisely if (~PINB & (1<<PB3)) {}, a 1 is put on RPx allowing the tri-state buffer output to reflect the value of the hardware pin onto the databus.

This diagram from the datasheet represents the logic behind one pin of one port. How many times is this block repeated in the ATmega328P?

How a µC pin works as an output

Figure 18-2. General Digital I/O(1)



To make the pin an output requires the DDR to output a 1, this opens the tri-state buffer 'tap' allowing the value stored in the PORT register to go through to the physical pin. This is the effect of the command DDRA |= (1<<A5). Or we could make all 8 bits of a port outputs all at once by writing DDRA=0xFF . At any stage in our program we can drive the physical pin high or low using the commands PORTA |= (1<<A5) or PORTA &= ~(1<<A5). The 1 or 0 is stored in PORTA.5 and because the tri-state buffer is not hi-Z (tap is open) it goes through to the pin.

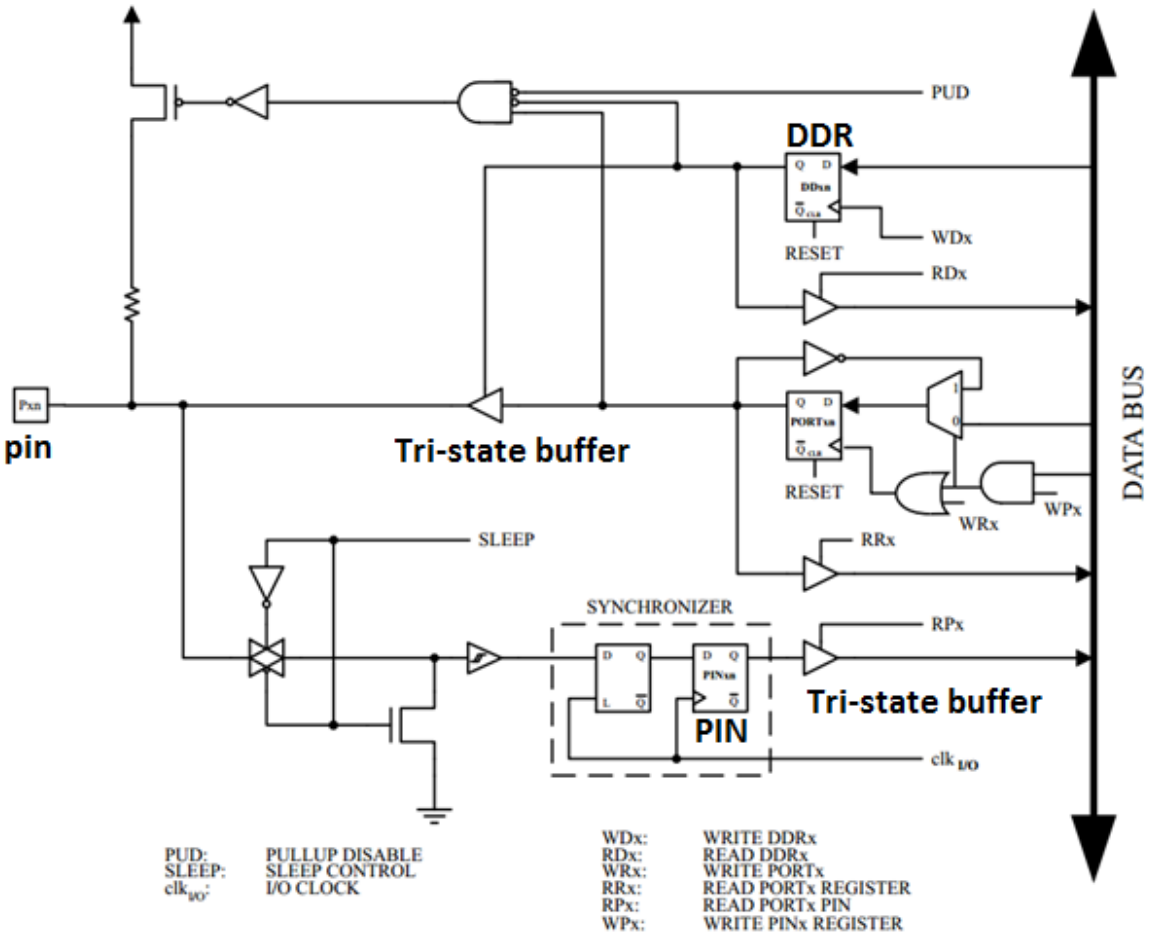
A special effect of this hardware is that the PIN register always reflects the state of the hardware pin even when the port is setup as an output. So you can check the state of an output device at any stage in your program by reading the PIN register - can you see how this could beused in the quiz game controller?

Confusing terminology alert - pin & port: When we say 'port' we are generally referring to a collection of 8 physical I/O pins not the PORT register. When we say 'D3' we are referring to a single physical pin. When we talk about reading or writing a pin, we mean a single physical pin not the PIN register. When we say 'port D4' we are referring to the physical pin and may be reading or writing it. Try to refer to the register explicitly 'PORTA register', 'PINA register', but sometimes you just have to recognise the context the term is being used in to know if it is the input register, the output register or the physical hardware pins being referred to.

2.12.2. **The internal pullup resistor**

The pullup resistor as part of ES is such a common component that the uC designers have added internal pullup resistors to every I/O pin.

Figure 18-2. General Digital I/O(1)



Configuring the internal pullup resistor.

1. The pin must be configured as an input
2. Turn on the …

There is a second reason for providing the internal pullup resistor. Floating (unconnected) pins can become troublesome for the uC's. An unconnected pin can detect stray electric charge (noise) surrounding the uC which can causes the input circuit to flip-flop between high and low. Switching like this causes unwanted power consumption.

There are at least 3 solutions to this problem:

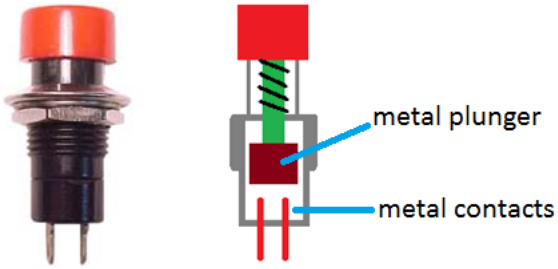
1. Make all the pins outputs to start with, unused pins will always have a defined value of...
2. After setting up all your I/O registers set the pullup resistor for any unused pins, value =…
3. Add your own external pullup resistors- why??

How does the logic controlling the internal pullup work?	

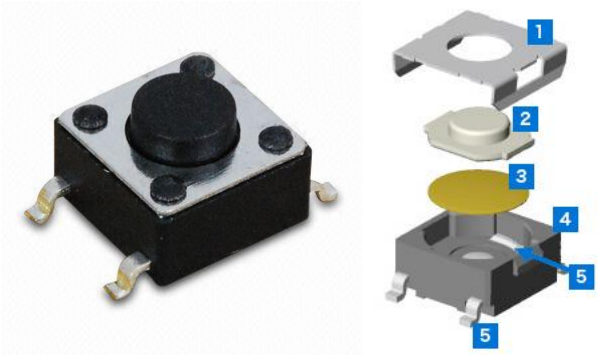
2.12.3. **Hardware topic: Switch issues**

Explain contact bounce of physical switches and simple software de-bounce code

- Anatomy of a cheap push-button switch: When the button is presses the whole metal plunger moves down closing the circuit between the two metal contacts. It can bounce a lot when this happens.



- The tact switch is much better (but still not perfect), when the button (2) is pressed the metal dome(3) flexes in a snap action touching the contacts(5); this is a rapid and firmer motion so there are few bounces.



http://www.omron.com/ecb/products/sw/special/switch/basic02-03.html

What happens when two solid objects come into contact at high speed?

We get what is called contact bounce

https://www.youtube.com/watch?v=4sA1c1WErUk

Don't try this at home!
https://www.youtube.com/watch?v=lzsTO3YMYAU

How do you think a uC responds to switch contacts bouncing?

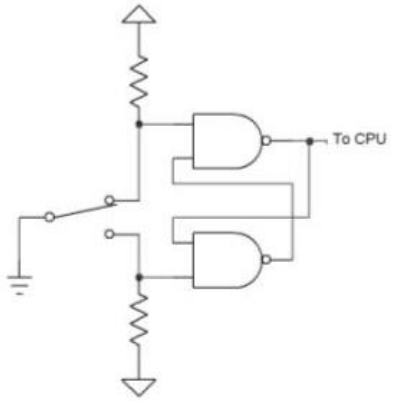
De-bouncing switches

De-bouncing is the process where we solve contact bounce issues using either hardware or software

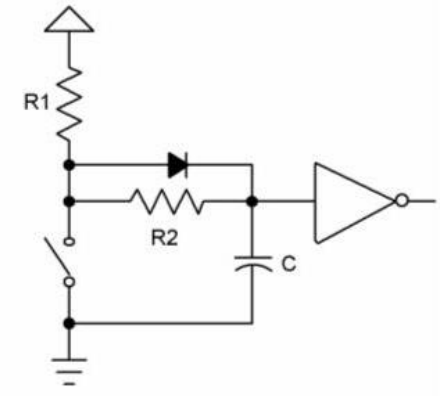
Hardware solutions

The Xplained mini board uses a simple 0.1uF capacitor across the switch

This circuit is an



A very reliable hardware solution

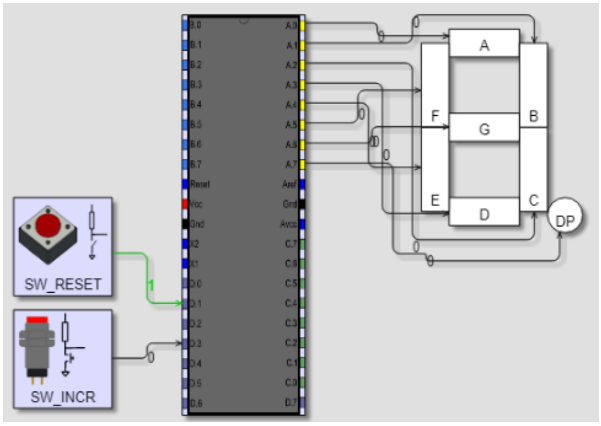


Reference: ganssle.com

Debounce software solutions

Often a short delay using _delay_ms(…) will be sufficient, sometimes more complex timing methods are used. We will look at using simple delays to solve issues in specific contexts.

2.12.4. A 'not-so-simple' debounce problem



Contact bounce can cause a switch to be detected multiple times. There is a potential issue from contact bounce in this program.

1. Initially understand the program. When the switch is pressed we increase the count and then wait for the switch to be released before continuing. It is important to understand why this is done. We only want one switch press to be detected, if we did not wait the loop would run so fast that count would increase rapidly even if we pressed and release the switch as fast as we could. 2. Now recognise the second complication of contact bounce in this program. A single press of the button can be interpreted as several presses and releases. Also when you release the switch it can be interpreted as several presses and releases as well

```
//no debounce
while (1) {
    if (SW_UP_IS_LOW) {
        count++;
        while (SW_UP_IS_HIGH){
        }
    }
    if (SW_CLEAR_IS_LOW) {
       count = 0;
    }
    display(count);
 } //end while(1)
```

```
//with debounce
#define DEBOUNCE_DELAY 10
while (1) {
    if (SW_UP_IS_LOW) {









    }
   if (SW_CLEAR_IS_LOW) {
       count = 0;
   }
   display(count);
 } //end while(1)
```

How long should the delay be?

2.13. AVR memories

Describe the different memories of the microcontroller and their uses

Knowing some details (initially size and type) of the 3 different types of memory in the uC helps us understand what some of the crucial constraints and limitations of ES's will be.

- FLASH –storing programs
  - the ATmega328P has a _____ of flash-_____
- SRAM – stores temporary data
  - the ATmega328P has _____ of SRAM(also called IRAM) -_____
- EEPROM – stores long term data
  - the ATmega 328P has a _____ EEPROM - _____

2.13.1. What type of number are you?

When writing programs for a uC, computer languages offer us the ability to constrain a number to a defined range so that we can make the most of the limited memory we have. If data varies over a small range we may be able to use just 1 byte of the RAM. Data that may vary over a larger range or data that requires more accuracy will require more than 1 byte.

Describe the different memories of the microcontroller and their uses

Memory types in C include int, unsigned int, short, long, float, etc. We will not use these names in this course; we will use names for data types that more clearly express the limitations of each type. For example in a PC the int and unsigned int types are 4 bytes in size, whereas with an AVR they are 2bytes. The limits of the types are:

- uint8_t unsigned 8 bit range
- uint16_t unsigned 16 bit
- uint32_t unsigned 32 bit
- uint64_t (1) unsigned 64 bit
- int8_t signed 8 bit
- int16_t signed 16 bit
- int32_t signed 32 bit
- int64_ t (1) signed 64 bit ()
- float/double (2) 32 bit - 1.2E-38 to 3.4E+38

(1) if you are using numbers this big then you probably shouldn't be using an 8 bit uC! The AVR is an 8 bit (single byte) device, this means that calculations or comparisons using multi-byte variables require many operations e.g. adding two 16 bit numbers requires two operations, first adding the lower byte of each number then adding the upper byte of each number.

(2) in programming AVR's a double and a floats are both the same - 4 bytes – not like in C on a PC where a double is 8 and a float is 4. Using doubles requires the use of a math library, as the AVR does not have the hardware to do floating point calculations, using the extra library takes up a lot of your flash. Division is done by repetitive subtractions – trying to do this with floating point is ugly and will really slow down the overall processing with your ES , if you need decimals it may be better to use a fixed rather than floating point number system (you did exercises with these in ElectEng101).

OOPS

Some data type errors can create BIG problems. In 1996 the Ariane rocket exploded 40 seconds after launch at a loss of $500M. Specifically a 64 bit floating point number relating to the velocity of the rocket was converted to a 16 bit signed integer. The number was larger than 32,767, and thus the conversion failed.

In 1991 28 soldiers were killed and 100 more injured when a patriot missile failed to track an incoming Scud missile. It was discovered that a 24 bit variable was used to store real time in 1/10ths of a second losing some 0.000000095 Sec of accuracy, so the patriot missile system failed to detect the Scud approaching and never launched.

Essential ES understanding: it is important to understand why it important to know the limitations of your data types.

Guidelines for type selection:

- use the smallest type possible
- use unsigned types where possible
- know the boundary cases (limits) for your data

Here is a question from a survey on ES development that I recently received.

*** 20. If the product resulting from your current project malfunctioned, what is the worst possible outcome?**

- ○ Death of Multiple People
- ○ Death of One Person
- ○ Serious Injury of One or More People
- ○ Minor Injury to One or More People
- ○ Product Recall by Company
- ○ Diminished Sales and/or Brand Reputation
- ○ Customers Return Products
- ○ Customers are Annoyed
- ○ I don't know.

This question reveals the crucial importance of non-technical (social) factors to what engineers do. If there are any doubts about the validity of asking these questions then reading about the Therac-25 should make this quite clear.

Explain over/under flow of variables

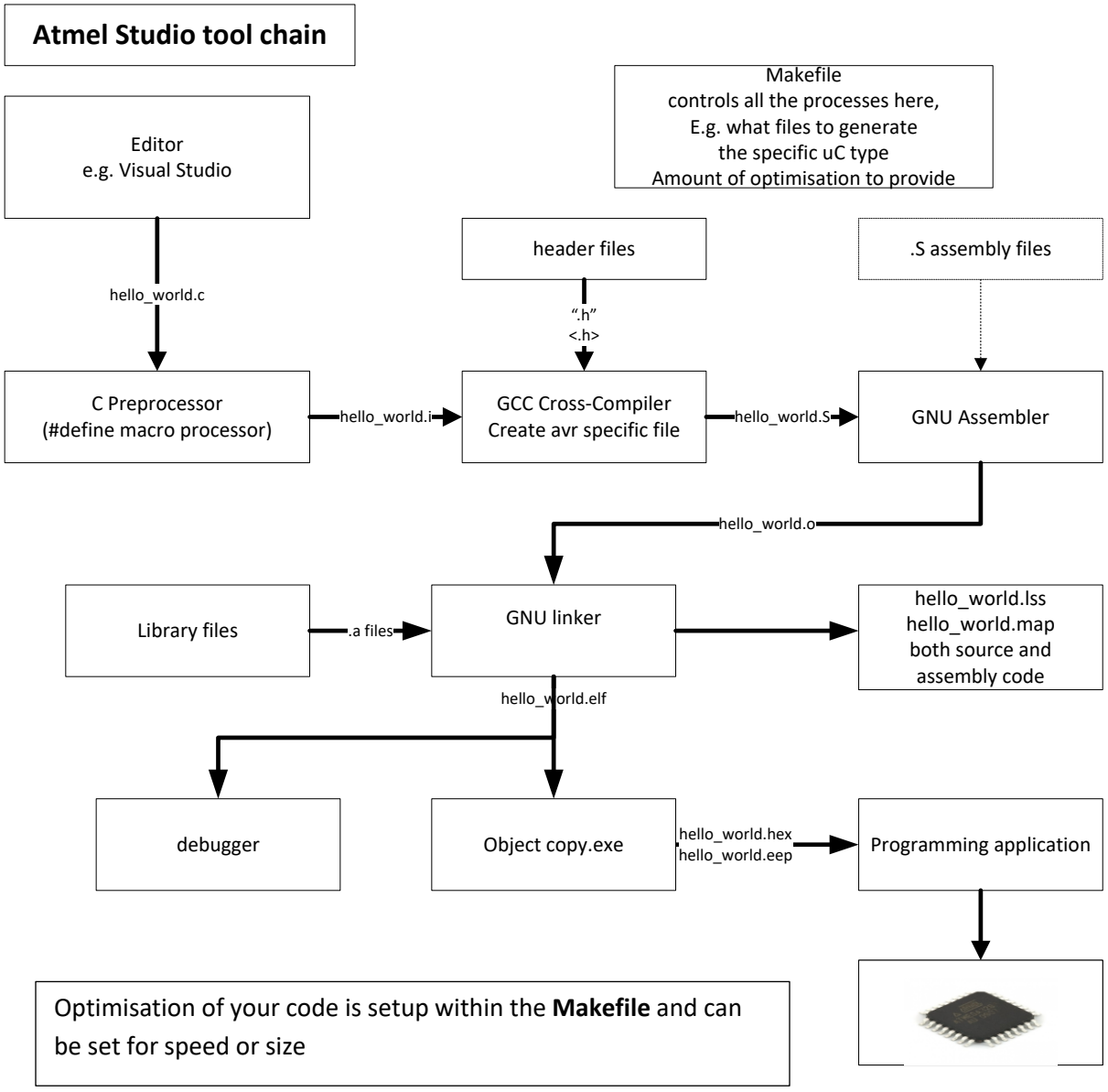
Would you use an 8 bit type to store speed for a speed camera?

> The fastest known speeding ticket was issued in May 2003 in Texas. The motorist was operating a Koenigseggs CCR, a Swedish sports car and was allegedly going **242 mph** (**389 km/hr**) in a 75 mph zone. The driver was arrested and his sport car was towed.

if you declared a variable `uint8_t km_per_hour = 0;`
and then detected the speed of the car at 389km/hr
what value would be stored in the variable?

2.13.2. **How we get from C to hex in ATMEL Studio 7.**

Describe the functions of an IDE



Alternatives to the while(1) automaton code

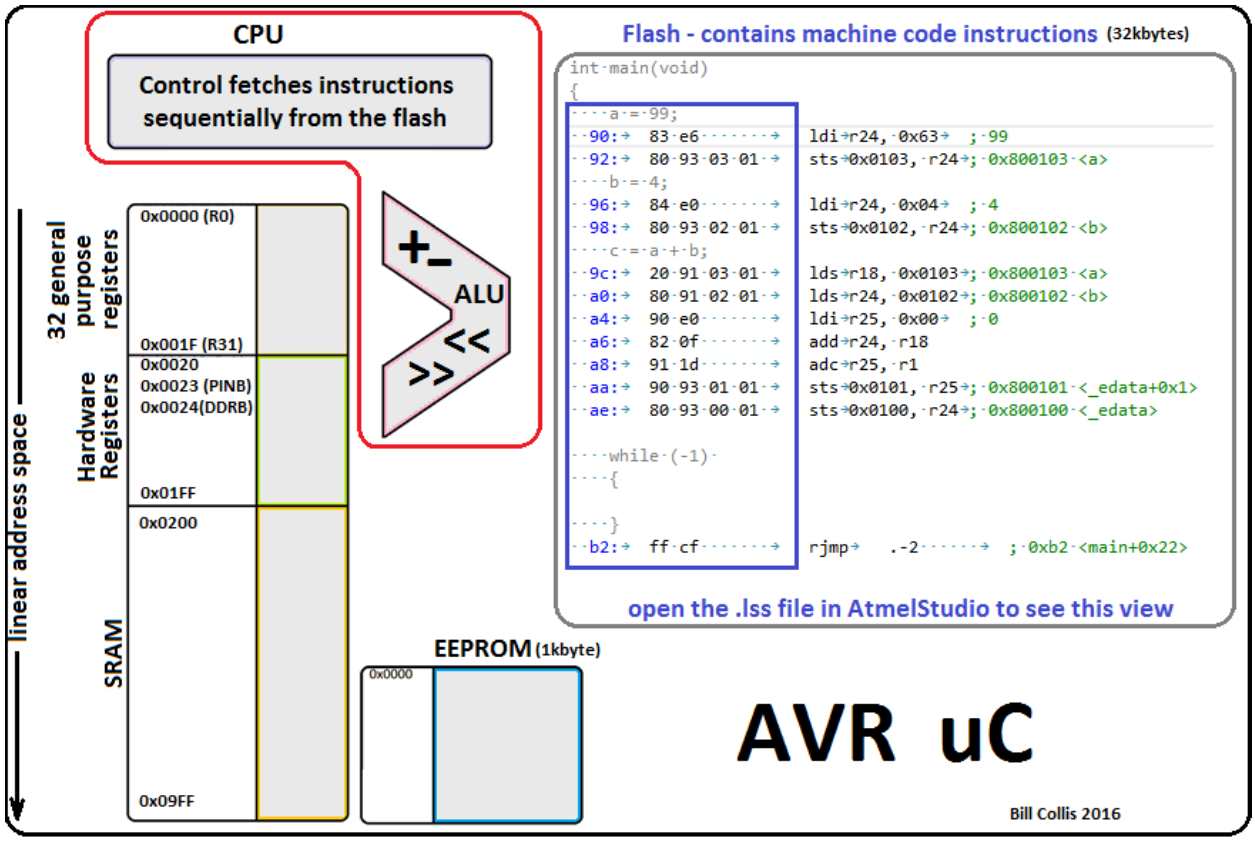
```
while (1) {
    …
    …
}
```

What do you suspect are the effects of these alternatives on the final program?

2.13.1. The processor/CPU/ALU/HOWRU

Microprocessors and microcontrollers contain a CPU (central processing unit) which has an ALU (arithmetic logic unit). Microcontrollers also contain memory: SRAM, FLASH and EEPROM. In a microprocessor based system, memory is separate e.g. a hard drive. The CPU control section sequentially fetches instructions from the program memory ('flash') and decodes them. Take the simple program to add 2 numbers in the box on the right. The machine code stored in Flash (starting at address 0x0090) is: 83, e6, 80, 93, 03, … The machine code can be viewed in the .hex file – however a more intelligible view of flash is found in the .lss file.

```
#include <avr/io.h>
volatile uint8_t a = 0;
volatile uint8_t b = 0;
volatile uint16_t c = 0;
int main(void)
{
   a = 99;
   b = 4;
   c = a + b;
   while (-1)  {   }
}
```



You need to know a few things: C programs are not stored in the uC – they are compiled into machine code and that is stored in the Flash. The human readable version of machine code is called assembler code or assembly language (assembler commands: ldi, sts, lds, add, adc are described in the datasheet). Instructions either transfer data between the general purpose registers and SRAM/EEPROM/hardware registers or carry out simple operations on data in the general purpose registers using the ALU. Instructions such as +, -, << are single ALU operations and limited in that most of them only work on registers so take only 1 clock cycle. Thus the AVR is a RISC- reduced instruction set computer as opposed to a Complex (CISC) where a single instruction may be more powerful containing multiple read, write and ALU operations. The AVR ALU is a simple logic device with no multiplication or division built into hardware. There is a contiguous address space for the general purpose registers, the hardware registers and the SRAM; whereas EEPOM and FLASH are separate address spaces. We refer to the IO as being memory-mapped IO – because it's addressed like it was SRAM. The AVR is a Harvard Architecture; if program and data memory were contiguous then it would be a Von-Neuman architecture.

2.14. Timer/Counters

LO: understand the ES as reactive and responsive to its environment

The uC's ability to react to the real world with any sort of timed accuracy is limited by its sequential nature. It can be impossible to calculate accurate timing within even a simple while(1){} loop because a program can react to different inputs, take different paths and therefore different time for each run through the loop.

To assist ES engineers accurately control timing, uC manufacturers have included a number of hardware counter/timers into their devices as standard features. This means that while the main software process is doing its regular jobs, it can be interrupted on a fixed timing schedule to do a critical task or to control an output or check an input.

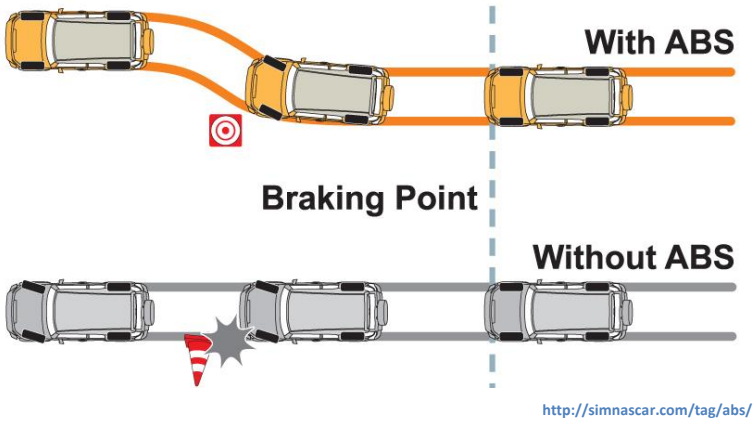
Some events in the real world need to be accurately controlled within tight timing constraints- sometimes in the order of 10^{-6} or 10^{-9} seconds. An example is a petrol engine and controlling the timing of the explosion of fuel in the cylinder increases fuel efficiency, develops the most power and reduces emissions.

A microcontroller is required to control the timing of output devices such as this not only with great accuracy but reliably (without ever missing an event).

See the animation at https://en.wikipedia.org/wiki/Petrol_engine

Some events in the real world occur at varying intervals and the timing of these events must be measured sometimes with 10^{-6} or 10^{-9} second's accuracy.

For example in vehicles the speed of each wheel needs to be measured accurately for the ABS (anti-lock braking system) to work. A microcontroller measures the speed of each wheel then determines whether one has a different speed to the others to see if the car is at risk of skidding; the brake for that wheel is then released slightly so that it keeps traction with the road. Polling (regular checking) of the input sensors for each wheel may not be good enough in this situation; the ES may need to react more quickly than the polling in a while loop allows. We can use the internal timers of the uC to interrupt normal program flow to check the wheel speed, meaning the uC will never miss an event.



The AVR Timers

The AVR uC's have one or more internal hardware binary timer/counters that count at a fixed rate, when a counter gets to the top of its count and overflows an interrupt is triggered. Then it starts counting again.

Watch out for the use of the word: timer – counter – divider – prescaler in this section. These devices are all physical just binary counters based upon flip flops, but we use the words to describe their different function.

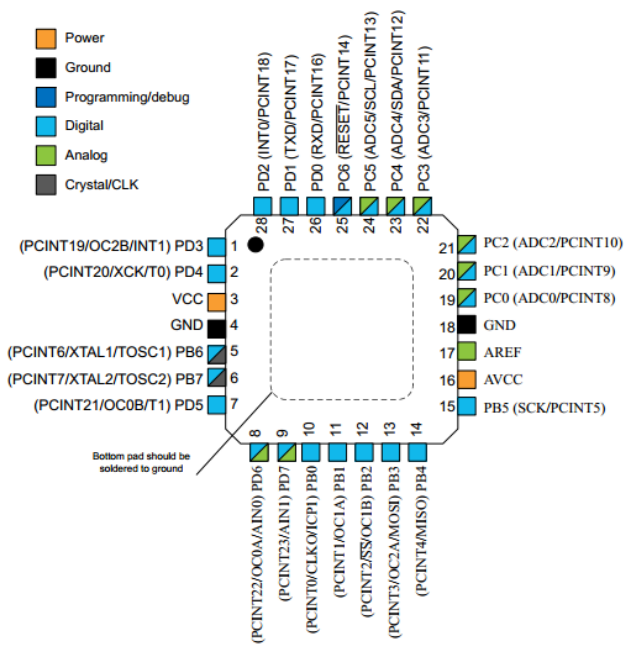
2.14.1. Timer options

- AVR's generally have
  - two 8 bit timer/counters
  - one 16 bit timer/counter
- clocking of the timer can be configured to be from:
  - the CPU clock source
  - the CPU clock source divided down (via a pre-scaler/divider)
  - Timer0 and Timer1 can be clocked via a related pin
  - Timer2 can be clocked from an external crystal
- counting is generally from 0 upwards
- timing completion/time out can be configured to occur
  - when the timer has overflowed
  - at a pre-set value,
- when the timer completes its count the timer can be configured to
  - modify a related output pin (set/clr/toggle)
  - interrupt the program and run the code in a function (called an ISR - Interrupt Service Routine)
- timers can be configured to start
  - when the input clock is configured
  - automatically after they overflow
- timers can be stopped
  - by disabling the input clock
- PWM is a further special mode for the timer

Each timer has a few related pins associated with it

Timer0
T0
OC0A
OC0B
Timer1
T1
ICP1
OC1A
OC1B
Timer 2
OC2A
OC2B
TOSC1
TOSC2

Figure 5-2. 28-pin MLF Top View



2.14.2. Timer example – 'heart beat' led flash

Describe how internal uC timers are used to make an ES responsive

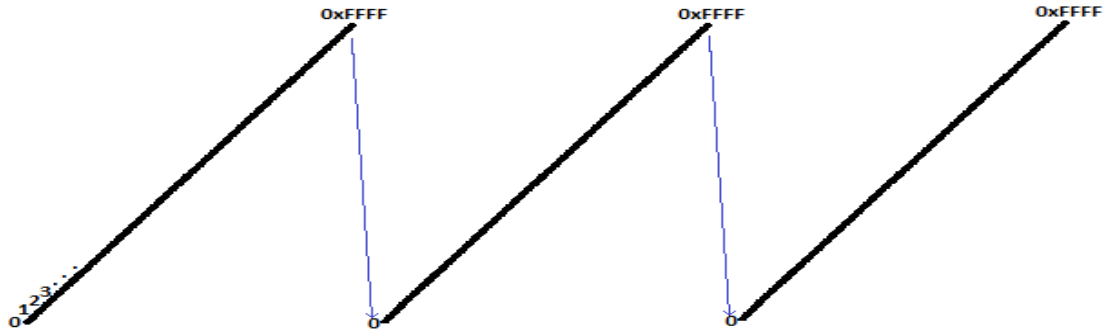
How to accurately control the timing of an external device without involving counting in the programs main loop.

Often users need to feel confidence that a piece of equipment is functioning. A steady lit LED is used to indicate that power is on; a flashing LED makes users believe that the equipment is actually working (like its alive, so we use the term heart beat).

Goal: Configure a timer to toggle an LED on an output every second

Timer Counter 1, (16 bit) overflow (the simplest operation)

The timer register (TCNT1) counts up from 0 to 65535 then overflows back to 0 (total count = 65536). In TIMSK1 bit TOIE1 is set so that on overflow (TCNT1 goes back to 0) an interrupt occurs and the code in an ISR (interrupt service routine) is run.

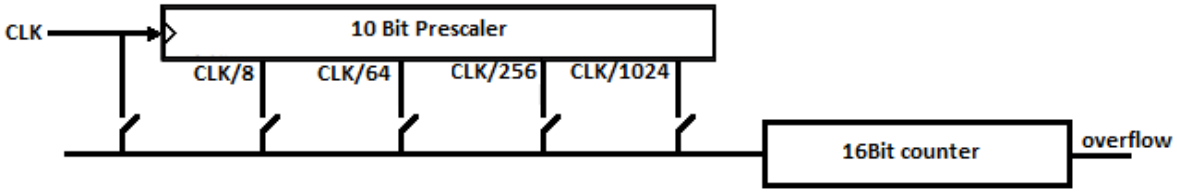


The rate of the counting is dependent upon F_CPU, the clock frequency.

At 16MHz a count of 1 takes _____ Sec = _____ mS = _____ uS = _____ nS

65,536 counts take _____ mS. in terms of some real world applications this is too fast.

A slower rate can be produced using a prescaler (counter/divider) for dividing the crystal frequency down before it goes to the main timer1 counter.



16Bit timer counter

Clock	No prescale	/8	/64	/256	/1024
TCNT0 input	16Mhz				
Time period of 1 count	0.0625 uS 62.5 nS				
Overflow (65536 counts)	4,096 mS				

2.14.3. The 'about' 1 second timer

```
/***** Project Header *****/
// Project Name: Timer1-FFFFOverflow

/***** Hardware defines *****/
#define F_CPU 16000000//crystal

/***** Includes *****/
#include <avr/io.h>
#include <avr/interrupt.h>
/***** Hardware macros *****/
#define TOGGLE_LED_1   PORTD ^= (1<<PD3) //could be any pin
/***** ISRs *****/
ISR(TIMER1_OVF_vect) { //called when the timer overflows
    TOGGLE_LED_1;
}
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD |= (1 << PD3);            //setup led pin as output

    /***** Timer Config *****/
    TCCR1B |= (1 << CS12);         // Prescaler /256
    TIMSK1 |= (1 << TOIE1);        // enable T1 overflow interrupt
    sei();                         // enable global interrupts

    /***** Loop code *****/
    while (1) {
        //nothing is needed here
        // however whatever is here will be interrupted every second
    } //end while(1)
} //end of main
```

What would have to change to make this an LED on A.2?

If we used the 8 bit timer / counter then the prescaler outputs and 8 bit overflow would be

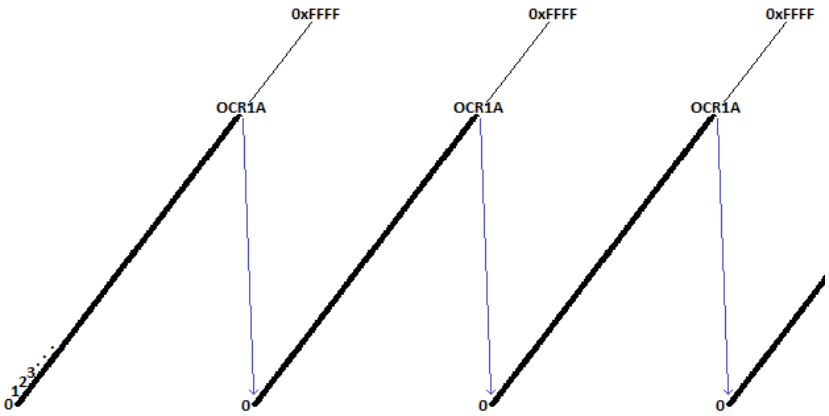
Clock	No prescale	/8	/64	/256	/1024
Counter input	16Mhz				
Time period of 1 count	0.0625mS				
length of 256 counts (include overflow to 0)					

Can you think of a way to use an 8 bit timer to get a delay of about 1 second?

2.14.4. The 'exact' 1 second timer

To get a more accurate value for a 1 second timer we can configure OCR1A so that the counter no longer counts to 0xFFFF (65535) then overflows but to some lesser value _____and then overflows

This value is setup in register _____




```
/***** Project Header *****/
// Project Name: Timer1-1Sec OCR1A Interrupt
/***** Hardware defines *****/
#define F_CPU 16000000//crystal

/***** Includes *****/
#include <avr/io.h>
#include <avr/interrupt.h>

/***** Hardware macros *****/
#define TOGGLE_LED_1   PORTD ^= (1<<PD3) //can be any pin on the uC

/***** ISRs *****/
ISR(TIMER1_COMPA_vect) { // ISR called every time T1 overflows OCR1A value
    TOGGLE_LED_1;             //TCNT1 automatically clears to 0 at OCR1A
}

/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD |= (1 << PD3);             //set led pin as output

    /***** Timer Config *****/
    TCCR1B |= (1 << WGM12);    // CTC clear timer on compare match mode
    TCCR1B |= (1 << CS12);     // Prescaler
    OCR1A   = 62499;          // the value to count up to
    TIMSK1 |= (1 << OCIE1A);  // enable CTC interrupt
    sei();                    // enable global interrupts

    /***** Loop code *****/
    while (1) {
        //nothing is needed here for the timer
        // however code that is here will be interrupted every second
    } //end while(1)
} //end of main
```

2.14.5. The 'exact' 1 second flash direct hardware output timer

A further option is to not use an ISR but to connect an LED to one of Timer1's related pins (OC1A or OC1B) This means once the timer is setup there is no ongoing software processing at all (i.e. no ISR) it is all hardware.

```
/***** Hardware defines *****/
#define F_CPU 16000000//crystal

/***** Includes *****/
#include <avr/io.h>

/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRB |= (1 << PB1);              //setup OC1A pin (B1) as output

    /***** Timer Config *****/
    TCCR1B |= (1 << WGM12);      // CTC mode
    TCCR1A |= (1 << COM1A0);     // toggle OC1A (B1 )output on match
    TCCR1B |= (1 << CS12);       // Prescaler /256
    OCR1A = 62499;

    /***** Loop code *****/
    while (1) {
    } //end while(1)
} //end of main
```

This option is completely autonomous, after it is setup it keeps going, no loop code, no interrupts

Can you develop a mathematical model for calculating timer counts?

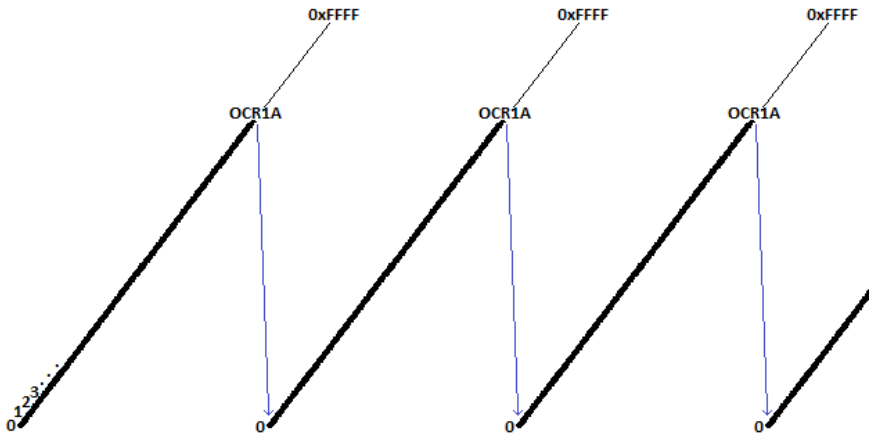
if there is a formula for this then there are calculators to help you figure it out as well, HOWEVER…

What would OCR1A need to become if we wanted 500mS on and 500mS off, and which prescale value should we use?

- prescaler 1 not possible as the max count is 65535 and we need to count to 16,000,000
- prescaler 8 not possible as the max count is 65535 and we need to count to 2,000,000
- prescaler 64 not possible as the max count is 65535 and we need to count to 250,000
- prescaler 256 OCR1 = _____
- prescaler 1024 OCR1A = _____

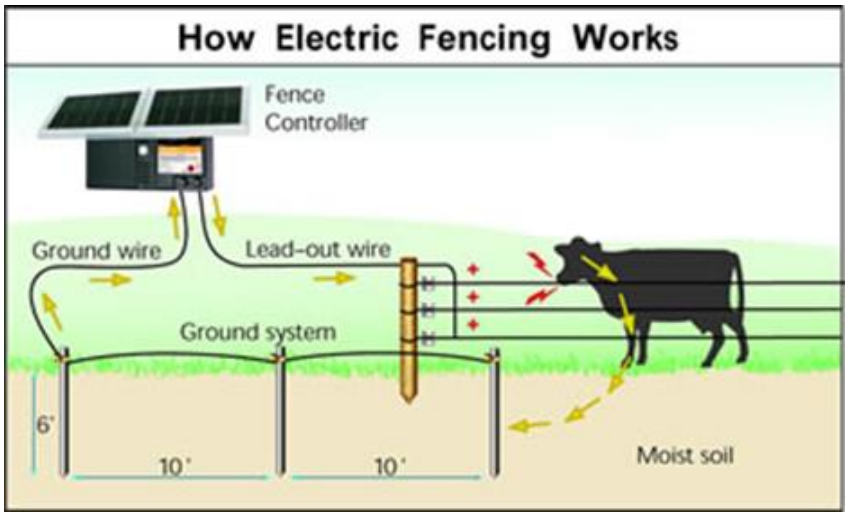
2.14.6. **Make a short duration pulse at a regular time interval**

OCR1A and OCR1B registers can be configured to control their respective pins OC1A and OC1B to control two different output devices, or the two interrupts COMPA_vect and COMPB_vect can be configured to provide two different interrupt events. One example that combines the two timer Interrupts would be to make an LED flash briefly once per second




```
/***** Includes *****/
#include <avr/io.h>
#include <avr/interrupt.h>
/***** Hardware macros *****/
#define SET_LED_1   PORTD |= (1<<PD3) //could be any pin
#define CLR_LED_1   PORTD &= ~(1<<PD3) //could be any pin

/***** ISRs *****/
ISR(TIMER1_COMPA_vect) { //called every time TCNT1 reaches TCCR1A
    CLR_LED_1;
}
ISR(TIMER1_COMPB_vect) { //called every time TCNT1 reaches TCCR1B
    SET_LED_1;
}
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD |= (1 << PD3);             //set led pin as output
    /***** Timer Config *****/
    TCCR1B |= (1 << WGM12);         // CTC mode
    TCCR1B |= ((1 << CS10) | (1 << CS12)); //Prescaler /1024
    OCR1A   = 19531;              // off time
    OCR1B   = 18531;                // on time
    TIMSK1 |= (1 << OCIE1A);        // enable CTC interrupt match A
    TIMSK1 |= (1 << OCIE1B);        // enable CTC interrupt match B
    sei();                          // enable global interrupts
    /***** Loop code *****/
    while (1) {
    } //end while(1)
} //end of main
```

This sort of timer could be used in an electric fence controller.



http://www.zarebasystems.com/blog/farm-equipment/why-electric-fence-system/

What is the on and off time for the LED in the previous code?

What should OCR1A and OCR1B be set to for an electric fence pulse of 35mS every 0.75 seconds?

Setup the timer for an electric fence pulse of 35mS every 2.7 seconds

2.14.7. Timer Registers

Each timer in the ATmega328P has a set of registers- these are described in the ATmega328 datasheet (sections 19,20,21,22)

- the timer/counter register itself
  - 8 bit - counts form 0 to
  - 16 bit - counts from 0 to
- a number of Timer Counter Control Registers
- a number of Output Compare Registers
- an interrupt mask register
- a flag register

Timer 0 Registers (8Bit)

- TCNT0 T0 (Timer 0) – the actual 8 bit binary counter
- TCCR0A T0 counter control register A – setup timer features
- TCCR0B T0 counter control register B– setup timer features
- TIMSK0 T0 interrupt mask register –enable T0 specific interrupts
- TIFR0 T0 interrupt flag register – we can check/clear interrupt conditions
- OCR0A T0 output compare register A the value that TCNT0 will be compared to
- OCR0B T0 output compare register B the value that TCNT0 will be compared to

Timer 1 (16 bit)

- TCNT1H/TCNT1L Timer 1 - High and low bytes of TCNT1
- TCCR1A
- TCCR1B
- TCCR1C
- TIMSK1
- TIFR1
- ICR1H/ICR1L
- OCR1AH/OCR1AL
- OCR1BH/OCR1BL

Timer 2 (8bit)

- TCNT
- TCCR2A
- TCCR2B
- TIMSK2
- TIFR2
- OCR2A
- OCR2B
- ASSR

The number of registers involved with each timer indicates that they have many options making them initially complex. A number of examples will help you become familiar with the datasheet, the control bits in the various control registers and what they do.

Setting up Timer1 registers (datasheet 20.14.2)

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

20-7 Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Write code to setup the following options: assume that the timer is being reconfigured during program use. This means that we must both clear (make 0) some bits and set (make 1) some bits.

Timer 1 no prescale:

Timer 1 prescaler =1024:

Timer1 stopped:

if we were working with Timer0 rather than Timer1 then we would change?

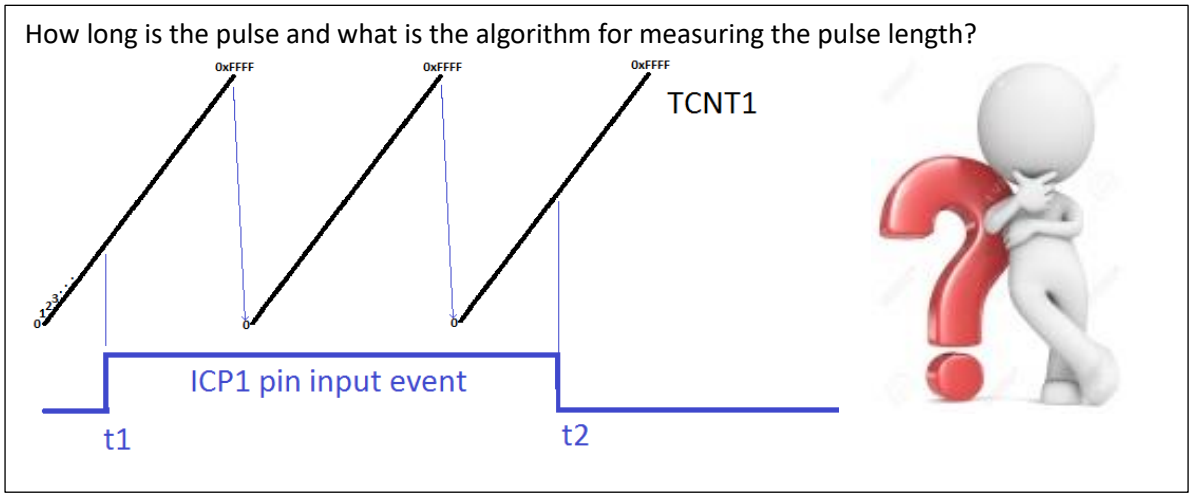
Feeling  then come to the extra tutorials and also ask questions on PIAZZA.

- ~~What is the answer to…?~~
- This is what I understand/have done… what does …?
- What does …mean?
- I am unsure about…?

2.14.1. **Use ICP1 to measure pulse width**

There are two main methods for using the timer related input pins these are: counting the number of pulses that occur on a related input pin (T0,T1) and counting the length of a pulse using ICP1. The lab requires you to use the second option.

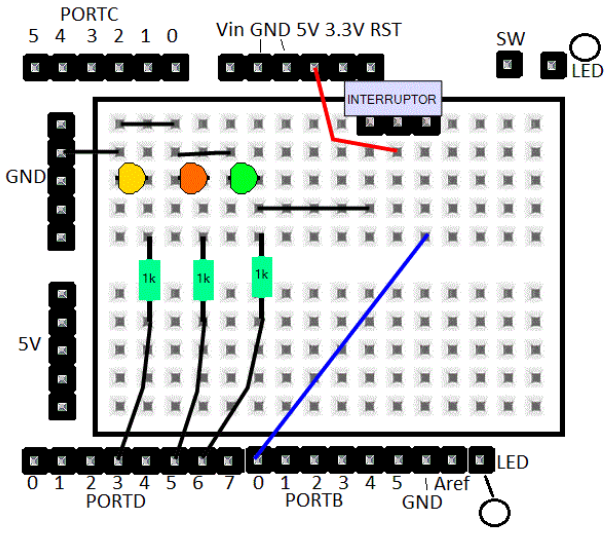
ICP1 is PIN B.) of the uC allows an external event on the pin to cause the uC to capture the value in TCNT1 when the input event occurred.



The timer is free-running (always counting) so the positive edge of ICP1 could be at any count of the timer. In the diagram above if the two counts were 15345 and 32071 in the above diagram what would the total pulse width be? The clock is 16MHz and the prescaler is set to 1024.

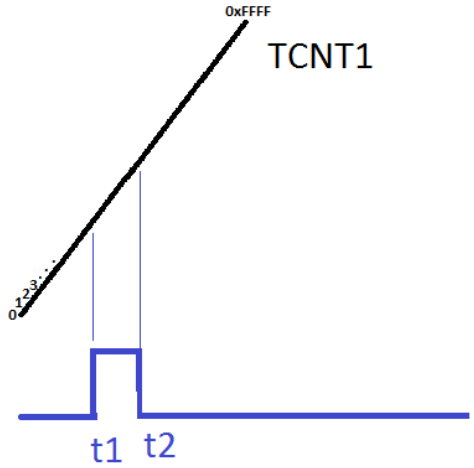
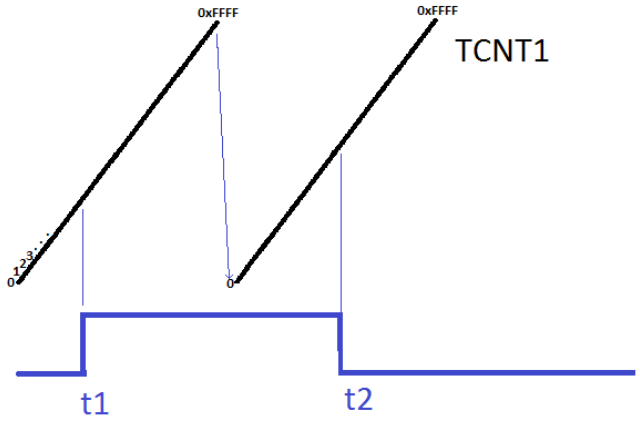
In the lab you will be displaying the time in hundredths of seconds (0.1S to 25.5S) using flashing LEDs

e.g. 13.7 seconds would be 1 flash of the first LED, 3 flashes of the second LED, and 7 flashes of the third LED.



We need an algorithm for the process, there are two complications that have be taken into account:

1. The ICP1 can be setup as either positive or negative edge triggered - but not both at once. So once the positive edge has been detected the pin must be reconfigured to sense a negative edge detect. And after the negative edge has been detected the pin must be reconfigured to sense a positive edge detect.
2. The overflow may or may not occur during the pulse measurement, this must be accounted for in your algorithm. Your logic must work whether there is an overflow or not

situation	formula
	Pulse duration =
	Pulse duration =

Write 1 algortithm that works for both situations.

Lab setup - Timer1 prescale = 1024

- TCCR1B |= (1<<CS12) | (1<<CS10);

Positive edge detect wth noise canceller

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ICNC1: Input Capture Noise Canceler**

Setting this bit (to one) activates the Input Capture Noise Canceler. When the noise canceler is activated, the input from the Input Capture pin (ICP1) is filtered. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output. The Input Capture is therefore delayed by four Oscillator cycles when the noise canceler is enabled.

- **Bit 6 – ICES1: Input Capture Edge Select**

This bit selects which edge on the Input Capture pin (ICP1) that is used to trigger a capture event. When the ICES1 bit is written to zero, a falling (negative) edge is used as trigger, and when the ICES1 bit is written to one, a rising (positive) edge will trigger the capture.

When a capture is triggered according to the ICES1 setting, the counter value is copied into the Input Capture Register (ICR1). The event will also set the Input Capture Flag (ICF1), and this can be used to cause an Input Capture Interrupt, if this interrupt is enabled.

When the ICR1 is used as TOP value (see description of the WGM13:0 bits located in the TCCR1A and the TCCR1B Register), the ICP1 is disconnected and consequently the Input Capture function is disabled.

TIMSK1 – Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 5 – ICIE1: Timer/Counter1, Input Capture Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Input Capture interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 57) is executed when the ICF1 Flag, located in TIFR1, is set.

- **Bit 0 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow interrupt is enabled. The corresponding Interrupt Vector (See "Interrupts" on page 57) is executed when the TOV1 Flag, located in TIFR1, is set.

- TCCR1B |= _____ //positive edge capture
- TCCR1B | = _____ //noise canceller
- TIMSK1 |= _____ //enable ICP interrupt
- TIMSK1 |= _____ //enable overflow interrupt
- sei();//enable global interrupts

```
#define ICP_ON_POS_EDGE _____________________ //detect ICP positive
edge

volatile uint16_t t1_count = 0;
volatile uint16_t t2_count = 0;
volatile uint16_t overflow_count = 0;
volatile uint32_t total_count = 0;
volatile uint8_t pulse_width = 0;


ISR(TIMER1_CAPT_vect)
{
    if ( ICP_ON_POS_EDGE ){         //pulse start, positive edge









    }
    else {                          //pulse finish, negative edge









    }
}

ISR(TIMER1_OVF_vect) {
    overflow_count++; //increment overflow count
}
```

Timer resolution, range, step size and accuracy

Timers are either 8-bit **resolution** or 16-bit **resolution** in the AVR, so they have a range of discrete values either from 0 to 255 or 0 to 65535. The process of quantization involves changing continuous values (e.g. time) to discrete values (values in a timer) so there are going to be some inaccuracies involved, the larger the step-size the more inaccurate the time period will be. For a clock frequency of 16MHz

Prescaler value of	Frequency input to counter	**Step-size** ** time for a count of 1	Period for a count of 2	**Range (8 bit timer)** Timer starts at 0, counts to 255 and overflows back to 0 = count of 256	**Range (16 bit)** Timer starts at 0, counts to 65535 and overflows back to 0 = count of 65536
1	16MHz	62.5nS	125nS		4mS
8	2MHz	500nS	1uS		32.768mS
64		4uS	8uS		262mS
256		16uS	32uS		1.04S
1024		64uS	128uS		4.194304S

** The best accuracy we can get using that prescaler with the 16MHz clock

How much accuracy is enough accuracy? This depends entirely upon the situation.

1. Does an electric fence controller that pulses once every 3 seconds need better than _____uS accuracy?

2. What timing accuracy does ABS need in your car?

3. What timing accuracy does a microwave timer need?

Can you see why engineers often work in teams?

Who would an engineer team up with to design answers to 1/2/3?

Timer Accuracy calculations

The discrete (non-continuous) nature of timers means that we may not always get the exact timing value we desire, so we must often do accuracy calculations for different frequencies, timers and prescale values. In the table below we can work out:

- an exact value for the number of counts
- then quantize (choose the nearest whole number for OCR)
- decide which timer we can use knowing that
  - Timer0 & Timer2 can count from 0 to 255
  - Timer 1 can count from 0 to 65535
- work out the error for each option to determine if we can get the accuracy we require.

Set up a timer to provide a timing period of 0.419mS**(+/-0.5%)** = 416.905uS to 421.095uS

	OCR value = F_CPU * delay period /prescaler -1				
Freq	16000000	16000000	16000000	16000000	16000000
prescaler	1024	256	64	8	1
Step-size (sec)	0.000064	0.000016	0.000004	0.0000005	62.5E-09
desired period (Sec)	**0.000419**	**0.000419**	**0.000419**	**0.000419**	**0.000419**
Number of counts	5.546875	25.1875	103.75	837	6703
discrete OCR value (floor)	5	25	103		
discrete OCR value (ceiling)	6	26	104		
	delay period = (OCR Value + 1) * prescaler / F_CPU				
Chosen OCR value	5	6	25	104	
actual period		448uS	416uS	420uS	
error		29.0E-6	3.0E-6	1.0E-6	
error%		6.9%	0.7%	0.2%	0%	0%

0.419mS(+/-0.5%)

T0/T2 use OCR0A/OCR2A =

T1 use OCR1A =

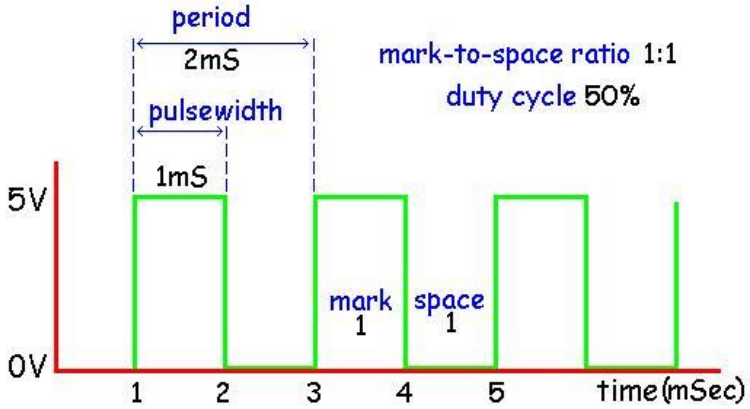
Timers and PWM (pulse width modulation)

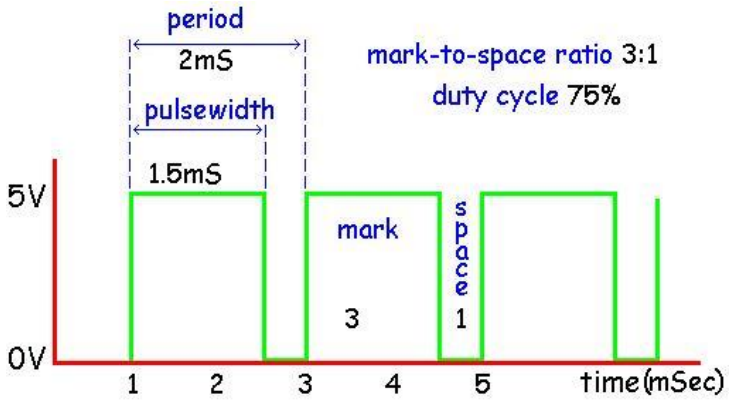
Timers can be used to turn outputs on and off with different ratios of on to off time, this is called PWM. There are many uses for PWM, some of the common ones include:

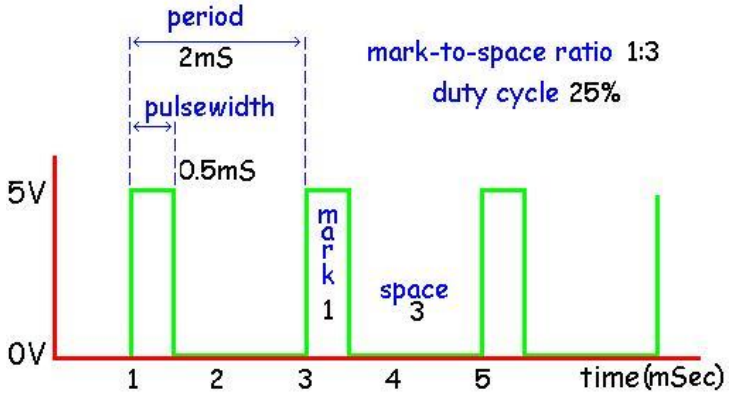
- Fan/motor speed
- LED / light brightness
- Servo motors

The key concept is to generate a fixed frequency signal but vary the ratio of the on/off time

If we connected a motor to a uC pin, which of the following three pulses would make the motor go slowest?







LO: describe how internal uC timers are used to make an ES responsive

Timer uses	**How timer(s) could be used**
Anemometer	Count number of revolutions per second or per 10seconds, Or measure the period between pulses
time-lapse camera shutter controller	Press a switch to turn on an output, start a timer, When timer timeouts, turn off output and stop timer
Beeper	Make an output of a certain frequency and a certain duration
engine rpm	
fridge door	When a switch is opened start a timer When the switch is closed stop and reset the timer After the timeout make a beeper sound until switch is closed
Siren	Varying the frequency of a tone
scanning 7 segment displays	Very common- drive 1 digit at a time, but fast enough so that users will not see the flickering.
Real time clock	Timer2 has a special mode that divides down a 32,768Hz crystal to get a 1 second interrupt If a crystal already exists on the board, then configure an timer to for a 1 second or some fraction of 1 second interrupt Then make a make a clock in software (remember to account for different days in months and leap years)
De-bouncing switches	To avoid blocking code – the use of _delay_ms() use a timer instead.
ultrasonic object ranging	Send a short duration pulse of ultrasonic frequency, start a timer. Stop the timer when the pulse reflection is detected and work out the length of time
servo motors	Pulse width modulation
Motor speed control	PWM, make sure the frequency matches the characteristics of the inductor in the motor, so that the has time to respond to the changing between 1 and Ramp up a motors from stopped to full speed slowly
delayed turn on /off of an output	Lock lecture theatre doors so that late students cannot come in.
Dead-man switch	Press start switch, output goes on, Press the switch every so often or the output goes off As used in …
Man-down switch	Mount a switch to the side of a two-way radio, if the radio lies on its side for more than ½ second send out an emergency signal As used in …
Mix colours of RGB LED into any colour	RGB LEDs have three separate LEDs in one package, to make other colours vary the brightness of each LED individually using PWM.

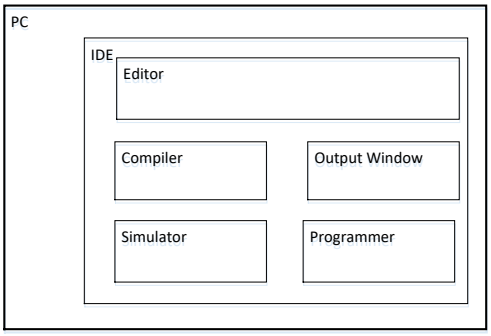
2.15. ES development overview

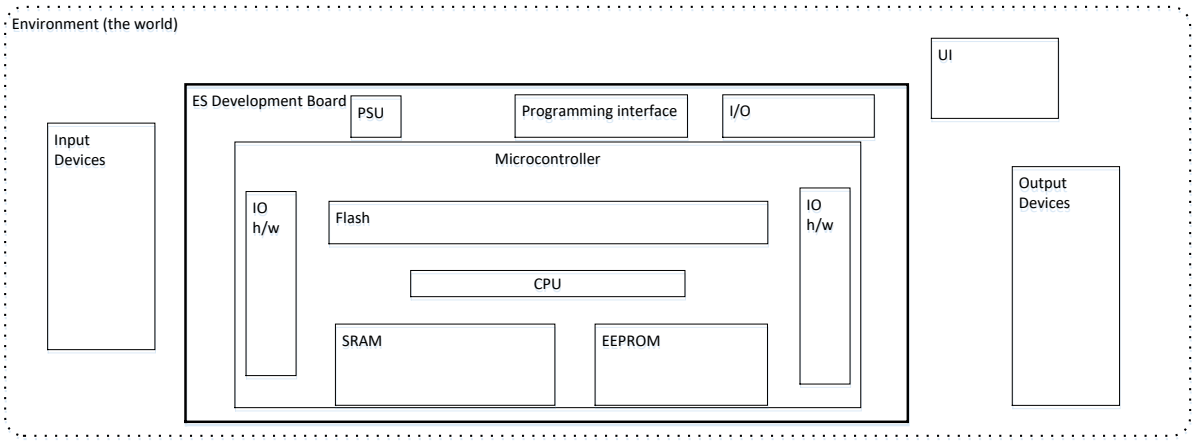
So far you have used a simulator to develop your ideas around simple AVR uC based ES's, you can see an overview YouTube video about this here at HelloWorldSimulation.XplainItToMe.com

Here we will look at how to get code into a real uC development board an overview of this process can be found in a YouTube video here at HelloWorldProgram.XplainItToMe.com

When developing ES software a number of processes are encountered

1. IDE – Atmel Studio 7
2. Writing code
3. Syntax errors
4. Compiling
5. uploading

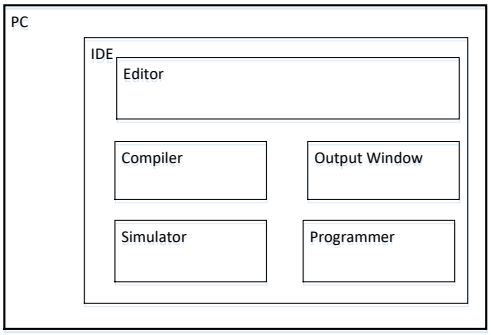


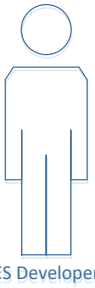


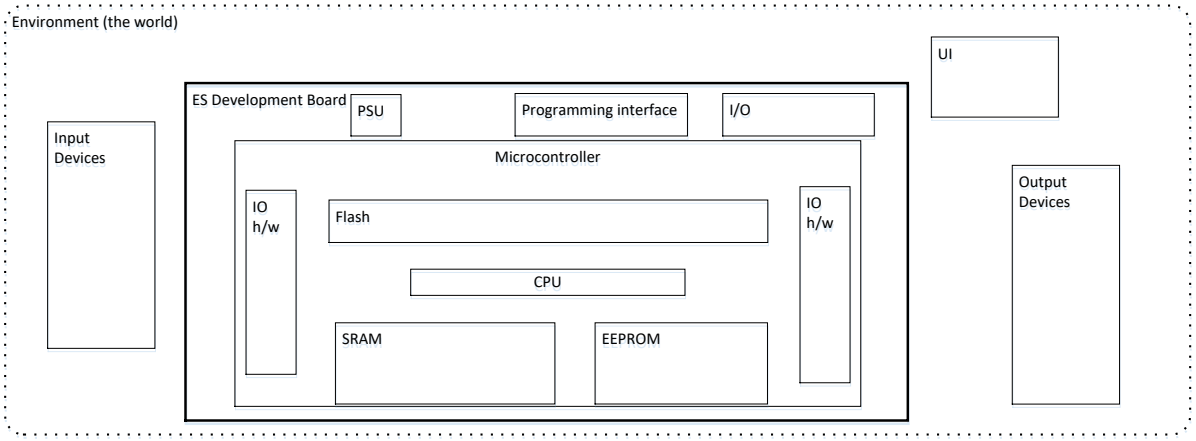
What about when the ES doesn't work quite right

These are called semantic or logic errors

We need to check the systems correct operation and identify/debug any errors







Why is a smart phone an ES (or not)?

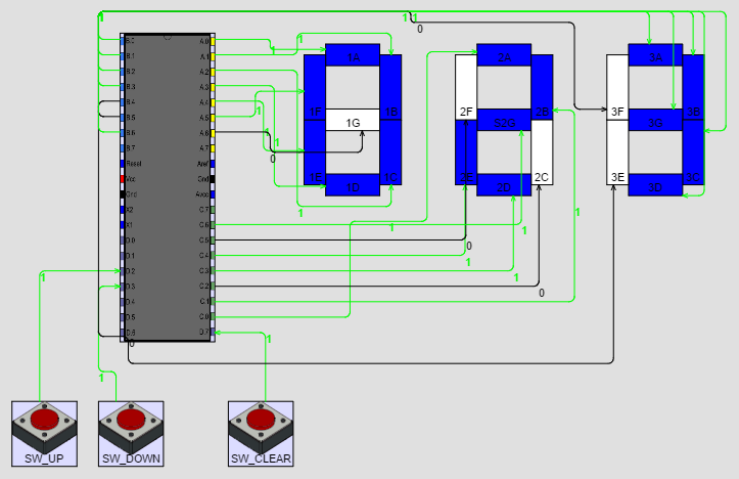
2.16. Memory mapped IO and pointers

Describe the different memories of the uC and their uses

On a boarding pass is the seat number, in the seat is me.
How does each of these relate to pointer use?

- the boarding pass?
- the seat number?
- the passenger?

In ES one use for pointers is to refer to hardware registers, in this counter the 3 displays are connected to different ports, but wired alike.



We **could** write three different display functions

```
void disp(int16_t count){
    //split the count into 3 digits
    disp_hundreds (hundreds);
    disp_tens (tens);
    disp_units (units);
}
void disp_hundreds(uint8_t hundreds){
    if(number == 0){
        PORTA = 0b11000000; //PORT A
    }
    if(number == ...
}
void disp_tens(uint8_t tens){
    if(number == 0){
        PORTC = 0b11000000; //PORT C
    }
    if(number ==...
}
void disp_units(uint8_t units){
    if(number == 0){
        PORTB = 0b11000000; //PORT B
    }
    if(number ==...
} (but this is WASTEFUL of our memory)
```

in C we use knowledge of integer arithmetic and a very helpful function called modulus (%) to help us separate a number into individual digits.

In **integer arithmetic**

9/3 =3 9 % 3 = 0

10/3 = 3 10 % 3 = 1

11/3 =3 11 % 3 = 2

12/3 = 4 12 % 3 = 0

modulus = A-B*(A/B)

no rounding

e.g. if count = 246

hundreds = count / 100 =

tens = count % 100

tens = tens /10

units = …

It is more efficient to write one function than three and pass the address of the specific port register in a pointer. We can do this because the IO is memory-mapped.

```
void disp(int16_t count){
    volatile uint8_t* p_portaddress =  & PORTC; //declare a pointer
    //split the count into the required 3 digits
    p_portaddress = &PORTA; // pointer contains the address of PORTA
    display_digit(p_portaddress, hundreds);









}

void display_digit(volatile uint8_t *port, uint8_t number){
    if(number == 0){
        *port=0b11000000; //store the value in the correct register
    }
    if(number == 1){……}
```

Memory	Name	contents	
0x103			
0x102			
0x101			
0x100			
…			
0x29	PIND		
0x28	PORTC		
0x27	DDRC		
0x27	PINC		

How does each of these relate to pointer use?

- the boarding pass?
- the seat number?
- the passenger?

2.17. Volatile

Compilers convert high level languages to machine code, but they also are written to perform optimization during this process. Optimizations they perform include the removal of useless or unreachable code. When the same variable appears in different scopes it can appear to the compiler as unreachable or different variables, it will then remove them. The word volatile in English means unpredictable and when used as a keyword instructs the compiler not to optimize code related to the variable as it may change in ways outside the compiler's understanding. Variables associated with pointers may change during runtime so can appear as useless or unreachable so must be declared as volatile.

2.18. Writing bug free code (hopefully)

It is cheaper and easier to prevent bugs from creeping into your programs than it is to find and kill them once you realise they are there. 56% of the coding work done is rewriting existing code, so the clearer and more readable your code is the easier it will be to modify (even if you are the only person who will ever read it)

http://img.timeinc.net/time/daily/2010/1010/360_stein_1018.jpg

Become a _____

- think about how you will document your code before you begin writing it
- think about code reliability and portability (re-use) and not about execution efficiency, your convenience or trying to show off by writing esoteric code
- stick to one naming convention: for variables and functions in embedded C use lowercase with underscore separators, for constants uppercase with underscore separators
  void hello_world()
  #define MY_AUID 2345678
  #define TEMPERATURE_COEFFICICENT 324
- Use descriptive variable and function names:
- Think about the type of variable to use
  - the temperature outside **uint8_t temp** (is this ok?) int8_t
  - the temperature in a cool store
  - the relative humidity in the greenhouse
  - the floor a lift is on
  - whether the car door is locked or unlocked

- Only put one task into a function
- Keep functions to 1 viewable page of code
- Always use {} with if statements- even though not entirely necessary
- Indent your code – 4 spaces
- Comment blocks and lines of code with their purpose and function (draw diagrams)
  - avoid stating the obvious about syntax unless it is complex or abstract
- Use local variables whenever possible. If a variable is used only in a function, then it should be declared inside the function

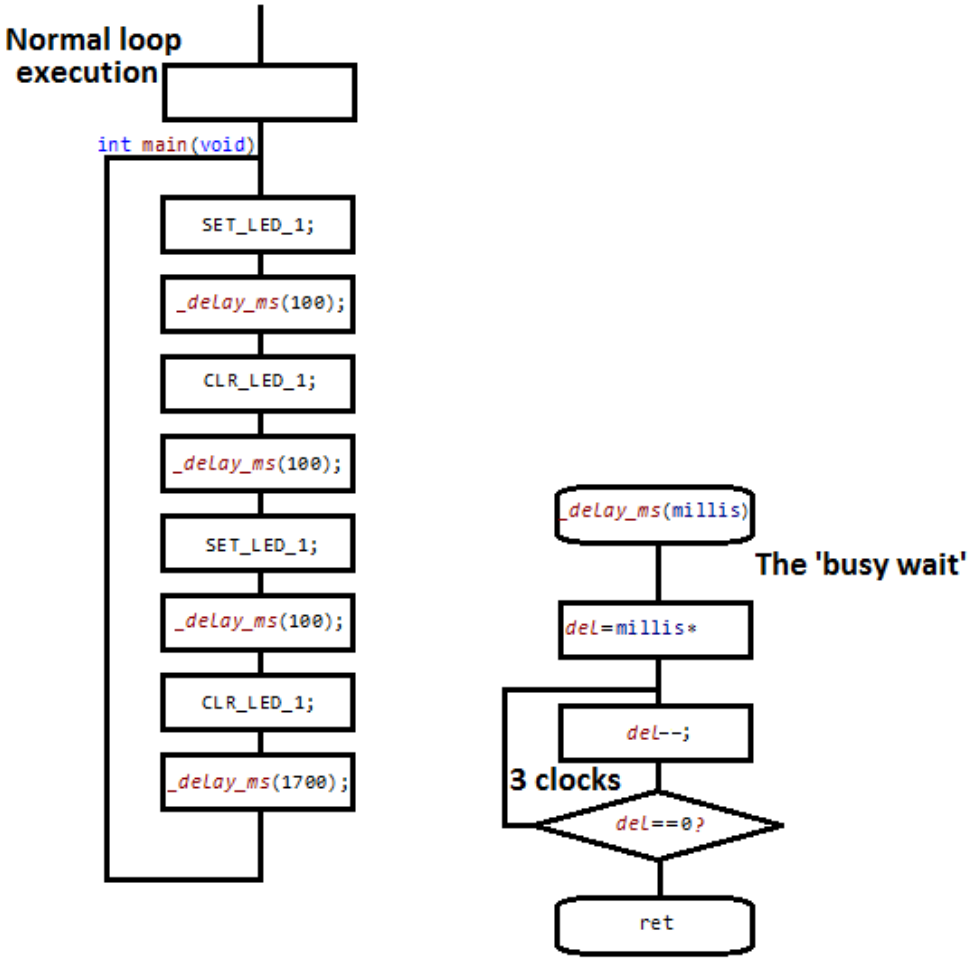
2.19. External Interrupts

LO: describe how uC external interrupts are used to make an ES reactive

Timers are great – however uC's have another type of interrupt called an external interrupt, this allow the ES to react to untimed but urgent things that occur around in its environment, without having to poll for them.

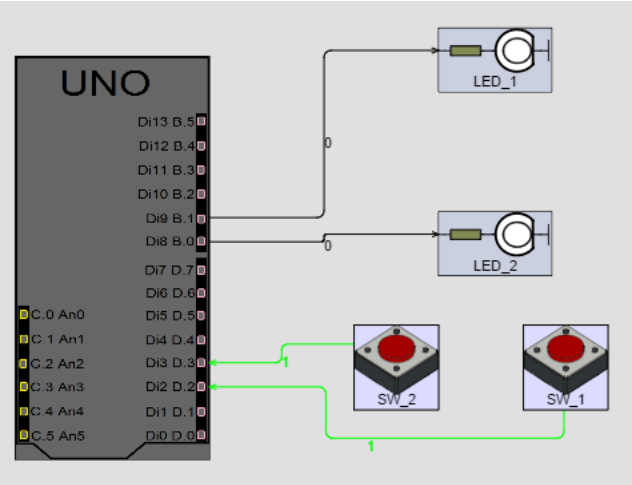
uC's are sequential devices so they can only do one thing at a time; if the uC is busy doing one task it might miss an input of short duration, or if the number of inputs to check is large then even if it checks them all, it may be not be often enough, and miss one, or more. Importantly if the input is mission critical then it must be consistently monitored so that it is never missed.

Also delays like _delay_ms() are sometimes essential in ES software because we need to slow outputs down so that slow environments are given the time to react to the system, or so that a UI component can be recognised by a user (e.g. a flashing LED). While the uC is waiting however it cannot do anything else, hence the name busy wait for _delay_ms() commands.

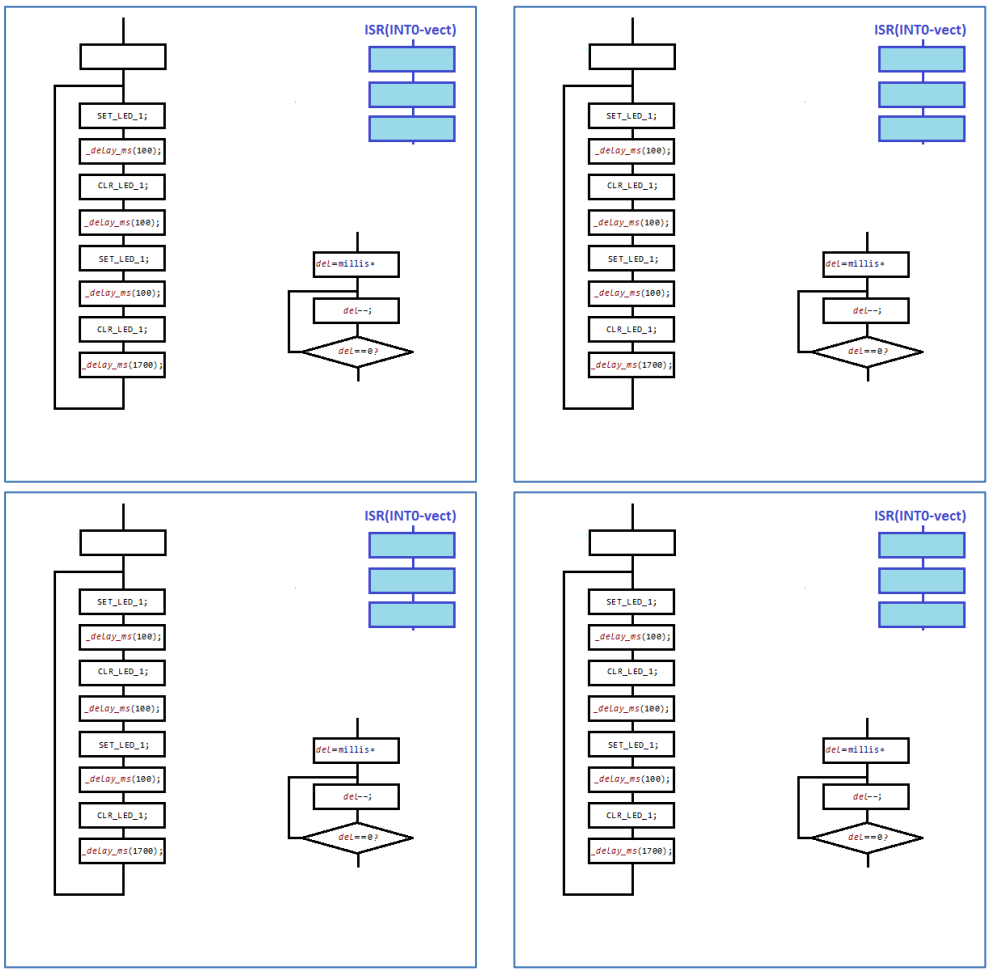


In response to this manufacturers now implement special hardware features in uC's called external interrupts. There are several different interrupt facilities in the AVR. By connecting an input to an external interrupt pin we don't have to poll the pin but let the interrupt do it for us. A hardware interrupt is effectively ....

External interrupts INT0 and INT1



LED_1 does a double flash every 2 seconds. The switches are connected to D.2 and D.3, these are also the interrupt inputs INT0 and INT1. When the interrupt is triggered the uC completes execution of its current instruction then carries out the instructions in the ISR (Interrupt Service Routine), when those instructions are finished it carries on execution at the next instruction in the main (or delay) function Because of interrupt features we can give the ..



It doesn't matter where the code is it will jump to the ISR, carry it out, then return

External interrupts setup

The External Interrupts are triggered by the INT0 and INT1 pins. The External Interrupts can be triggered by:

- falling (negative) edge/potential on the pin
- rising (positive) edge
- low level- will repeatedly happen while the pin is low

The set up for INT0 or INT1 requires bits in 3 registers to be setup and appropriate ISR (Interrupt Service Routines) to be written

17.2.1 EICRA – External Interrupt Control Register A

The External Interrupt Control Register A contains control bits for interrupt sense control.

Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

write the code to setup INT1 as rising edge

EICRA _____

EIMSK – External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

EIMSK _____

SREG – AVR Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The AVR Status register is manipulated using special GCC macros sei(); and cli(); to set/clear the Global Interrupt Enable in SREG.

The default setting is that bit I is 0 in SREG, so are any interrupts enabled on powerup?

Setting up External Interrupts INT0 & INT1

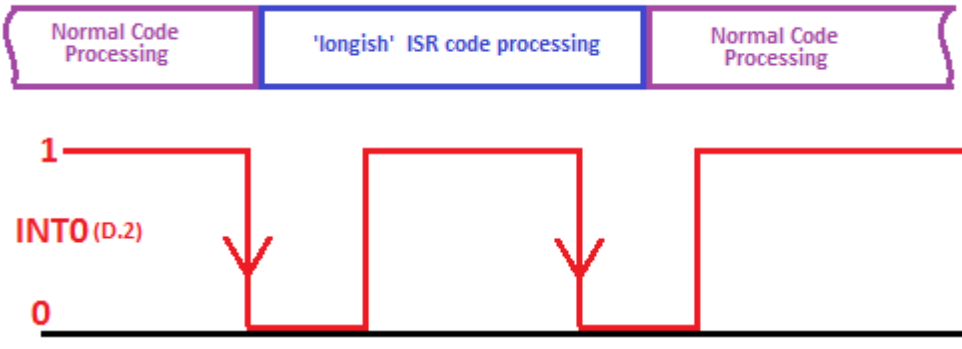
```
/***** Includes *****/
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
/***** Hardware macros *****/
//Hardware macros for outputs
#define SET_LED_2   PORTB |= (1<<PB0)
#define CLR_LED_2   PORTB &= ~(1<<PB0)
#define TOGGLE_LED_1   PORTB ^= (1<<PB4)
//Hardware macros for inputs
#define SW_1_IS_LOW   ~PIND & (1<<PD2)
#define SW_1_IS_HIGH   PIND & (1<<PD2)
#define SW_2_IS_LOW   ~PIND & (1<<PD3)
#define SW_2_IS_HIGH   PIND & (1<<PD3)
/***** Interrupt Service Routines *****/
ISR(INT0_vect){
    SET_LED_2;
}
ISR(INT1_vect){
    CLR_LED_2;
}
/***** Main program *****/
int main(void) {
    // Initially make all micro pins outputs
    DDRB = 0xff; //set as outputs
    DDRC = 0xff; //set as outputs
    DDRD = 0xff; //set as outputs
    // make these pins inputs
    DDRD &= ~ (1 << PD2);
    DDRD &= ~ (1 << PD3);
    PORTD |= (1 << PD2); //activate internal pullup resistor
    PORTD |= (1 << PD3); //activate internal pullup resistor
    //Interrupt setups




    /***** Loop code *****/
    while (1) {
        TOGGLE_LED_1;
        _delay_ms(100);
    } //end while(1)
} //end of main
```

2.19.1. **Interrupt complications**

Discuss issues of an ES's responsiveness with regard to polling, blocking and interrupts

Lengthy interrupts

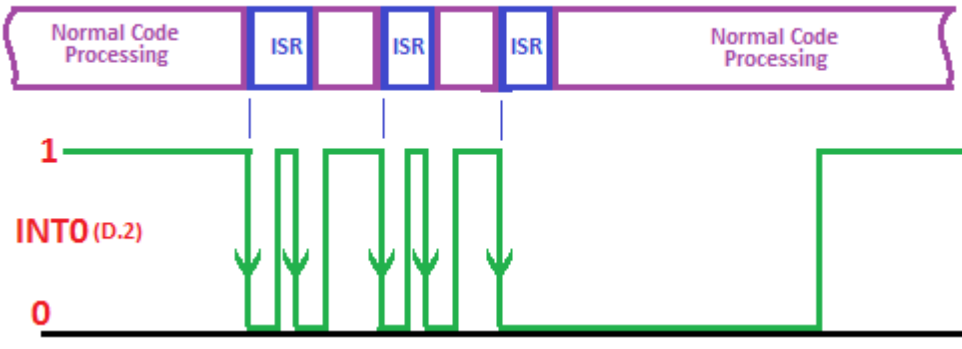
Once an interrupt is triggered and the ISR begins to execute, the global interrupt bit in the SREG is disabled, this means that if the same interrupt occurs during this period of time the uC will not sense it and it will not be detected (other, different types of interrupts however will be queued).



So how long should an ISR be?

Short interrupt code

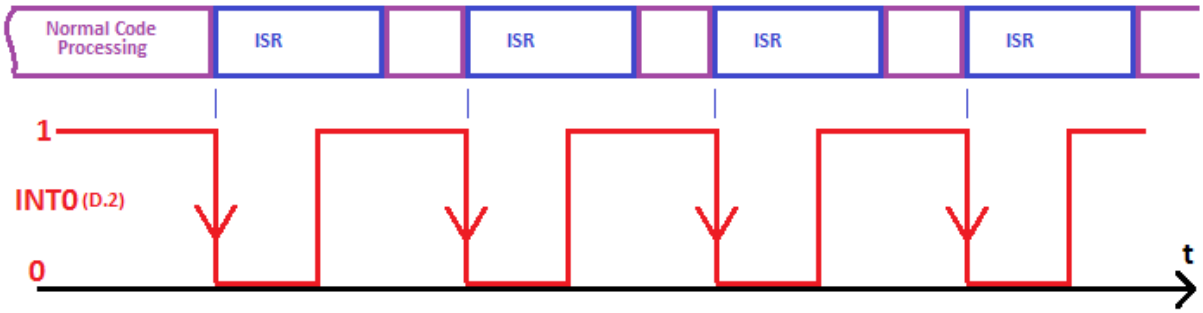
What about when ISR's are very short – only taking a few cycles of the clock – if we connect mechanical switches to an external interrupt pin – we might find that one bouncing contact input causes many interrupts



So how short should an ISR be?

Too many interrupts

When your interrupt is executing your main program is not, so excessive interrupts might…?



Interrupts and variables

When variables are shared between the main function and interrupt functions they must be declared as volatile so that the compiler knows not to optimise them away

Atomicity

The AVR memory is 8 bits wide, so when 16 bit variables are manipulated they get changed 1 byte at a time. However should an interrupt happen at the point in time when only 1 half of the variable has changed and then the interrupt routine changes the variable the value will get corrupted? This issue relates to what is called atomicity. While in C a command may appear as a single operation, in the machine code multiple operations may be taking place. Interrupts should not be able to occur during non-atomic operations

```
/***** Declare & initialise global variables *****/
volatile uint16_t time_delay = 500;

void my_function(){

   _________ //disable interrupts to protect critical section

   uint16_t count = time_delay; //actually involves multiple operations

   ___________
}
ISR(INT0_vect){
    time_delay++; //ISR changes value in variable
}
```

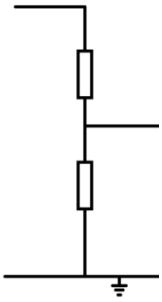
Why wouldn't atomicity be an issue when developing apps for a PC?

2.20. ADC

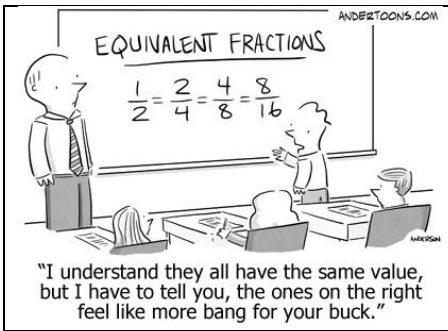
The real world is not digital, it is analog, so for an ES to be reactive to real world events it must be able to convert information from the analog world to the digital one.

2.20.1. The voltage divider

At the core of converting energy in the real world into the electrical world is this fundamental electrical circuit



Working in context with voltage dividers is at the core of using ADCs and sensor subsystem design, and this involves understanding **ratios** (equivalent fractions)

	The voltage divider is a

This ratio can be expressed in 4 different ways depending upon what we want to use it for

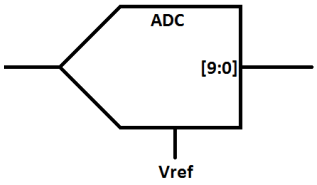
Situation	Solve for the unknown
	Vout =
we know the upper and lower resistor values and what we want the output to be so therefore	Vin =
	R1 =
	R2 =

ADC conversion process

Describe the significant characteristics of the internal ADC

Inside the uC is an analog to digital converter it turns a voltage into a 10 bit binary number and stores in TWO registers- your code can then read the registers and perform calculations on the value

This is called quantization, the process of mapping a continuous signal to a digital one. Quantization replaces each real number with an approximation from a finite set of discrete values



The Atmel AVR range has a **single** internal ADC with _____

resolution is the number of bits in the digital output code of the ADC

The AVR ADC has _____ discrete values or steps: 0 to _____

When Vin = 0 the adc output value is _____

When Vin = Vref the adc output value is _____

The relationship between input voltage and output number is a linear process and can be expressed as a _____

What is the simple ratio that expresses the relationship between input voltage, ADC resolution, ADC output value, reference voltage?

The ATmega328P has 3 different reference options:

- 1.1V (accurate internal reference)
- AVCC (use the power supply as a reference)
- AREF (connect your own reference voltage - cannot exceed VCC)

Vref = 1.1V, Vin = 0.67V	ADC value =
Vin = 2.2V, Vref = 5.03V	ADC value =
ADC value = 754, Vref =5.07V	Vin =
ADC value = 432, Vref =1.1	Vin =

ADC reference select

Figure 28-1. Analog to Digital Converter Block Schematic Operation

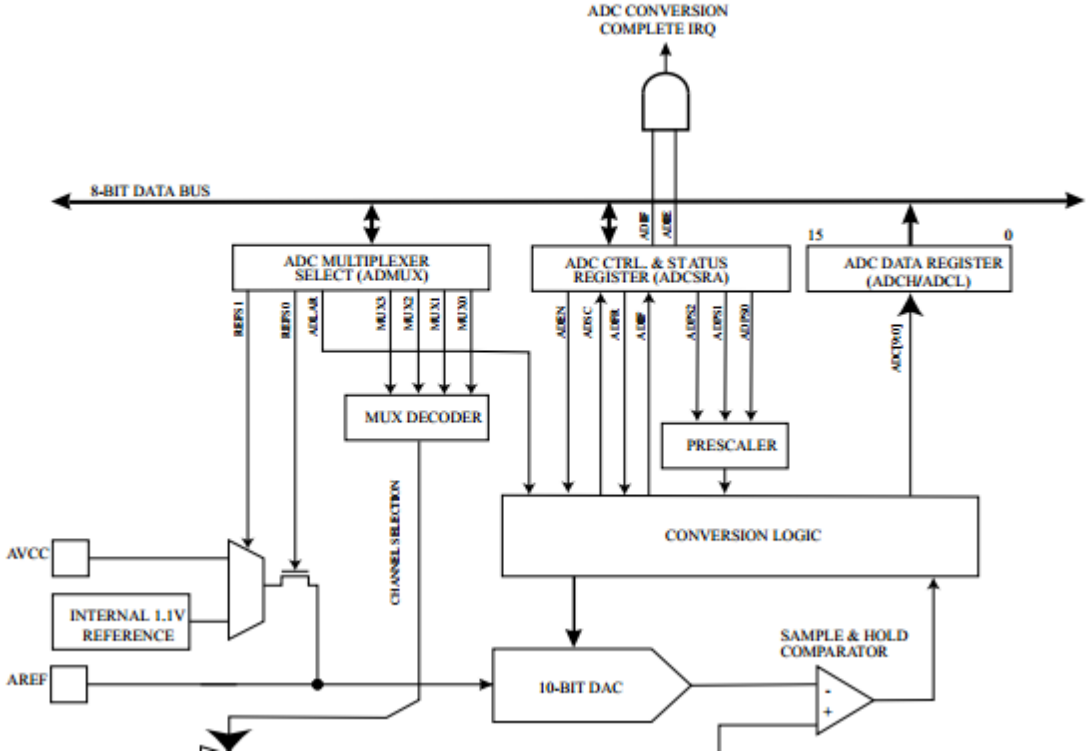


Table 28-3. ADC Voltage Reference Selection

REFS[1:0]	Voltage Reference Selection
00	AREF, Internal V_{ref} turned off
01	AV_{CC} with external capacitor at AREF pin
10	Reserved
11	Internal 1.1V Voltage Reference with external capacitor at AREF pin

What would be the command(s) to select the internal Vref?

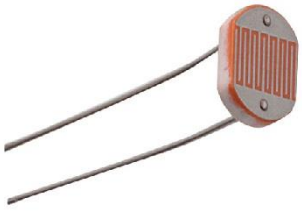
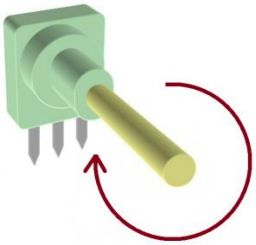
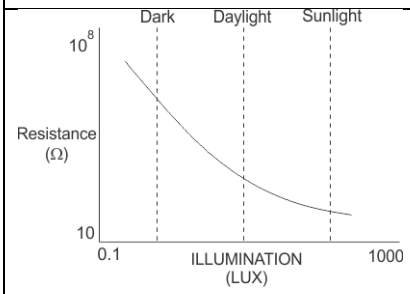
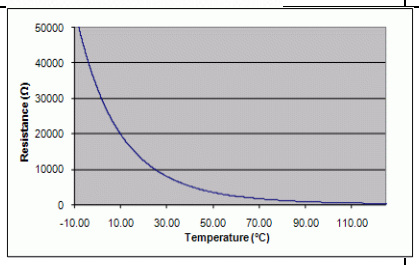
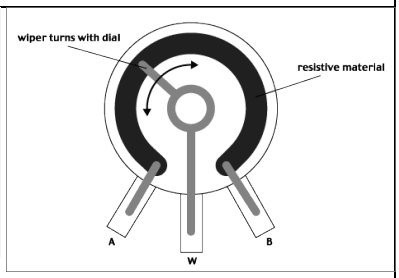
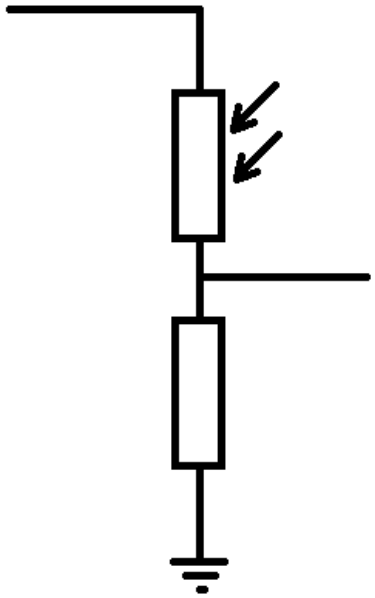
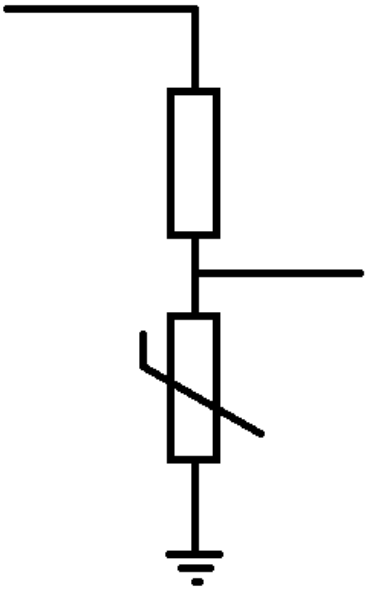
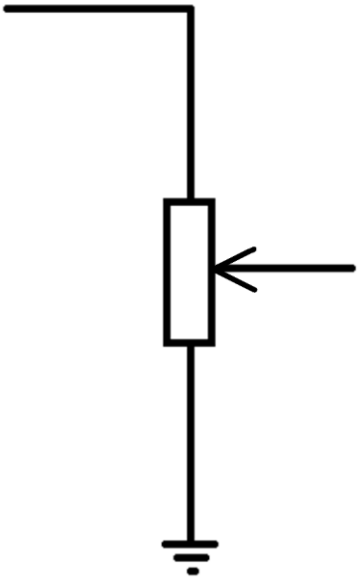
If you needed to change Vref during the program,

what would be the command(s) to select AVcc as the reference?

2.20.2. **ES analog sensors**

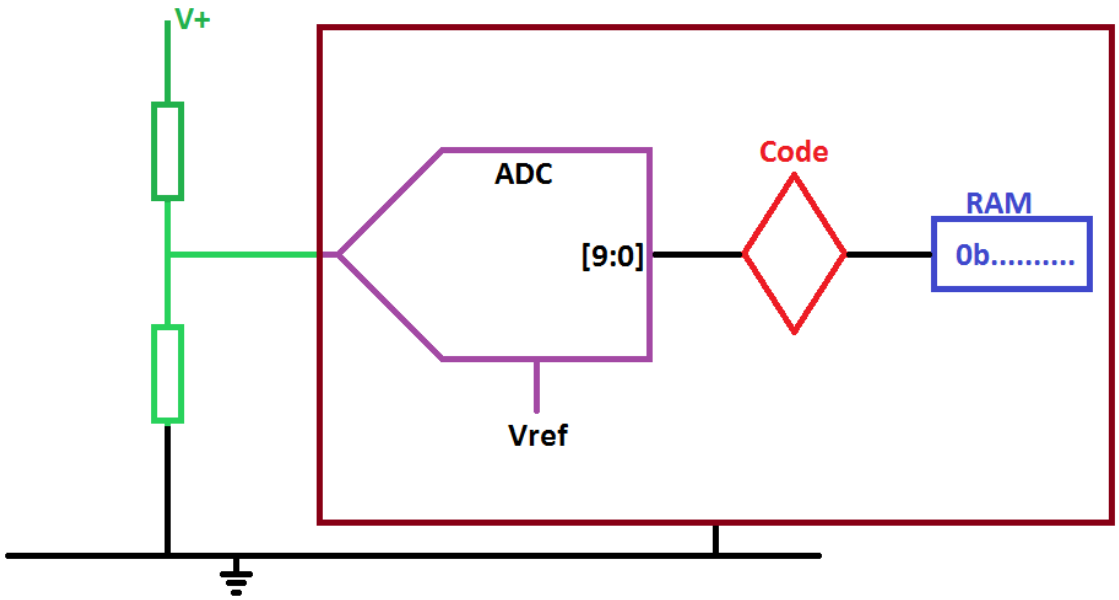
Describe the use of a sensor in a voltage divider circuit

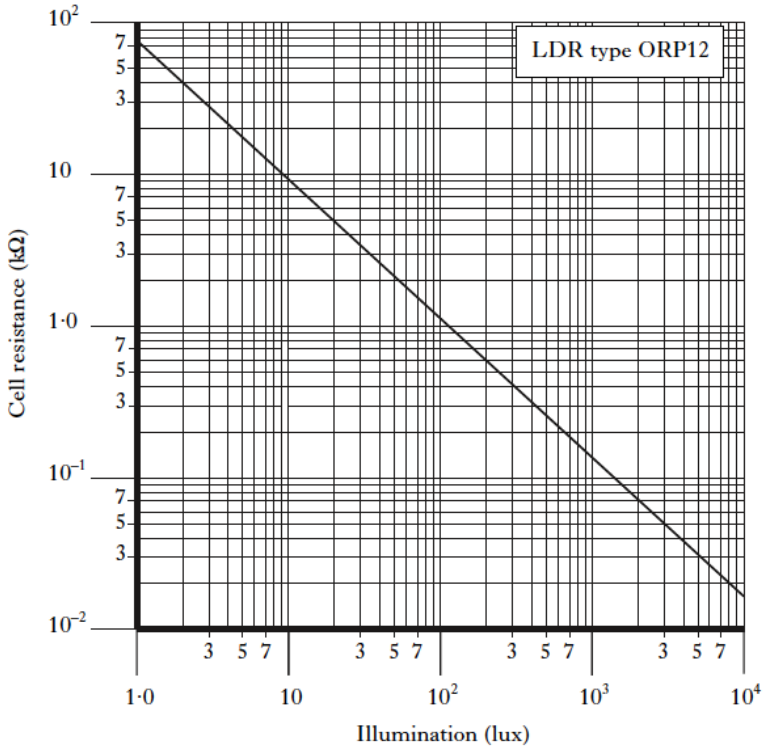
In an ES voltage divider circuit there are not two fixed value resistors but one component which has variable resistance and one which is fixed. Either can be the upper or lower resistor in the circuit.

Light measurement	Temperature measurement	Displacement (potentiometer)
		
		 (linear or lagarithmioc over 270 degrees range)
		
As the light level goes up the output voltage goes _____	As the temperature goes up the output voltage goes _____	As the displacement of the object changes the output voltage changes

When selecting a value for the fixed resistor in a voltage divider start by choosing a value in the middle of the range you are wanting the device to be most sensitive in; e.g. in a thermistor circuit if the range of measurements is -10 to +40 then find out what the resistance is at 15 degrees. This is where the measurements will be most sensitive to change in temperature.

2.20.3. **The full conversion process**





R1 = LDR in sunlight (400lux) Vin = 5V, R2 = 10k, Vref =1.1V

Which way around should our Vdiv circuit be?

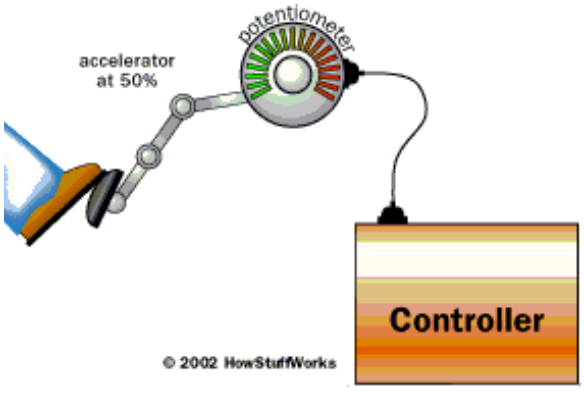
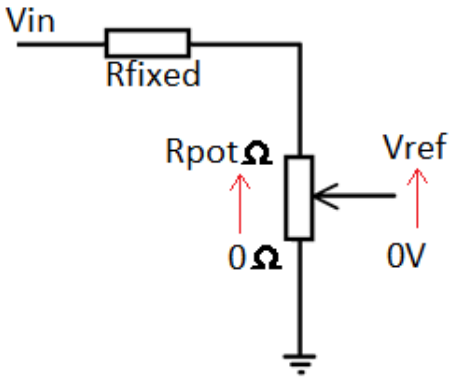
The value converted by the ADC will be:

Solve problems of dynamic range and quantization when employing analog sensors

Mostly we need to solve the voltage divider so that we have a useful range of values
In the last exercise we converted lux to a number in the ADC using a fixed value of R of 10K
Complete the table and plot the ADC voltage on the graph (scale RHS

LUX	R LDR (from graph)	VDiv out (R=10K, Vin=5V)	ADC (ref=1.1V)
10	90k	4.5V	1023
20	5k	1.7V	1023
30	3k5	1.3V	1023
40	2k5	1V	930
50	2k	0.83V	775
100	1k02	0.46V	430
400	300	0.1456	135
700	190	0.083V	86
1000	140	0.069V	64

How will we convert the non-linear exponential ADC readings of Lux to actual Lux values in a variable, that we could display? The AVR has no floating point processor so we could use the math library to add floating point capability
however that might use up a lot of flash
Is there a way we could do it with integer arithmetic?

Another Situation	Unknown R value
	
Car accelerator potentiometer Vin = 12V, Vref = 1.1V, RPot = 47k R fixed = ??	

ADC LSB voltage or step size (sometimes called voltage resolution)

This is the minimum change in voltage required to change the ADC output by 1

ADC value = 75, Vref =1.1V	Vin =	LSB voltage
ADC value = 76, Vref =1.1V	Vin =	

ADC value = 75, Vref =5V	Vin =	LSB voltage
ADC value = 76, Vref =5V	Vin =	

Which step size above is 'better'?

'better' relates to context and the system being fit-for-purpose. In an abstract or de-contextualised example like this there is no 'better' or 'best'.
We need to know the situation that an ADC will be used in to make a decision about what are the 'best' or minimum attributes for it.

2.20.4. **Quantization error**

In the above examples input voltage levels that are in between the two values e.g. 0.080677V or 0.08106V cannot be accurately represented. In an ADC with a 5V reference an output value of 0 means the input voltage was somewhere between 0V and 4.89mV

ADC conversion is an approximation or rounding process. This means there is error in the quantization process and this quantization error is the difference between the actual sampled value and the quantized value.

The quantization error is what relationship to the two values?

To reduce the error

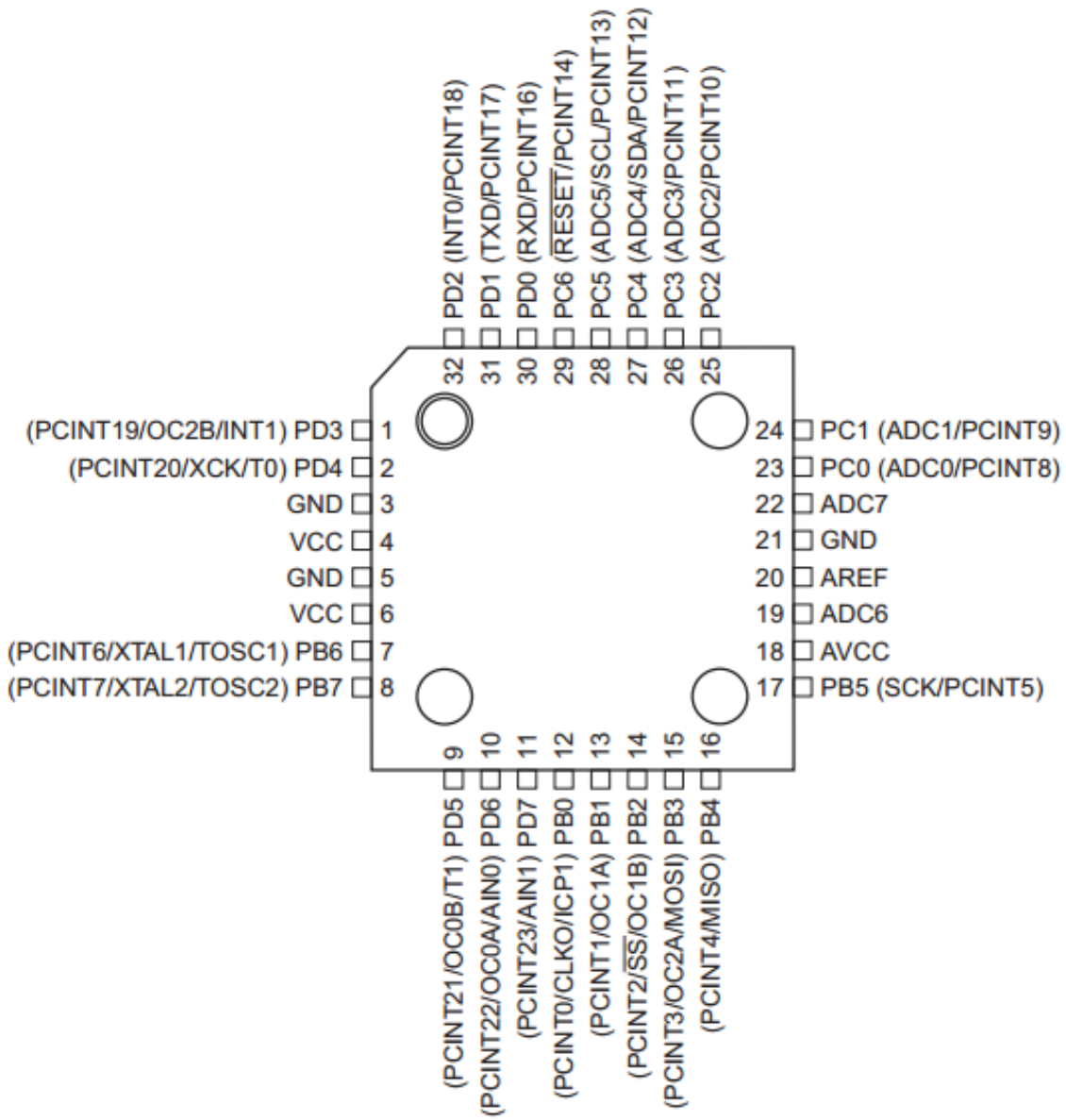
1. we can use an ADC with higher resolution (number of levels/bits)

2. we can reduce Vref

ADC converters become more expensive and complex as features such as resolution (number of bits) are increased. And circuits become more susceptible to noise with lower input voltage ranges, so changing these things is not easy. Atmel engineers and product people must have decided on a feature set at some stage that was suitable for a wide range of users and uses.

2.20.5. **AVR ADC inputs**

The AVR ADC has 8 channels (inputs) available for measurements external to the uC



ADC channels
ADC0 (PC0)
ADC1 (PC1)
ADC2 (PC2)
ADC3 (PC3)
ADC4 (PC4)
ADC5 (PC5)

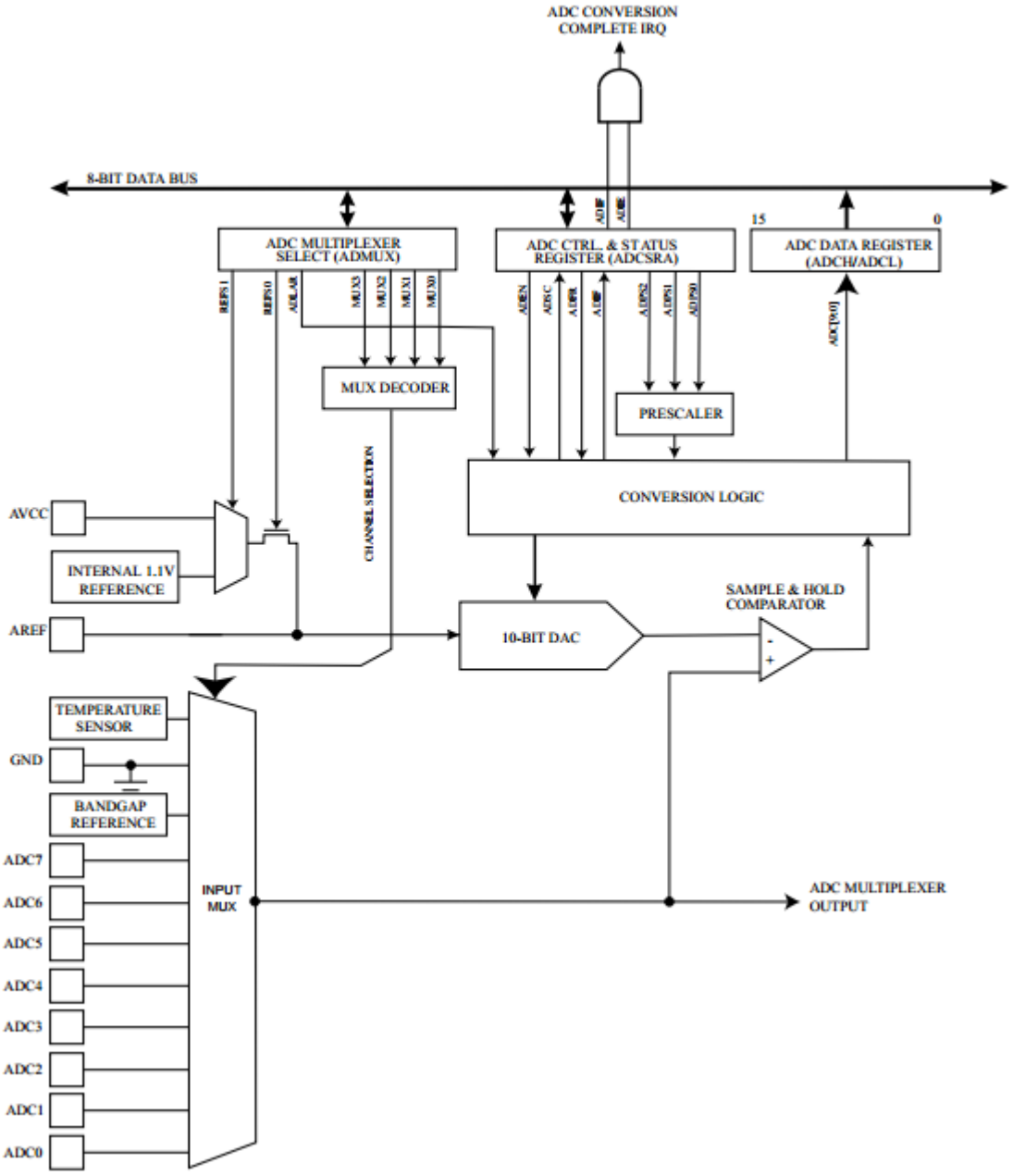
ADC6 (not an I/O pin)
ADC7 (not an I/O pin)

2.20.6. **ADC Channel select**

The ADC in the AVR's have multiple ADC pins but only 1 ADC converter. A multiplexer is used to select which channel a conversion will be carried out on. Channel selection is made via the ADMUX register

Although there are 8 external ADC channels on the ATmega328P (M1A package) the MUX is capable of 16 selections. Three of the other 8 selections are internal:

Figure 28-1. Analog to Digital Converter Block Schematic Operation



The selection of which channel is to be read from is made in the ADMUX register

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
(0x7C)	**REFS1**	**REFS0**	**ADLAR**	**–**	**MUX3**	**MUX2**	**MUX1**	**MUX0**
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Write the command to select channel 5 for an ADC conversion.

ADMUX | = 0x05 might or might not work. Why?

Write commands to setup the ADC to read from the internal; temperature sensor

2.20.7. **Sampling rate**

The process of making a conversion inside an AVR is via the method called successive approximation, it is reasonably complex involving 10 comparisons so it cannot happen at the full clock speed of the main CPU clock (F_CPU). To give ADC circuits the time needed for conversions they are clocked at rates lower than F_CPU and the actual conversion takes 13 cycles of this lower clock speed

> By default, the successive approximation circuitry requires an input clock frequency between 50kHz and 200kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200kHz to get a higher sample rate.

The lower clock rate is provided by the ADC prescaler which is setup using three bits in ADCSRA.

Figure 28-1. Analog to Digital Converter Block Schematic Operation

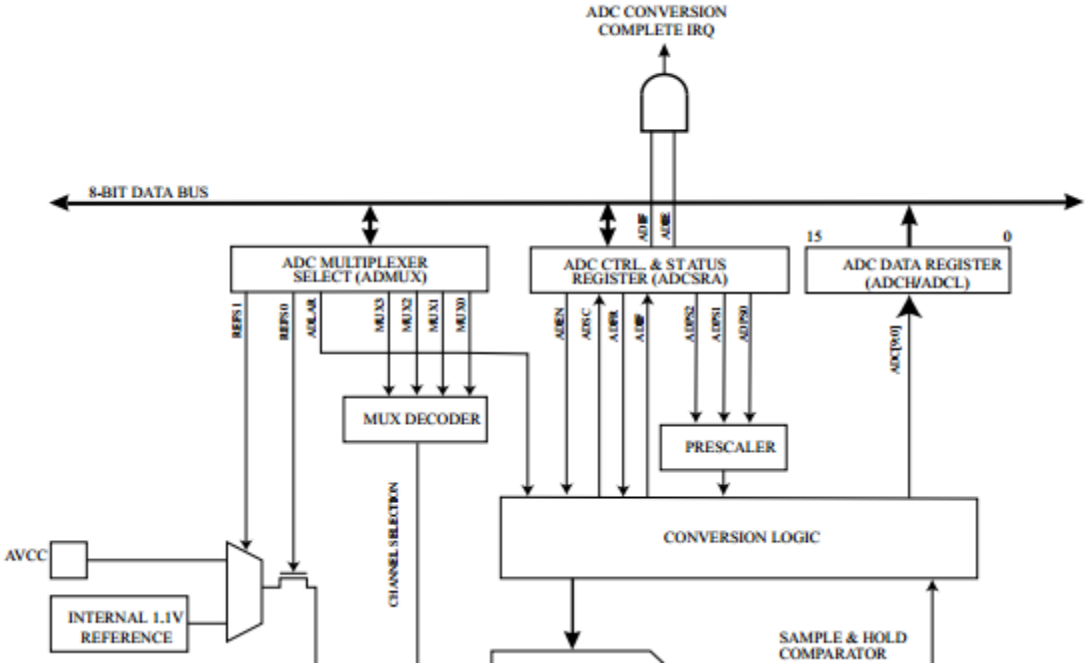


Table 24-5. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

If F_CPU = 10MHz, then what would be all the possible values of the ADC prescaler and which would be preferred?

- 10,000,000 / 2 = 5,000,000
- 10,000,000
- 10,000,000
- 10,000,000
- 10,000,000
- 10,000,000
- 10,000,000
- How long would a conversion take at

2.20.8. **ADC modes**

There are two modes for ADC conversions: single conversion and free running. In addition an ADC interrupt can be setup so that an ISR is called when a conversion is completed. We have 4 options.

Single conversion mode Non-interrupt	Single conversion mode With interrupt	Free running mode Non-interrupt	Free running mode With interrupt

ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0
(0x7A)	**ADEN**	**ADSC**	**ADATE**	**ADIF**	**ADIE**	**ADPS2**	**ADPS1**	**ADPS0**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Datasheet	notes
Bit 7 – ADEN: ADC Enable Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress will terminate this conversion.	
Bit 6 – ADSC: ADC Start Conversion In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC. ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.	
Bit 5 – ADATE: ADC Auto Trigger Enable When this bit is written to one, Auto Triggering of the ADC is enabled. The ADC will start a conversion on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in ADCSRB.	
Bit 4 – ADIF: ADC Interrupt Flag This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.	
Bit 3 – ADIE: ADC Interrupt Enable When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.	

Table 28-6. ADC Auto Trigger Source Selections

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match A
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

(ADCSRB)

code to trigger ADC on Timer/Counter0 overflow _____

2.20.9. **ADC resolution**

Do we actually need all 10 bits of resolution?

If we can use 8 bit resolution then we can set ADLAR (ADC left adjust) and just read ADCH

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

When an ADC conversion is complete, the result is found in these two registers.

ADLAR is 0 by default

unit16_t adc_result = ADCW; // 1023 = 0b00000011 11111111 ADCW get both bytes together

ADMUX |= (1<<ADLAR); //SET ADC to left adjust conversion into ADCH

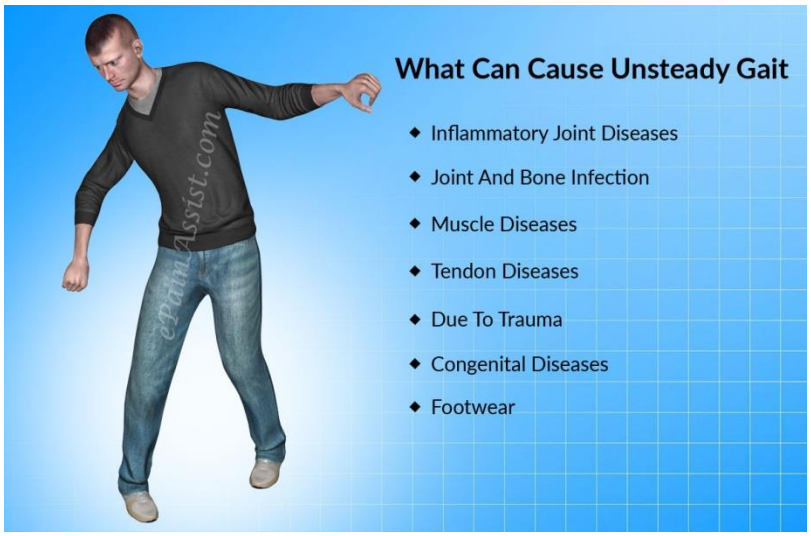
uint8_t adc_result = ADCH; //1023 = 0b11111111 11000000 which we read as 0b11111111(255)

2.20.10. ADC Example 1

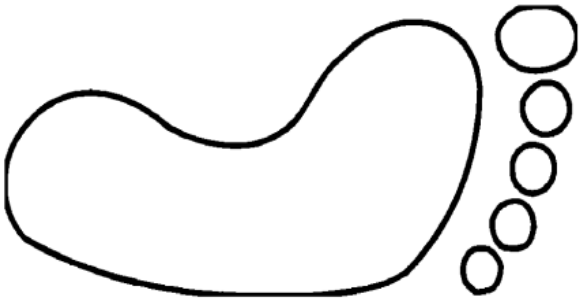
Single conversion mode Non-interrupt	Free running mode Non-interrupt	Single conversion mode With interrupt	Free running mode With interrupt
Setup prescaler Setup channel ADC Enable Start Conversion Loop while ADSC is high When low read ADC value Change channel if reqd	Setup prescaler Setup channel ADC Enable Start Conversion Monitor ADIF When low read ADCW Clear ADIF (write 1 to it)	Setup prescaler Setup channel ADC Enable Write ISR{ - change channel if req'd - process ADC value} Enable ADIE Setup ADTS Start Conversion	Setup prescaler Setup channel ADC Enable Start Conversion
requires polling and is blocking	requires polling and is blocking	No polling or blocking	No polling or blocking

We will choose one of these options for our example

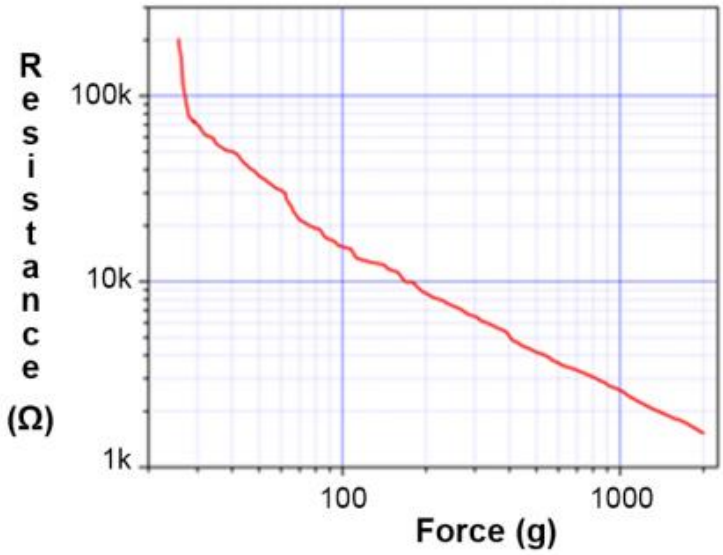
Corrective gait device: a person has an issue with their gait (manner of walking), we want to give them immediate feedback while they are walking to help them recognise and correct their gait.

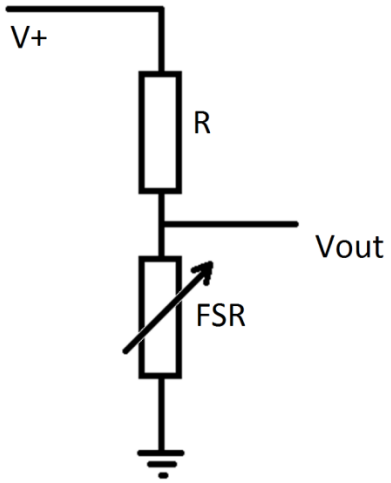


How many sensors would be required in the shoe and where would they be?



13mm diameter FSR
Interlink Electronics
($10 each for quantities of 1)

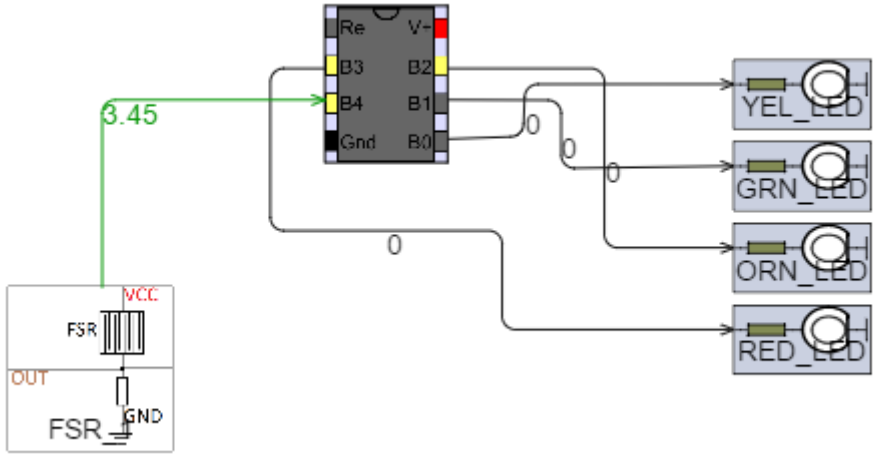




V+ = 5V, Vref = 2.56V, R= 10K

Force(gm)	FSR(R)	Vout	ADC out
0	>100k		
50			
100			
500			
1000			

FSR test setup

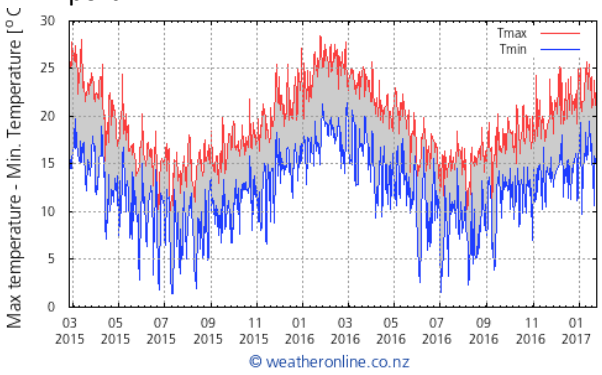
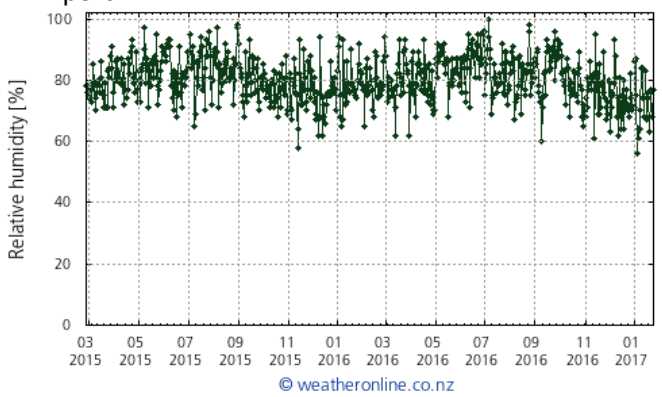
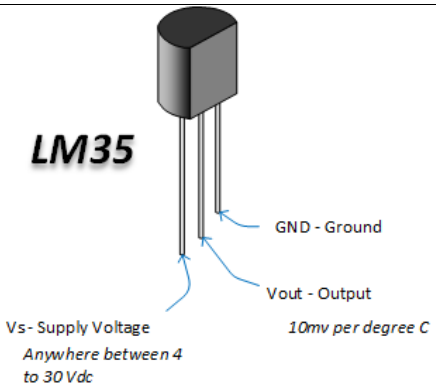


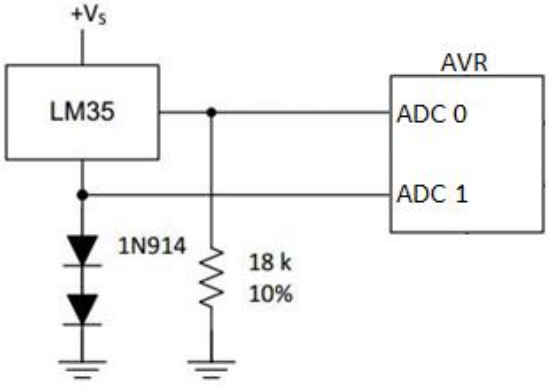
```
#define F_CPU 8000000//crystal
/***** Includes *****/
#include <avr/io.h>
//Hardware macros
#define SET_RED_LED   PORTB |= (1<<PB3)
#define SET_ORN_LED   PORTB |= (1<<PB2)
#define SET_GRN_LED   PORTB |= (1<<PB1)
#define SET_YEL_LED   PORTB |= (1<<PB0)
#define FSR_1 2     //macro to refer to ADC channel

void all_leds_off(){
    PORTB &= 0xF0;
}
void init_ADC() {
    ADCSRA |= ( (1 << ADPS2) | (1 << ADPS2) ); //8MHZ prescaler to 64
    ADMUX |= ( (1<< REFS2) | (1 << REFS1) );  //Vref 2.56V internal nocap
    ADCSRA |= (1 << ADEN); //Power on the ADC
    ADCSRA |= (1 << ADSC); //Start initial conversion
    ADMUX |= (1 <<ADLAR); //left adjust ADC reading into ADCH
    //ADATE is not set so system will be in single conversion mode
}
uint8_t read_adc(uint8_t channel) {
    ADMUX &= 0xF0; //Clear previously read channel
    ADMUX |= channel; //Set to new channel
    ADCSRA |= (1 << ADSC); //Starts a new conversion
    while (ADCSRA & (1 << ADSC)); //BLOCK until the conversion is done
    return ADCH;
}
int main(void) {
    init_ADC(); //setup the ADC to work
    DDRB = 0xff; //set all pins initially as outputs
    DDRB &= ~(1 << PB4); //adc 2 is PB4 on an ATtiny85 make an input
    uint8_t fsr_1_val = 0; //declare and initialise value for adc readings
    while (1) {
        fsr_1_val = read_adc(FSR_1);
        fsr_1_val = fsr_1_val >> 6; //reduce range from 8 bits to 2
        all_leds_off();
        switch (fsr_1_val){
            case 0:  //FSR reading of
                SET_RED_LED;
                break;
            case 1:  //FSR reading of
                SET_ORN_LED;
                break;
            case 2:  //FSR reading of
                SET_GRN_LED;
                break;
            case 3:  //FSR reading of
                SET_YEL_LED;
                break;
        }
    } //end while(1)
} //end of main
```

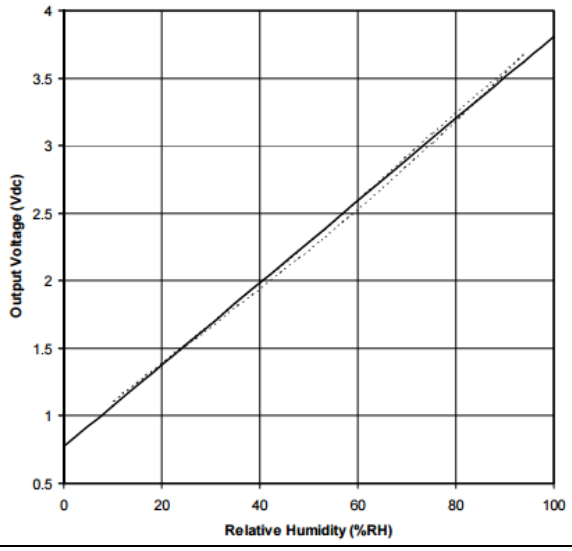
2.20.11. **ADC Example 2, Logging environmental parameters**

LO: describe the ES as reactive and responsive to its environment

Not all analog sensors require being part of voltage divider circuits; some sensors have these built in and provide a linear transfer function between the environmental source and output voltage as with the two sensors used here.

<table>
<tr><td>Temperature</td><td>Humidity</td></tr>
<tr><td>Ak Airport
 
</td><td>Ak Airport
 
</td></tr>
<tr><td>Range required: -10 to + 40 degC</td><td>Range required: 50% to 100%</td></tr>
<tr><td> 
</td><td> 

HIH-4030 humidity sensor
Sparkfun.com</td></tr>
<tr><td> 

-55 to + 150 degC
10mV per degC
To get negative values this circuit is recommended, then subtract the two readings to get temperature</td><td> 
</td></tr>
</table>

Where the set of data contains multiple values the data can be individually stored or they can be grouped in a struct. Here a struct is declared (env_log_s) and defined as a type(env_log_t)

```
typedef struct env_log_s {
    uint8_t tempr_h;
    uint8_t tempr_l;
    uint8_t humidity;
} env_log_t;
```

The data will be displayed on a graphics display which is 192 pixels wide and 64 pixels high, the available area for the graph will is 165 pixels, so an array of size 165 is required to store the data

```
env_log_t env_logs[165];
```

```
the array will need to be written to and read from, so two global
variables were declared and initialised
```

```
volatile uint8_t write_address = 0;
volatile uint8_t read_address = 0;
```

An early prototype is here

In this system a timer interrupt is designed to read the sensors at the required interval

e.g. daily hourly or by the minute.

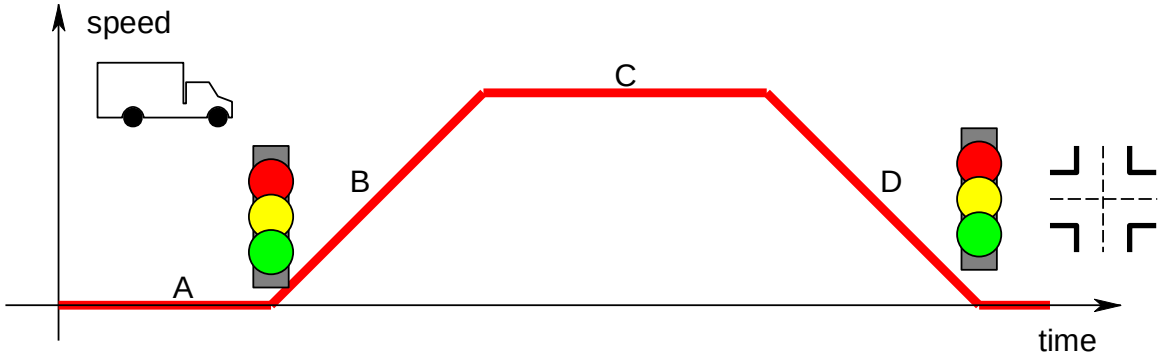
2.21. State machine programming

Often students learn programming using flowcharts as planning tools; however flowcharts are unable to express the complex set of behaviours and reactions required in embedded systems.

Think of the truck in the graph below as having 4 states.

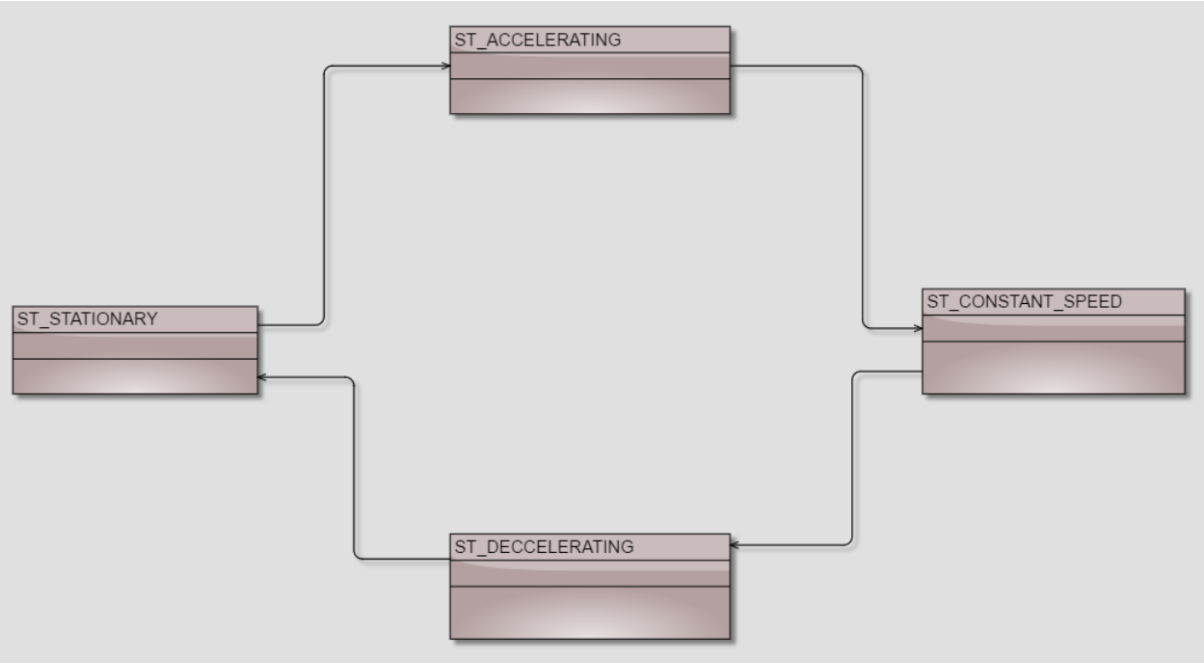
- stationary
- accelerating
- constant speed
- decelerating

A graph of velocity against time can be used to understand the four states and to consider the inputs and actions that are required to manage the system



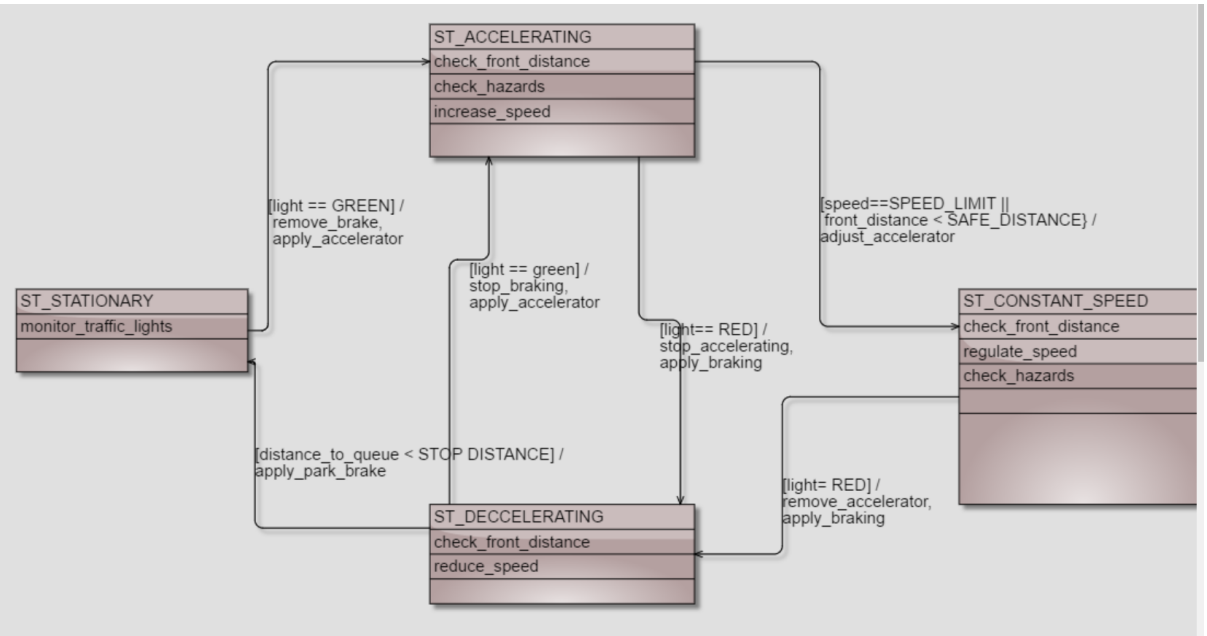
vehicle pulls away from the traffic lights a set of inputs must and outputs conditions exist and a complex set of behavioural choices are called for.

A first state diagram could look like this



What happens if the truck has not reached constant speed and the driver sees a red light ahead?
The state machine code needs to be able to transition from any state to any other state, at the moment it cannot. However this is easily extendable to add transitions and actions

The power of a state machine diagram can be seen in the situation where the truck is able to react to changes in the environment and **transition** from one state to any other state depending upon **conditions** such as inputs or calculations within the system.



The keywords are:

- state
- transition
- condition
- action

Mealy and Moore are two types of state machines, what is the difference?

What is this type?

A large number of state machine examples can be found on the internet, including:

- Pacemaker
- Stopwatch
- Calculator
- Greenhouse
- Traffic lights

2.21.1. **Coding state machines in C**

Coding state machines into programming languages is all about clearly identifying and laying out the code to reduce the spaghetti, making it easier to identify sources of bugs and to add new states, transitions and actions.

One common method of coding smaller state machines is using the switch, this can also be coded using an if-else or while loop pattern. A more comprehensive pattern uses a separate function for each state and calling the appropriate function using a function pointer.

```
#define EW 0
#define NS 1
#define PED 2

uint8_t state = NS;
```

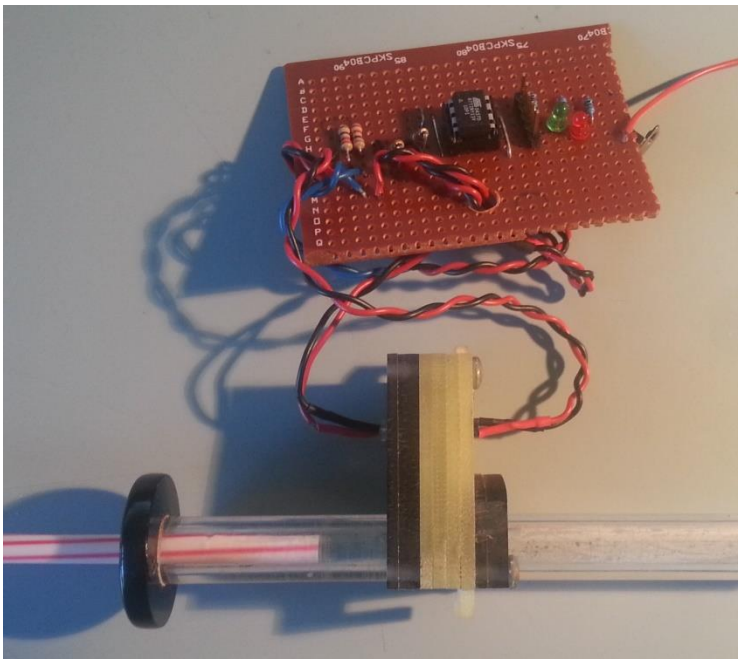
```
while (1){
   switch (state){
      case EW:
         //...
         if (PED_SW){
            state = PED;
         }
      break;
      case NS:
         //...
         if (PED_SW){
            state = PED;
         }
      break;
      case PED:
        //...
      break;
   }
}
```

```
while (1){
    if (state == EW){
       //...
       if (PED_SW){
           state = PED;
       }
    }
    else if (state = NS){
        //...
    }
    else if (state == PED){
        //...
    }
}
```

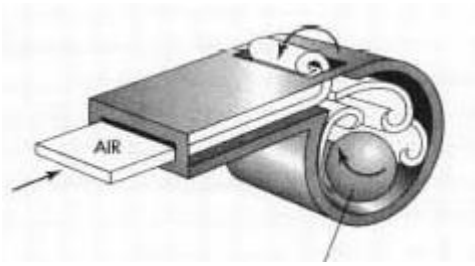
```
while (1){
    while (state == EW){
       //...
       if (PED_SW){
           state = PED;
       }
    }
    while (state = NS){
        //...
    }
    while (state == PED){
        //...
    }
}
```

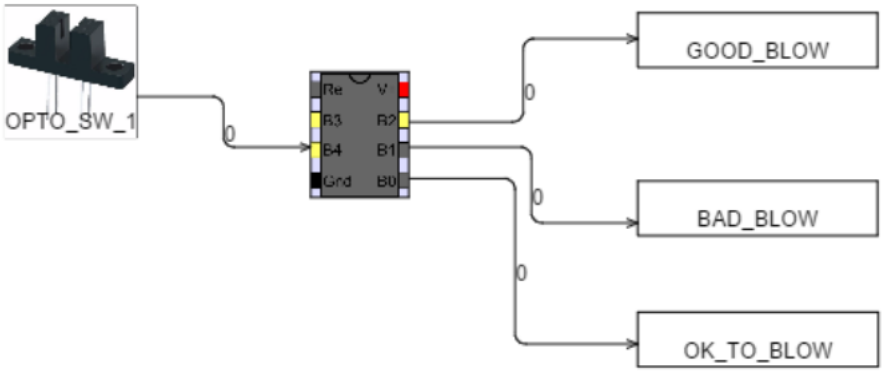
2.21.2. **Breath tester state machine example**

This device is used to detect excess alcohol intake by having someone blow through a straw onto an analog alcohol sensor. The device however can give unreliable readings if the person does not blow long enough or hard enough into the device. An addition to the device was designed to capture the length of the blow and amount of air blown. Inside the plastic block is a small ball which rotates around inside. An Infrared LED and an infrared sensor are mounted opposite each other, and when the pea rotates around inside the chamber the light path between the LED and sensor is broken. Speed can be calculated using a timer.

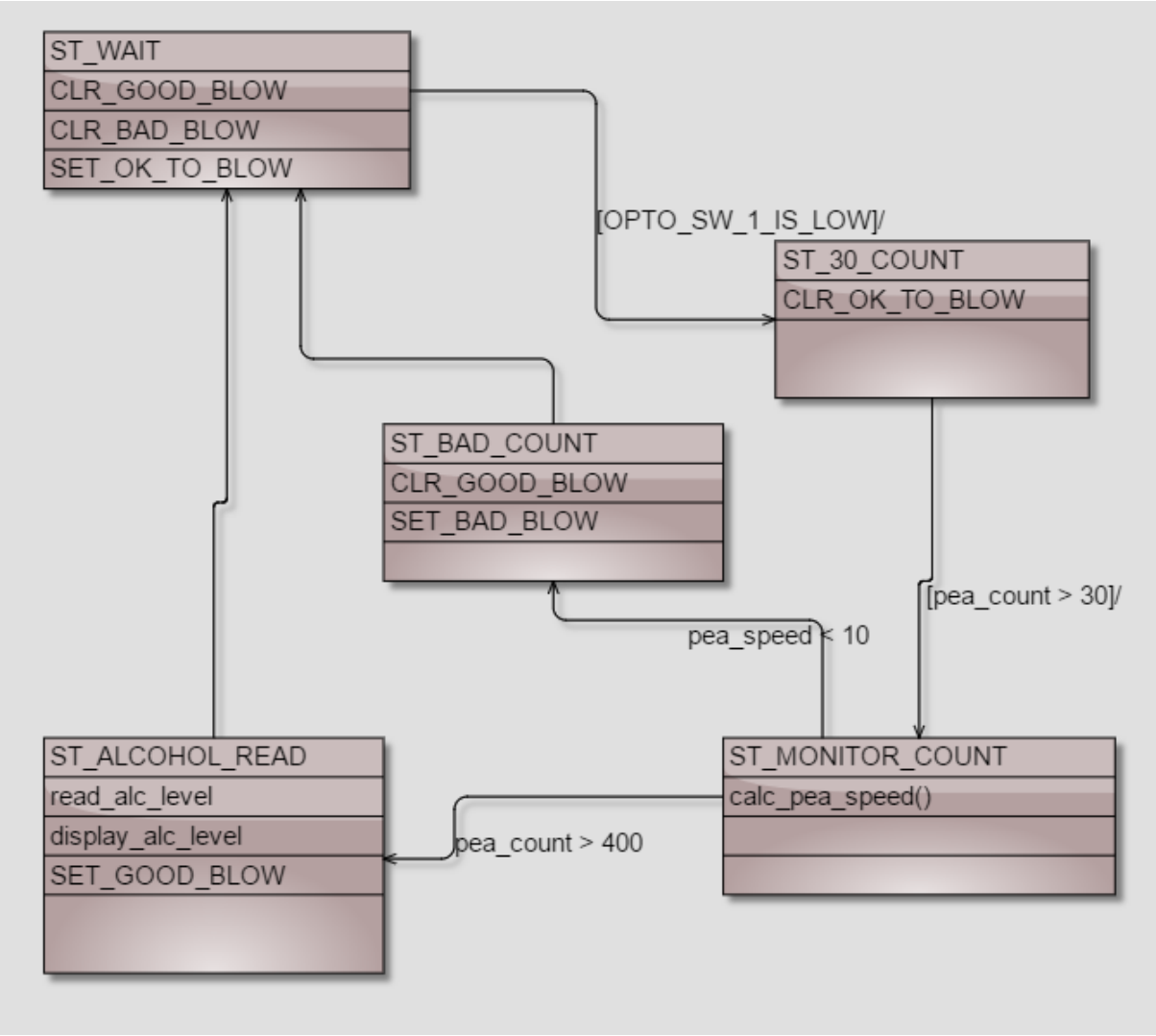


The mechanism is very similar to a 'pea' or referee's whistle



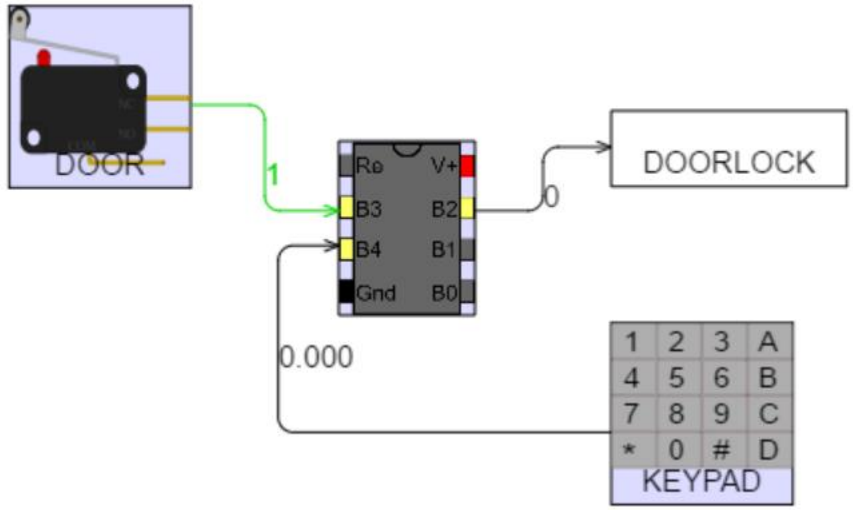


Here is an initial state machine for the system



This has a number of problems; can you identify some of them?

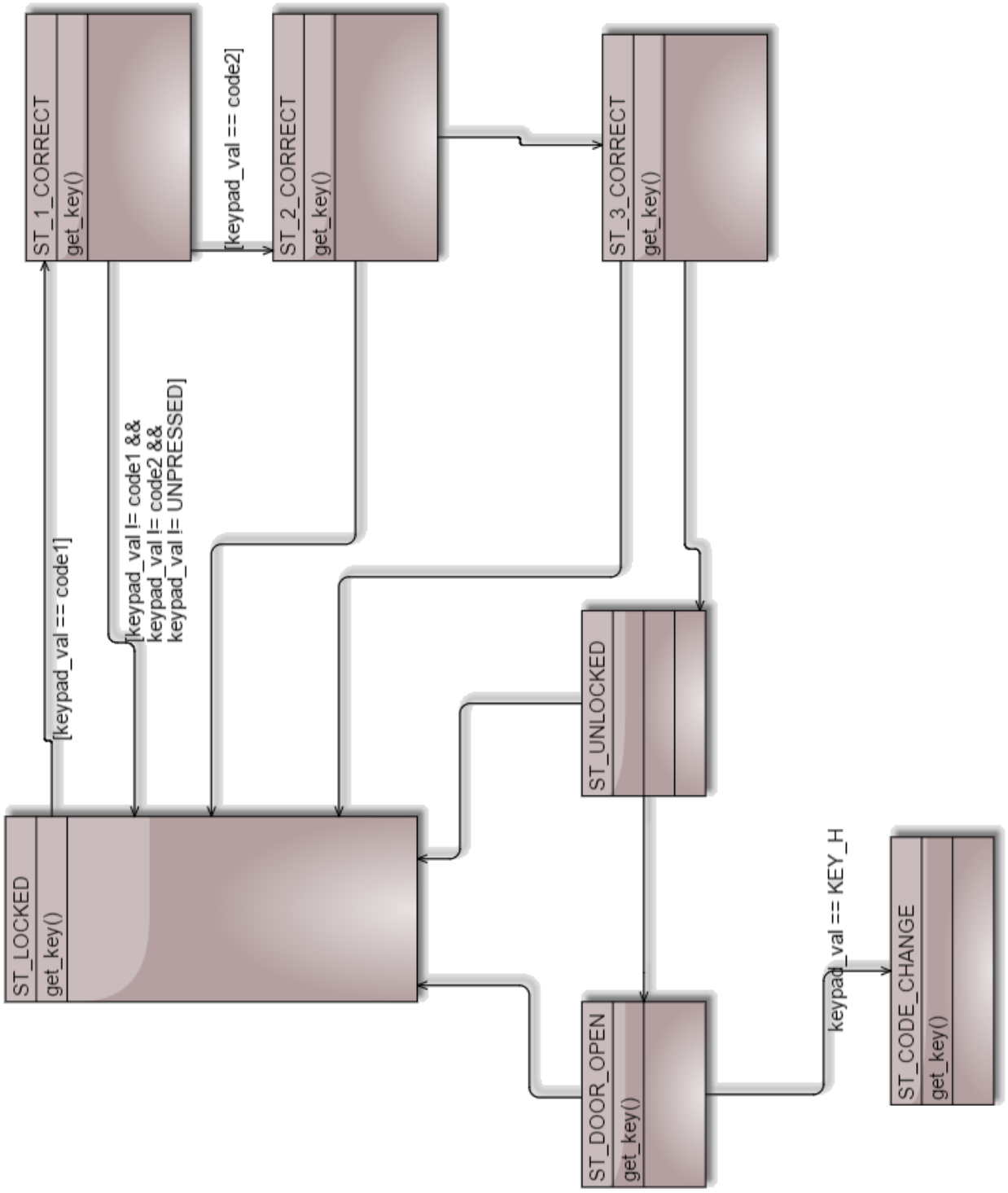
2.21.3. **Electronic safe state machine**



Operating the safe:

- Enter the correct 4 digits in the keypad
- If the code is correct the door unlocks
- If the door is unlocked and remains closed for 20 seconds it automatically locks
- If the door is open and then closed it locks
- If the door is open and the A key is pressed a new code can be entered
    - While entering a new code if C is pressed the new code is cancelled
    - After entering a new 4 digit code if D is oppressed the new code is saved
    - if the door is closed before the new code is saved or cancelled the new code is ignored

```
/***** Globals *****/
uint8_t code1 = 1;
uint8_t code2 = 2;
uint8_t code3 = 3;
uint8_t code4 = 4;
uint8_t state=0;

/***** Functions *****/
void unlock;
   CLR_LOCK;
}
void lock(){
   SET_LOCK;
}
void get_key(){
   keypad_val = read_adc(KEYPAD);
}
```



Does this code express every possible combination of the logic for exiting ST_1_CORRECT?

```
while (state == ST_1_CORRECT) {
        get_key();
        if (keypad_val != code1 && keypad_val != code2 && keypad_val != UNPRESSED) {
           state = ST_LOCKED;
        }
        if (keypad_val == code2) {
           state = ST_2_CORRECT;
           wait_for_unpressed();
        }
    }
```

2.22. Fitness-for-purpose

LO: understand 'fitness-for-purpose' in guiding microcontroller choice for an ES

Understanding the nature of an ES (the interrelatedness of hardware and software, the ES as an unattended automaton, the ES as reactive and responsive to the environment) allows the enginer a mindset of what will make the system fit-for-purpse. When choosing a microcontroller for an ES this mindset focuses us on the task the system is fulfilling and makes it possible to select an approproate device with the required feature set that is cost-effective. avoiding *'using a sledgehammer to crack a nut'*

The criteria for measuring fitness for purpose include:

- Device speed
- Power supply - battery or existing power supplies
- power consumption – being able to put a uC into some form of redued power or sleep mode is essential in reducing power
- Package (DIP, QFP) – size and assembly considerations
- Cost per unit and quantity needed
- Bus size – 8/16/32/64
    - The size of calculationsare floating point calcs required
- Memory
    - SRAM – how much data / how many and what calculations need to be processed
    - Flash – the size of the program – will extra features be added later
    - EEPROM – what non-volatile data needs saving
- Number of I/O pins
    - How many devices/peripherals – how many ADC channels
- DC characteristics -
- number and trigger type of external Interrupts
- number, size and capability of timers
- Tools and libraries available for development
    - Having to write libraries of functions for peripheral devices is not easy and can be avoided by choosing a uC/manufacturer who provides this support – the 8 bit AVRs are well supported – as are PICs and some others
    - Debug features are important for developers in cross compilation environments
- Supplier capability and longevity
    - Choosing a manufacturer that is committed to supporting their products in the long term, where new products are backward compatible with old ones
- Software security features
    - Can someone read your code out of the uC and steal your intellectual property
- Expansion capability,
    - Having special hardware interfaces built in like : UART, CAN, I2C, 1-wire,… built in
- Other features
    - Low voltage (brown-out) detection is common
    - watch dog detector

In your learning to date you should understand some of these and be able to explain them

2.22.1. **Devices characteristics**

Input or Output device?

	Device	Input or Output	Use
1	Thermistor		
2	LCD		
3	GPS receiver		
4	Accelerometer		
5	Microswitch		
6	LDR		
7	Humidity sensor		
8	Potentiometer (pot)		
9	Solenoid –electromechanical		
10	Solenoid valve		
11	Strain gauge		
12	FSR		
13	DC Motor		
14	LED		
15	7 segment display		
15	Photo diode		
16	keypad		

2.22.2. **Driving output devices**

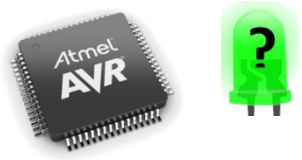
LO: list specifications of an uC that would inform choosing a uC for an ES

LEDs (Light Emitting Diodes)

Will it work?

Will it work?

Will it work?



How can we know?

we need to know the characteristics/specifications of the devices

before we can engineer a solution

LED Specifications:

Kingbright, super bright Red BL-BD-BF43R7M 4000mcd

Vf typical:2.0V at 20mA

this means it will draw 20mA when connected to 2.0 V,

When thinking about connecting an LED to a uC we need to know about both the uC and the specific board we are using.

A few ATmega328P Specifications from the datasheet (Page 2)

Operating Voltage:

Speed Grade:

- 0 - 4MHz@
- 0 - 10MHz@
- 0 - 20MHz @

Could I design a circuit that works at?

- 4V and 16MHz? 7V and 12MHz? 1.8V and 4MHz?

Could I over-clock an AVR?

Should I design a circuit that uses an over-clocked AVR?

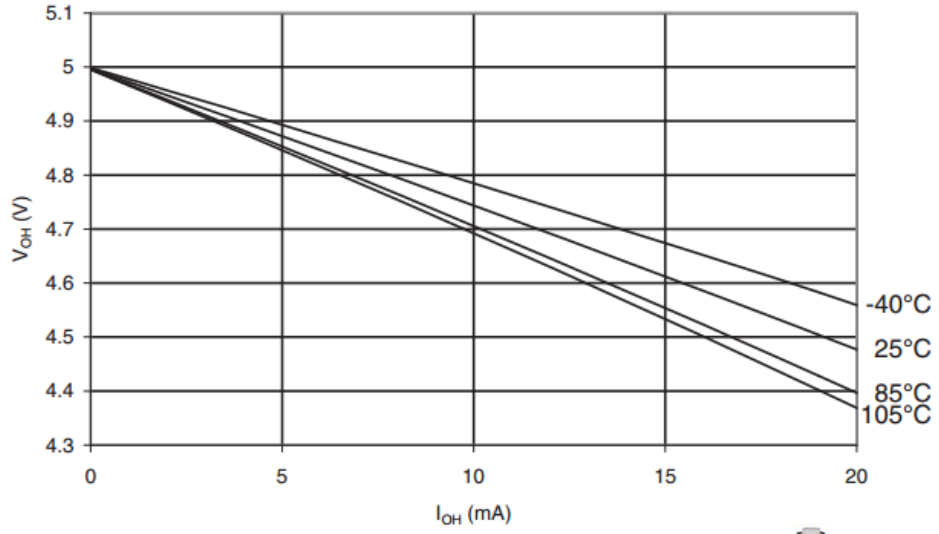
ATmega328P DC characteristics for -40 to +85 degC datasheet section 32.1

Absolute Maximum Ratings

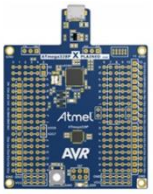
- DC Current per I/O Pin
- DC Current Vcc and GND pins

Although the absolute maximum specs are 40mAthe datasheet gives a graph of output voltage for different frequencies and output currents, that only goes up to 20mA. So connecting devices that would require more than 20mA would require some experimental investigation.

ATmega328P: I/O Pin Output Voltage vs. Source Current (V_{CC} = 5V)



Xplained Mini 328P development board specification



- Vcc =
- Crystal frequency =

This means that an LED cannot be connected directly between an I/O pin and ground

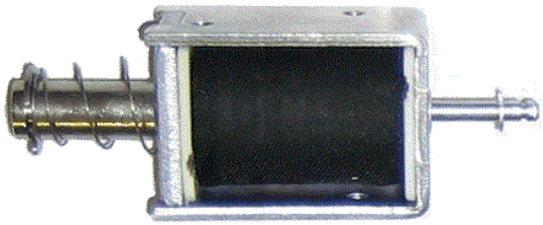
What is the output voltage of the pin on the Xplained mini board going to be if we connect an LED that draws 20mA to it?

How many LEDs could we connect and turn on at one time?

Look thoroughly through the datasheet to work out the current drawn by all the subsystems within the uC that you will be using and then use the maximum rating of 200mA that can be drawn through the power supply pins of the uC

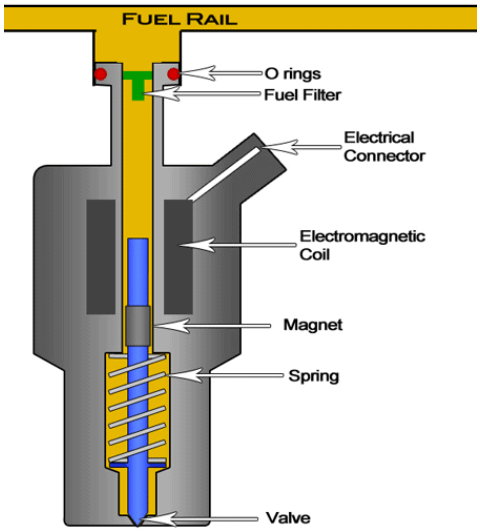
Driving devices with higher current / voltage requirements

Electromechanical devices such as solenoids and motors are commonly controlled by uC's



e.g. 12V 300mA

Solenoids become useful when they are connected to mechanisms such as those found in fuel control units of cars, production lines in factories and _____



A typical solution is to use a bipolar Darlington transistor or MosFET

2.23. Useful resources I have checked out as accurate

'Atmel AVR microcontroller primer programming and interfacing'
Barrett & Pack
An excellent resource on how to setup AVR timers, ADC, Serial communications
Download the full book from the UoA library – DO IT!!

TinyAVR microcontroller projects for the evil genius
Dhananjay V Gadre Nehul Malhotra
Has some interesting and novel project ideas

'Practical AVR Microcontrollers'
Trevennor has some interesting projects and excellent coverage of AVRDude
Download the full book from the UoA library

AVR Freaks – forum and tutorials
http://www.avrfreaks.net/forum/newbie-start-here
http://www.avrfreaks.net/forums/tutorials

De-bouncing switches
http://www.ganssle.com/debouncing-pt2.htm

Embedded C resources
volatile, const, static
http://barrgroup.com/Embedded-Systems/How-To/Efficient-C-Code
Safety (Therac-25)
http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html

Interrupts
http://www.avrfreaks.net/forum/tutsoft-traps-when-using-interrupts

Timers
http://www.atmel.com/Images/Atmel-2505-Setup-and-Use-of-AVR-Timers_ApplicationNote_AVR130.pdf
http://www.evilmadscientist.com/2007/quick-and-dirty-d-to-a-on-the-avr-a-timer-tutorial/
http://www.fourwalledcubicle.com/AVRArticles.php

- AVR-GCC and the PROGMEM Attribute
- AVR Programming Methods
- Interrupt Driven USART in AVR-GCC
- Modularizing C Code: Managing large projects
- Newbie's Guide to AVR Interrupts
- Newbie's Guide to AVR Timers
- Using the EEPROM in AVR-GCC
- Using the USART in AVR-GCC

State machine programming
https://barrgroup.com/Embedded-Systems/How-To/State-Machines-Event-Driven-Systems
http://codeandlife.com/2013/10/06/tutorial-state-machines-with-c-callbacks/

CISC/RISC
https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccis