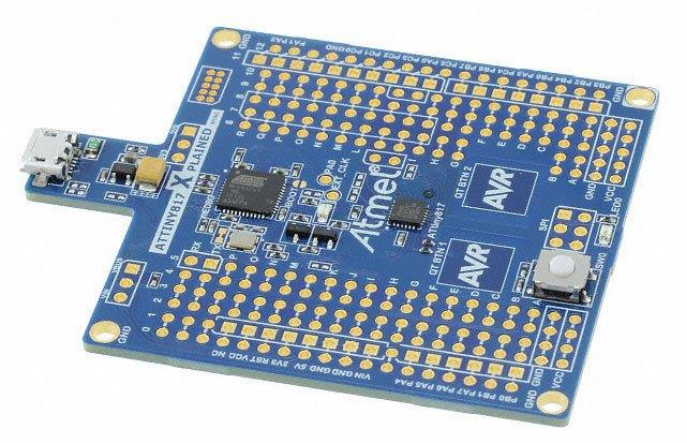
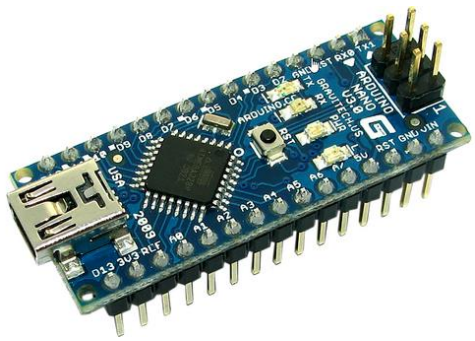
THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING





COMPSYS201 - 2021

Fundamentals of Computer Engineering

Part. 2 - Microcontrollers

Department of Electrical, Computer, and Software Engineering, the University of Auckland.

© Dr. Bill Collis 2021

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

Contents

© Dr. Bill Collis 2021

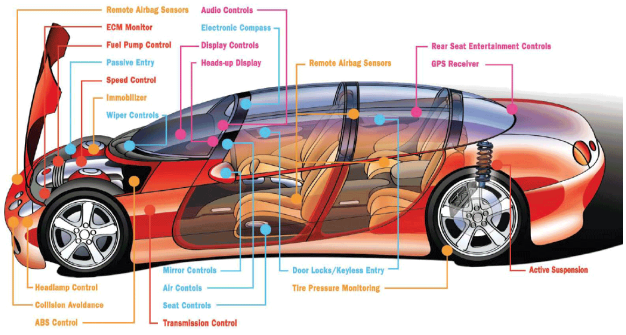
2.1.Introduction

Embedded systems (**ES**) are at the centre of the exponential increase of smart devices, the IoT (internet of things) and TinyML (embedded machine learning applications) happening 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. Common ES's have at their heart microcontrollers (**uC**) - other digital controllers include FPGAs, ASICs, and microprocessors. ES's are ubiquitous (everywhere) and so much so, that we are generally unaware of them. Over 250 billion of these devices automate, balance, maximise, minimise, regulate and control our everyday lives better than we can ourselves! A washing machine measures the volume and dirtiness 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 our toilet roll holders will reorder more paper for us before the roll has run out! While the number of smart phone and internet users are now growing at small rates every year, the number of ES is increasing by around 20% every year.

www.designtaxi.com

This part of the course is an introduction to uC's and their programming, and how as part of ES's they react to, and control, the environment. Because ES's come in so many varied forms, with so many different features and have so many diverse 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 ECSE 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
https://www.facebook.com/watch/?v=10152962127736021



www.microchip.com

2.2.Learning outcomes and success criteria

Course 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 (i.e.how registers work)
- with reference to a datasheet, write code that manipulates/checks single bits in any uC register

LO 2: understand the ES as an automaton

- describe three common C code variants used to make an ES into an automaton
- explain 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 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 an analog 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
- turn a state machine model into C code, using while / switch software pattern
- describe the serial protocol and its parameters used within the uC USART
- setup the uC USART registers

LO 4: understand the importance of high quality software practices

- apply a formal approach to writing program code through use of a template, functions and naming conventions that makes code transparent and therefore maintainable
- write comments in code that explain system function and not code syntax
- write single task self-contained functions

LO 5: recognise 'fitness-for-purpose' factors that guide 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 common variable types and their limitations
- explain over/under flow of variables and their implications
- interface an LED to a uC, taking into account the DC characteristics of each

LO 6: use an IDE

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

LO 7: develop as an engineer by working with complex tasks with multiple relationships





© Dr. Bill Collis 2021

2.3.The way of the program

We use mathematical models to express the dynamic relationships within the real world around us; whether this is the flow of charge, clouds, or continents. Programs are the ways of calculating these models using special languages (C, Java, Python…); and the computer is a numerical processing device that can interpret these languages and carry out computations of our models faster than we could do it ourselves. The embedded system takes this one step further, it allows us to measure changes in the real world as they occur, process these numerical representations using our models and control our world in real time.

2.4.Review your prior learning

Take some time to review the material below before this part of the course begins. This will mean you will be comfortable learning the new knowledge that is introduced.

EngGen131 review

Page 5: the difference between programs that are 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: Global and local variables
page 23: Casting
page 24: unary operators
page 25: special assignment operators and operator precedence
page 31: Arrays (basic types)
page 32: Boolean expressions
page 33/34: '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 as local variables
page 69-77: pointers and parameter passing

If it weren't for C we'd all be programming in Basi or obol

ElectEng101 review

This part of the course relies upon material from EE101, review sections

- 3.2.4 Number Systems
- 3.2.5 ADC
- 3.2.8 Embedded Systems

CompSys201 digital section review

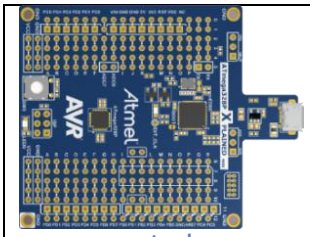
This part of the course relies upon the material in the first part of the course, review the material on:

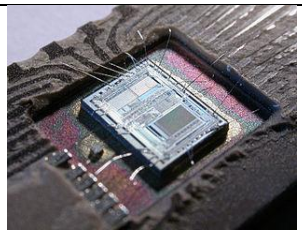
- Latch/Register
- Counters/dividers
- Comparators

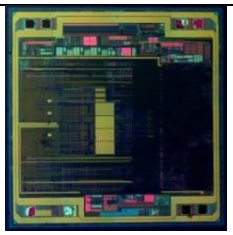
2.5. Atmel and the AVR range of 8 bit microcontrollers

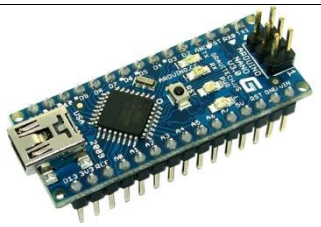
Atmel is one of many brands of microcontrollers. Other brands include: Texas, Cypress, Renasas, ST, TI, National, NXP, Freescale, and Microchip (who now own AVR) also make PICs

- ATMEL uCs represent about _____ of the world market share of 8 bit uC's – in excess of _____ per year are sold at an average cost of around _____ each
- The Atmel brand includes several different ranges of uC's - AVR8, UC3(32 bit) and ARM
- The 8 bit AVR range includes several sub types: ATtiny, ATmega, ATxmega, AT90
- Each sub type has several 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 Nano Board from aliexpress.com for about _____


www.atmel.com


www.wikimedia.org


www.zeptobars.com

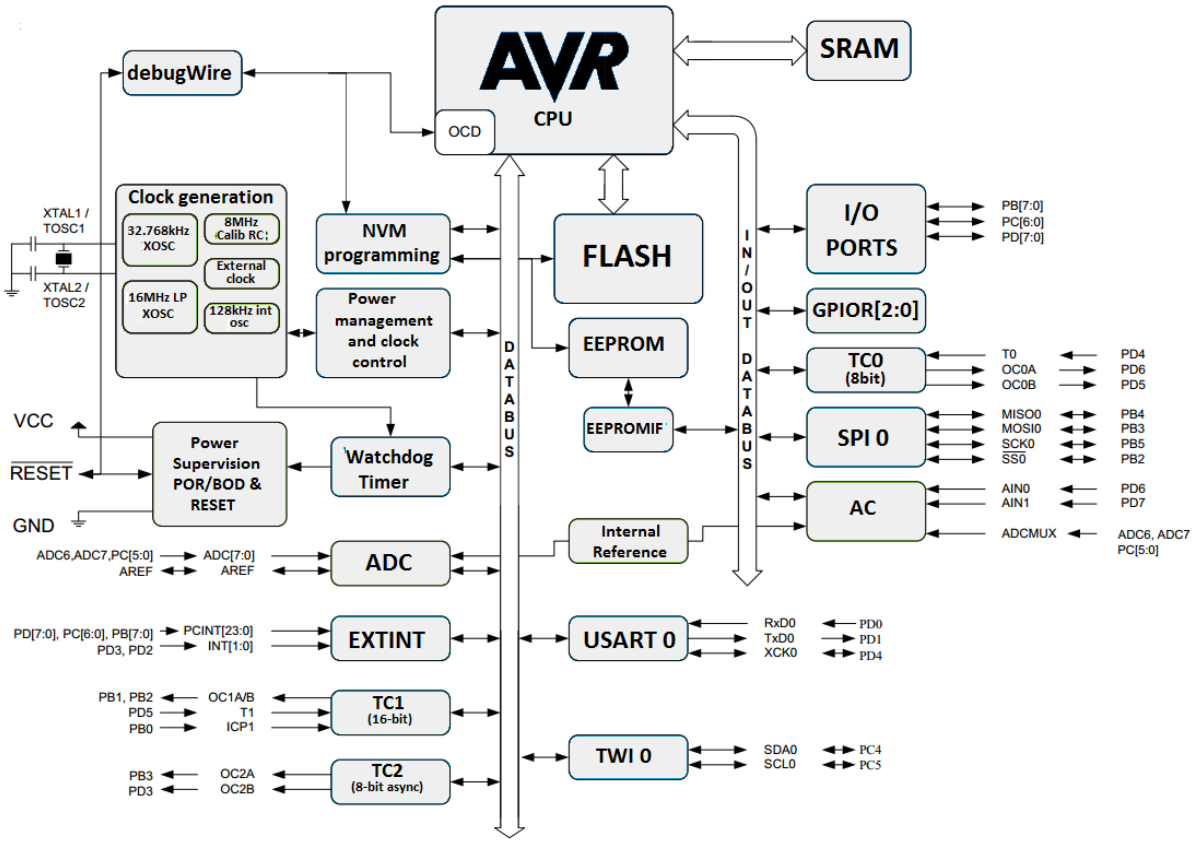

www.arduino.cc

In ElectEng 101 you were introduced to the ESP32 DevKit, we use the AVR in this course because it is less complex (it is still very complex) and is easier to learn about the inside workings of a Microcontroller.

2.5.1. AVR block diagram,

The AVR block diagram shows the CPU and the internal peripherals (parts of the chip that are not the CPU, and handle specific jobs) e.g. _____

Figure 4-1. Block Diagram



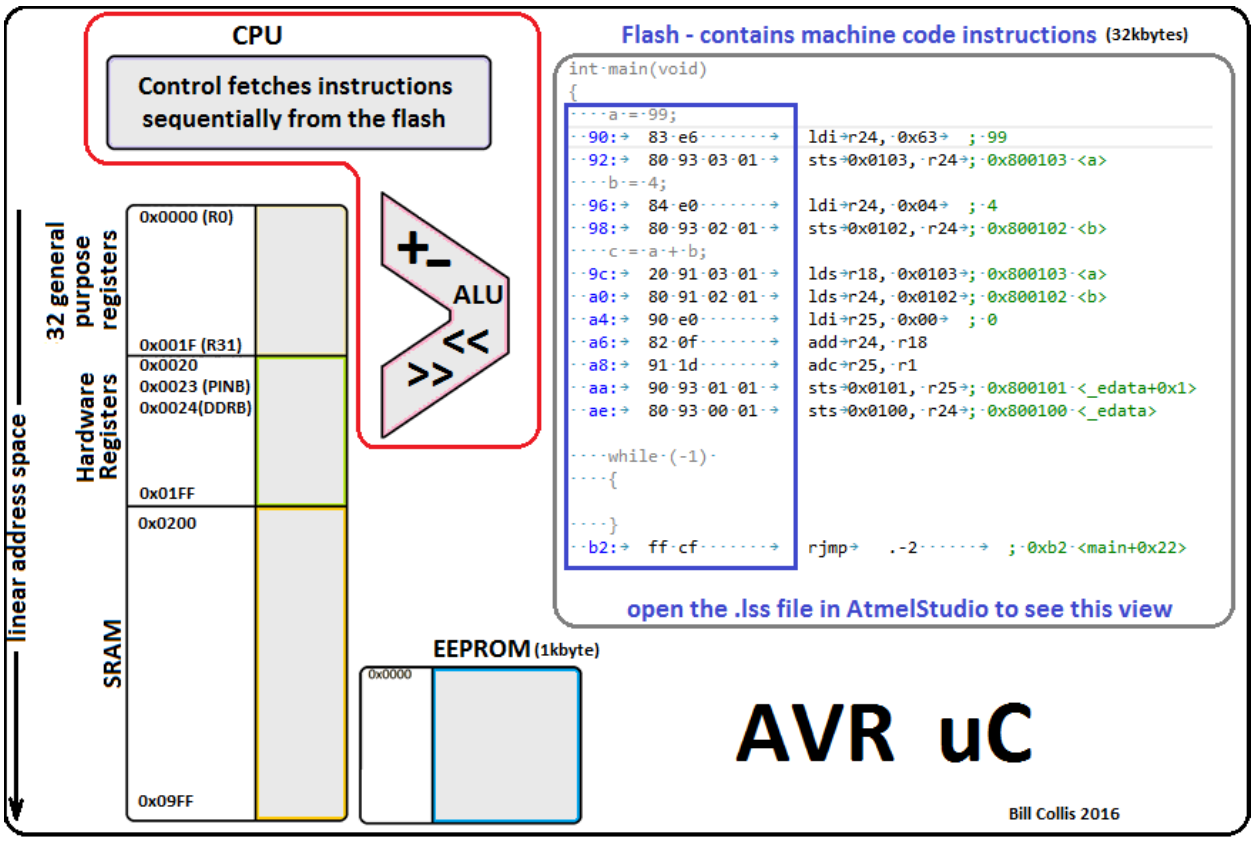
ATMega328 Datasheet

2.5.2. What is a computer/ microprocessor /microcontroller?

```
//C code example
uint8_t a = 0;
uint8_t b = 0;
uint16_t c = 0;
int main(void)
{
   a = 99;
   b = 4;
   c = a + b;
   while (1)   { … } //loop forever
}
```

Computers have microprocessors with external peripheral devices; such as ram, hard drive, displays etc. A microcontroller is a single computer, with a range of integrated peripherals, all inside one IC. Both a microprocessor and a microcontroller contain a CPU (central processing unit) which has a control unit and an ALU (arithmetic logic unit). Internal microcontroller peripherals include ADCs, Timers etc, and an array of **registers** to control them along with several different types of memory: SRAM, FLASH and EEPROM.

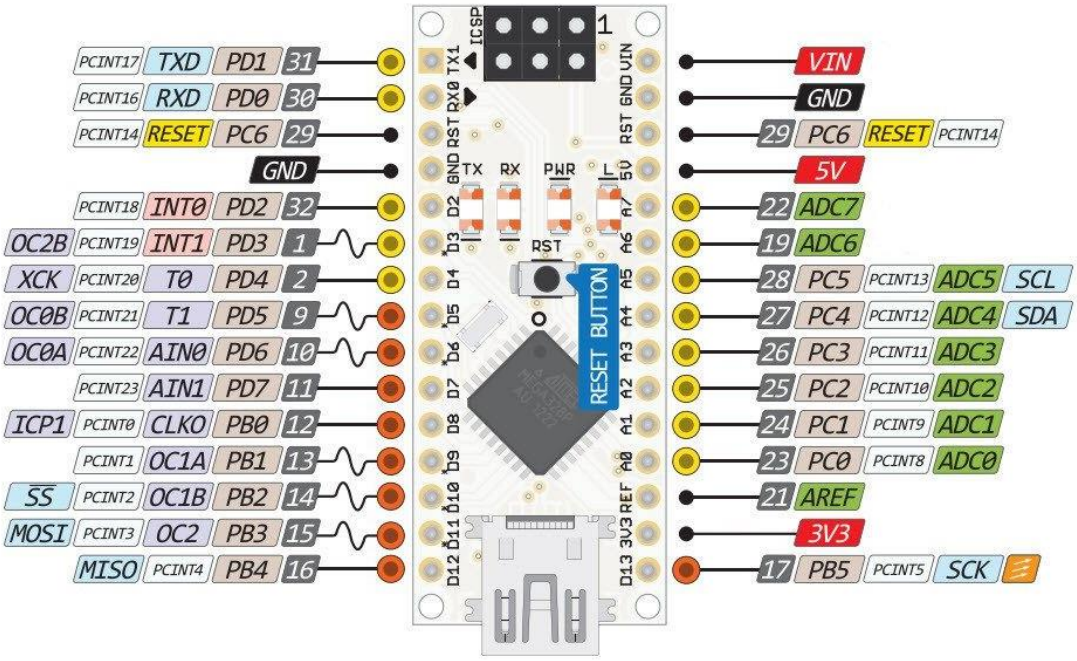
You need to know a few things: C programs are not stored in the uC – they are compiled into machine code and that is stored in the Flash. The human readable version of machine code is called assembler code or assembly language (assembler commands: ldi, sts, lds, add, adc are described in the datasheet). 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 work on registers so take only 1 clock cycle. Thus the AVR is a RISC (reduced instruction set computer) as opposed to a CISC (complex instruction set computer) where a single instruction may be more powerful containing multiple read, write and ALU operations, and take multiple cycles. The CPU control section fetches instructions sequentially from the flash (program memory) and decodes them. Take the simple program to add 2 numbers. The binary machine code stored in Flash (starting at address 0x0090) is: 0x83, 0xe6, 0x80, 0x93, 0x03, … The machine code can be viewed in the .hex file – however a more intelligible view of flash is found in the .lss file.



The AVR has a contiguous address space: the general purpose registers, the hardware registers and the SRAM are part of the same address space; 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 the same as the SRAM.

© Dr. Bill Collis 2021

2.5.3.The Arduino Nano development board



www.atmel.com

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

I/O PORTS – are made up of I/O (input/output) pins – places we can connect devices to:

I/P devices**:** _____

O/P devices**:** _____

ATMega328P has 3 I/O 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 development 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, input devices sense the real world, output devices control the real world. An initial aspect of using a pin is to set up which direction it will be; i.e. – input – to read an input device or as an output to control an output device.

ATmega328P datasheet: http://www.microchip.com/wwwproducts/en/ATmega328P

All datasheet references will be to the complete datasheet, we will use version DS40002061B

© Dr. Bill Collis 2021

2.5.4.AVR memory types

Describe the different memory types in 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) -_____
- Registers

- EEPROM – stores long term data
  - the ATmega 328P has a _____ EEPROM - _____

2.5.5.Number 'types'

www.gansle.com

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 RAM 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.

Memory types in C include int, unsigned int, short, long, float, etc. We do not often use these names when discussing uC programming; we use names for data types that 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 programs written for an AVR, they are 2 bytes. The limits of the types are:

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

Explain the difference between signed and unsigned?
What might be the implications of a bad variable choice?
What is the problem with using global variables in your program?
What is overflow/underflow? What sort of things could go wrong in a car engine control unit if an overflow/underflow occurred?

Guidelines for type selection:

1. know the boundary cases (limits) for your data use the smallest type possible, to…
2. use unsigned types where possible

© Dr. Bill Collis 2021

Here is a question from a survey on ES development from www.gansle.com. Should an engineer know the answer to a question like this?

*** 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 critical importance of social (non-technical) factors in 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.
http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html

One of the critical issues facing embedded systems engineers is the incorrect use of variables.

Explain over/under flow of variables

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

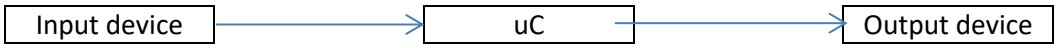
The fastest known speeding ticket was issued in May 2003 in Texas. The motorist was operating a Koenigseggs CCR, a Swedish sports car and was allegedly going **242 mph** (**389 km/hr**) in a 75 mph zone. The driver was arrested and his sport car was towed.

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

© Dr. Bill Collis 2021

2.5.6.External peripherals - inputs or outputs?

Research these devices, whether they are inputs or outputs and what they are used for.



	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		

© Dr. Bill Collis 2021

2.5.7. 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 usually 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*

```
//c programs begin executing at the main function
int main(void) {
    // setup code is placed here, it is only executed once
    …
    …

    //code in this while loop will run forever
    while (1) {     //
        …
        …
        …
    }    //jump back to the while and do the loop code again


    //your code will never get here

}
```

Research the alternatives to while(1){}

© Dr. Bill Collis 2021

2.6. Code Style

A design-pattern is a common approach to a common problem in software, an anti-pattern is the opposite, it is a poor programming practice (bad habit) that is ineffective and counterproductive (ultimately works against your efforts in the end).

Anti-patterns lead to software bugs. It is cheaper and easier to prevent bugs from creeping into your programs than it is to find and fix them once you realise they are there. Gansle says that 56% of coding work done is rewriting existing code (www.gansle.com). So the clearer and more readable your code is, the easier it will be to debug and modify - even if you are the only person who will ever read it.

> ***"Any fool can write code that a computer can understand. Good programmers write code that humans can understand"***
> Martin Fowler

Use design patterns.

- think about how you will document your code before you begin writing it
- think about code reliability and portability (re-use) and not about 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 read_temperature()
  uint8_t greenhouse_temperature = 0;
  #define TEMPERATURE_COEFFICICENT 324
- Use nouns for variable names and verbs for function names
- Make all names descriptive
- No magic numbers
- Think about the correct type of variable to use:
  - the temperature outside **uint8_t temp** (is this ok?)
  - the temperature in a cool store
  - the relative humidity in a greenhouse
  - the floor a lift is on
  - whether the car door is locked or unlocked

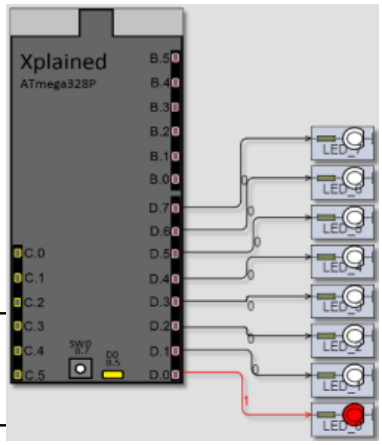
- Put only one task into a function
- Keep functions modular and small to 1 viewable page of code
- Always use {} with if statements- even though not entirely necessary
- Indent your code – 4 spaces
- Comment blocks and lines of code with their purpose or function from a system perspective, avoid stating the obvious about syntax unless it is complex or abstract
- Let the compiler do the optimization; most of the time our efforts at optimization fail us.
- Use local variables ~~whenever possible~~. A change to a global variable breaks everyplace it is used. Use the narrowest scope possible for all variables. (http://www.koopman.us/bess/chap19_globals.pdf)

© Dr. Bill Collis 2021

2.7.The interrelatedness of hardware and software in ES's

A core learning objective is to work fluidly between hardware and software. This involves understanding registers. Registers are used to control internal peripheral devices such as IO ports, and they are used to transfer data between all the peripheral devices and the CPU. Each IO Port peripheral has ____ registers

Describe how program commands work that link hardware and software

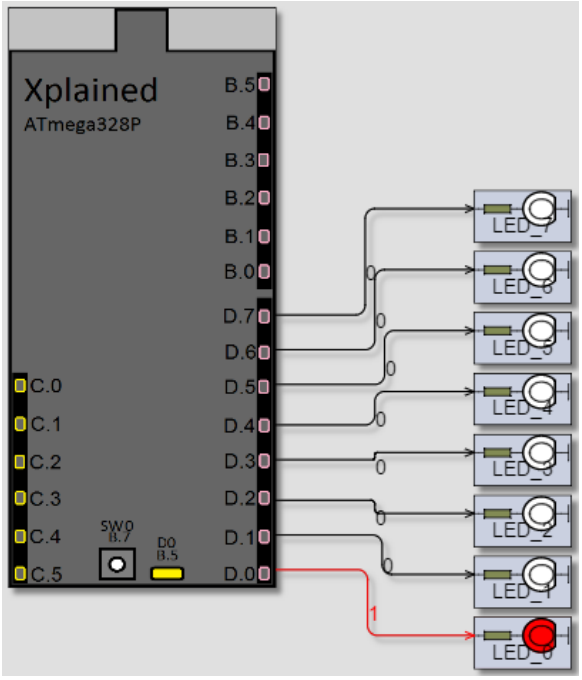



```
/***** Project Header *****/
// Project Name: Cylon 8
/***** Hardware defines *****/
//make sure this matches your oscillator
#define F_CPU 16000000//crystal
/***** Includes *****/
#include <avr/io.h> //_____
#include <util/delay.h>
/***** User macros *****/
#define TIME_DELAY 100 //_____
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD = 0xff; //shortcut to setup all 8 pins as outputs for the 8 LEDs
    /***** 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;      //
```

www.byyourcommand.net

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

© Dr. Bill Collis 2021




```
_delay_ms(TIME_DELAY);
```

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

Software that controls IO hardware peripherals

```
DDRD = 0xff;
```

1. make **all** 8 pins of portD outputs
2. by writing to Data Direction register D
3. alternatives:
   a. DDRD = 0b_____
   b. DDRD = _____

Software that transfers data from the CPU to the IO hardware peripheral

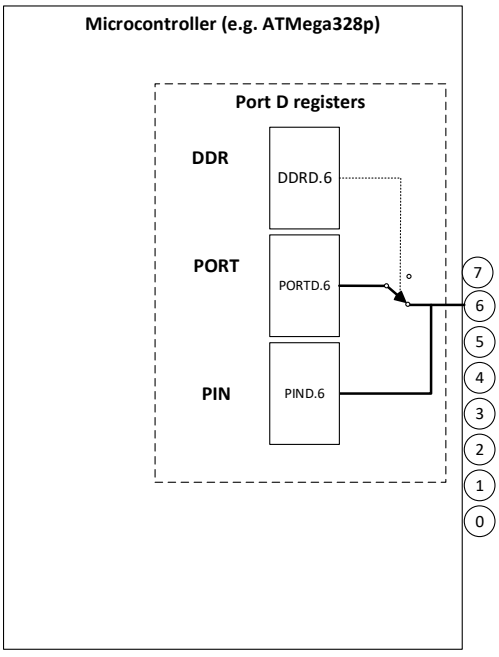
```
PORTD = 0b00000001;
```

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.

2.7.1.IO port overview

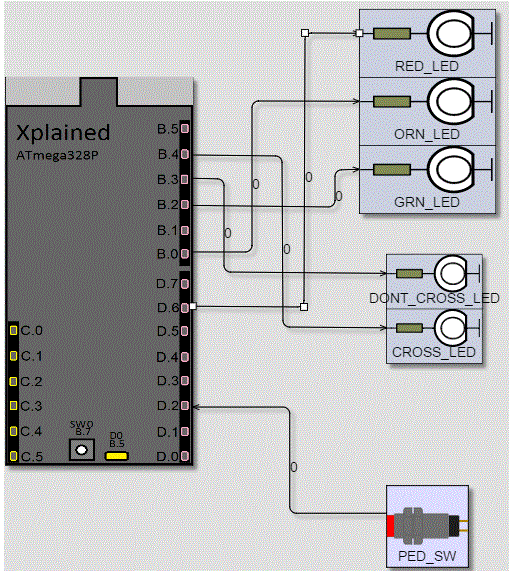


What is a register?

A register is a binary storage device (made of flip-flops). Registers can be single or multibit, and simple, shift, or multifunction. Here we mean a multifunction 8-bit register. It will have a reset (clear) function and of course a clock input. When the uC powers up, the reset lines on all the registers in the uC are set briefly, so the stored values in all registers are 0. When you want a new value in a register, e.g. PORTD= 0b10011000; the CPU will put the value 10011000 on the 8 bit data bus, then pulse the clock line for that specific register, and the new value will be stored in the register.

© Dr. Bill Collis 2021

2.7.2. IO peripherals, macros & bit masking



It is not usual to control all pins of an IO port peripheral at one time; we actually set and clear individual pins to control individual IO devices. We do this using an understanding of **Boolean identities linked to an understanding of registers.**

1. X **or** 1 = 1
2. X **or** 0 = X
3. X **and** 0 =0
4. X **and** 1 = X where 'X' means _____

For example: 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 a port register on its own. To control one output such as the GRN_LED we actually follow this process:

- read all 8 bits of register PORTB,
- change bit 2 (either high or low) while keeping all the other 7 bits unchanged *** requires boolean identities knowledge
- write the resulting value back into register PORTB.

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; // the '|' is the C command for a bitwise OR**

We are forcing only bit 2 to be high, it does not matter what value it has to start with, the effect of using boolean identity 1 will be only PORTB bit 2 will go high. The other bits are 'OR'ed with 0 (Boolean identity 2) so they remain unchanged; 0 stays as 0, 1 stays as 1, i.e. we don't care what t was to start with – it will be the same afterwards.

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; //the '&' is the C command for a bitwise AND**

Here we are forcing bit 2 low, it does not matter what value it was to start with, the effect of using Boolean identity 3 will be only PORTB bit 2 will go low. The other bits are 'AND'ed with 1 (Boolean identity 4) so they remain unchanged, 0 stays as 0, 1 stays as 1.

2.7.3. 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 almost impossible to debug. We learn early on to write macros at the top of your program that keep all the hardware control clearly transparent **and** in one place, making it easy to make changes later on.

- #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 macros 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.

© Dr. Bill Collis 2021

2.7.4.unpacking macros

The Anatomy of an output macro

- The pre-processor replaces the macro
- read the port register
- change only the required bit(s) using bitwise logic and Boolean identities
- write the new value back to the port register

Output macros

SET_RED_LED;

CLR_RED_LED;

The anatomy of an input decision?

- pre-processor replaces the macro
- read the pin register
- isolate the required bit(s) using bitwise logic and Boolean identiies
- write the new value to the port register

if (PED_SW_IS_LOW) { }

if (PED_SW_IS_HIGH) { }

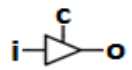
© Dr. Bill Collis 2021

2.7.5. How can a single uC hardware pin be used as both an input and an output?

Describe how program commands work that link hardware and software

Each IO pin has three bits, one from each of three 8-bit registers connected to it:

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

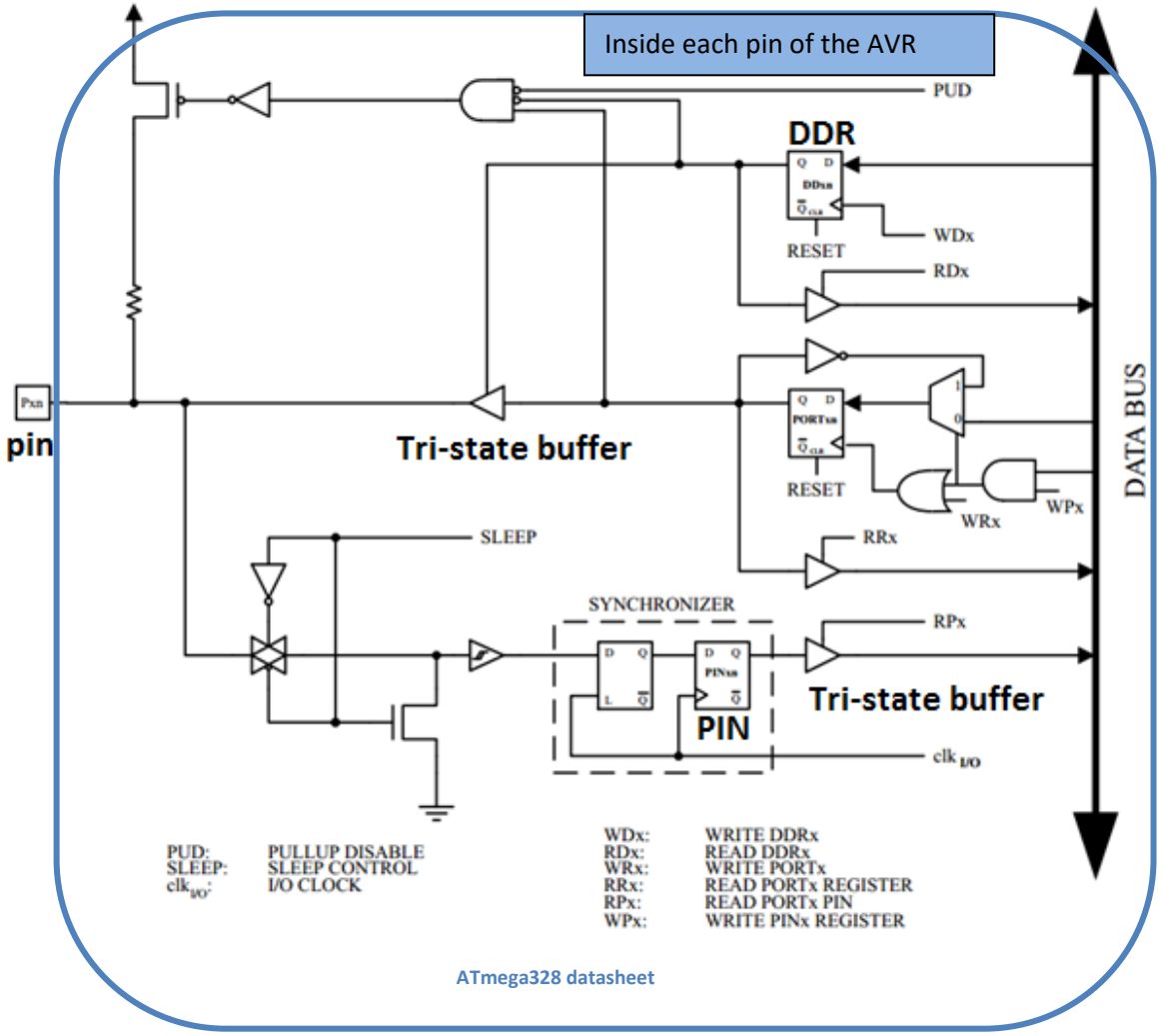
tri-state buffer 

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)



ATmega328 datasheet

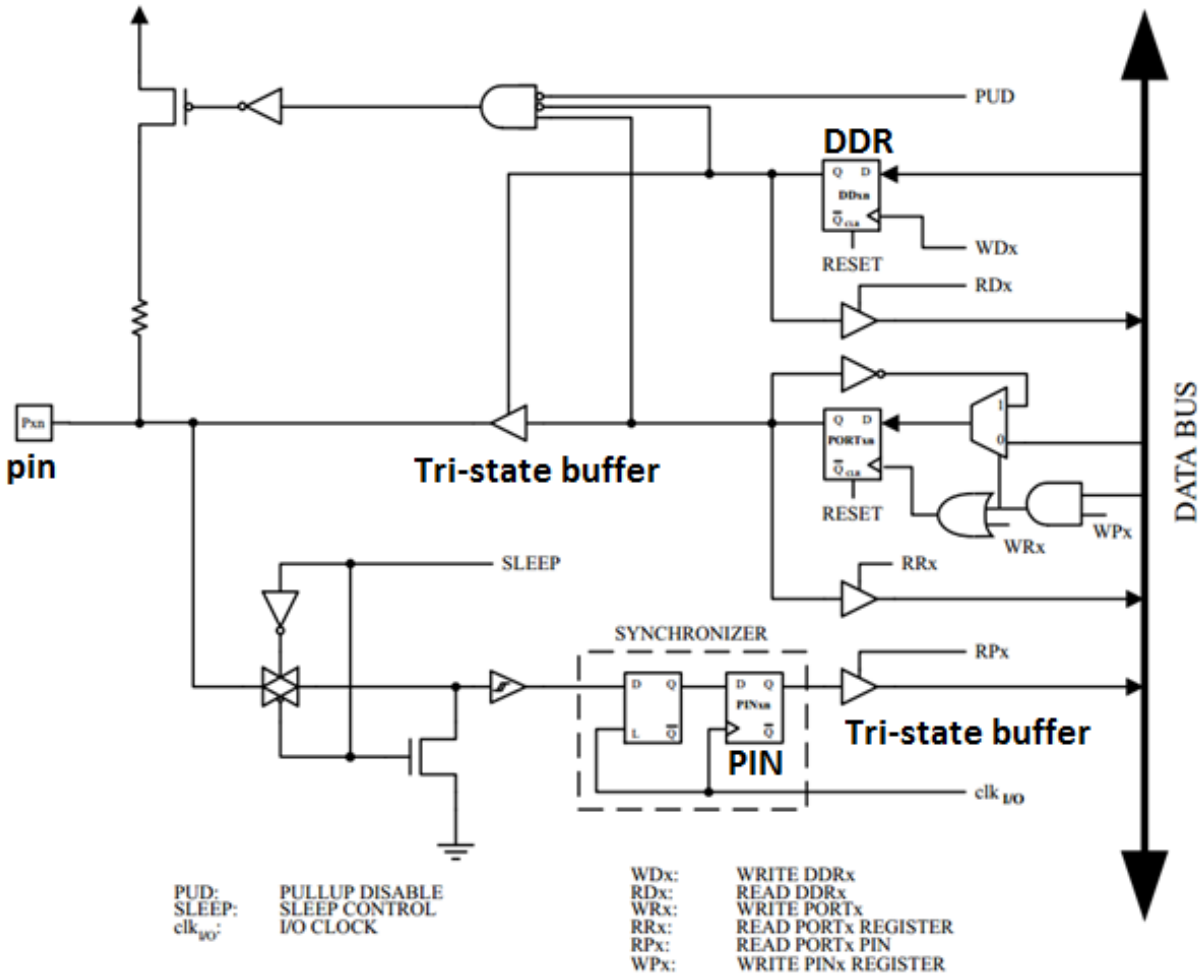
The initial power-up state of all pins is as inputs- the DDR (data direction register) output is 0. This means the **tri-state buffer** has a high Z output (it's like a tap is off) and signals from the PORT register cannot go through to the pin. The signal (1 or 0) from the physical pin comes through 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 for one pin of one port.
How many times is this block repeated in the ATmega328P?

© Dr. Bill Collis 2021

How a µC pin works as an output

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



ATmega328 datasheet

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 be used in the quiz game controller?

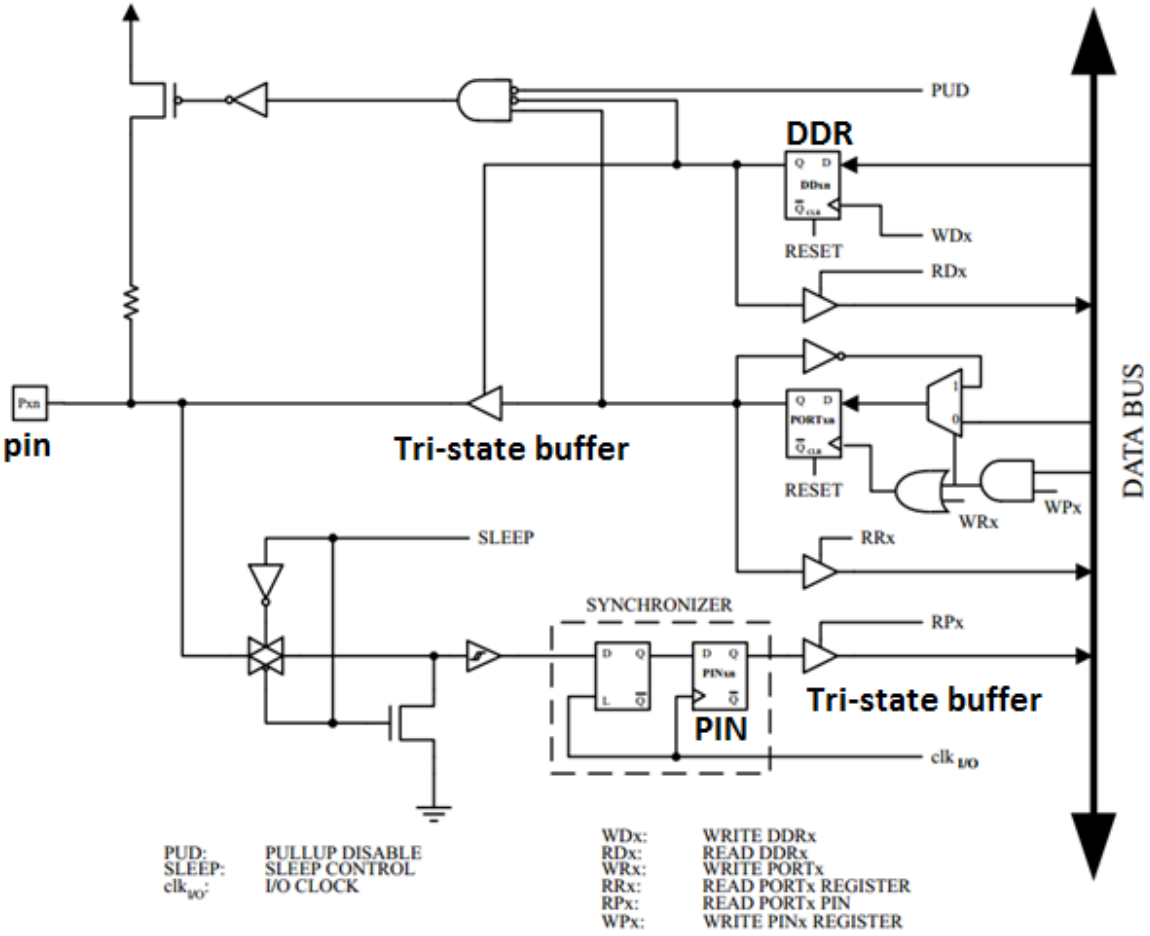
Confusing terminology alert - pin & port: When we say 'port' we are generally referring to a collection of 8 physical I/O pins not a PORT register. When we say 'D3' we are referring to a single physical pin on the uC. When we talk about reading or writing a pin, we mean a single physical pin not the PIN register which is for input only. 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.

© Dr. Bill Collis 2021

2.7.6.The internal pullup resistor

Pullup resistors are such common components in ES's that uC designers often add internal pullup resistors to every I/O pin. These can be switched in and out using software commands.

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



ATmega328 datasheet

Configuring the internal pullup resistor.

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

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

There are at least two solutions to this problem:

1. After setting up all your I/O registers set the pullup resistor for any unused pins
2. Add your own external pullup resistors - why??

© Dr. Bill Collis 2021

2.7.7.State

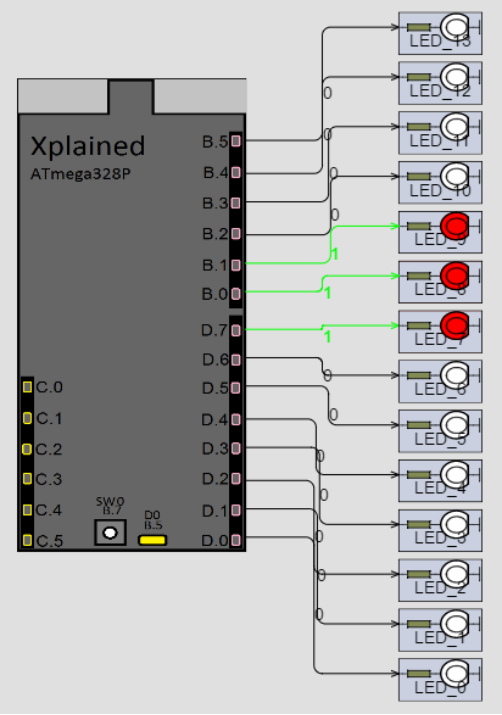
- *Explain state - the interrelatedness of the past, present and future conditions of an ES*

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 this exercise the concept of state can be seen when transitioning our Cylon program to more LEDs and having to work with two ports at once, and making the pattern run smoothly when LEDs on both ports must be controlled in one step.

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; //shift pattern by one
    _delay_ms(TIME_DELAY);
    PORTD = 0b00111000; //shift pattern by one
    _delay_ms(TIME_DELAY);
    PORTD = 0b01110000;
    _delay_ms(TIME_DELAY);
    PORTD = 0b11100000;
```



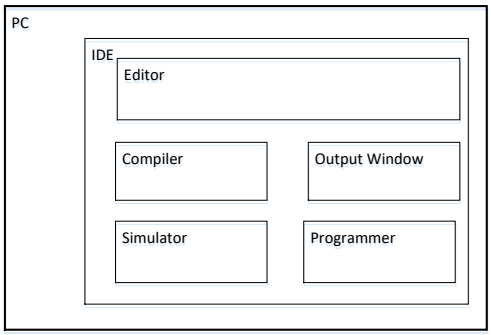
© Dr. Bill Collis 2021

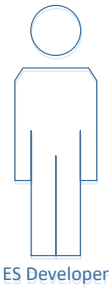
2.7.8.ES development overview

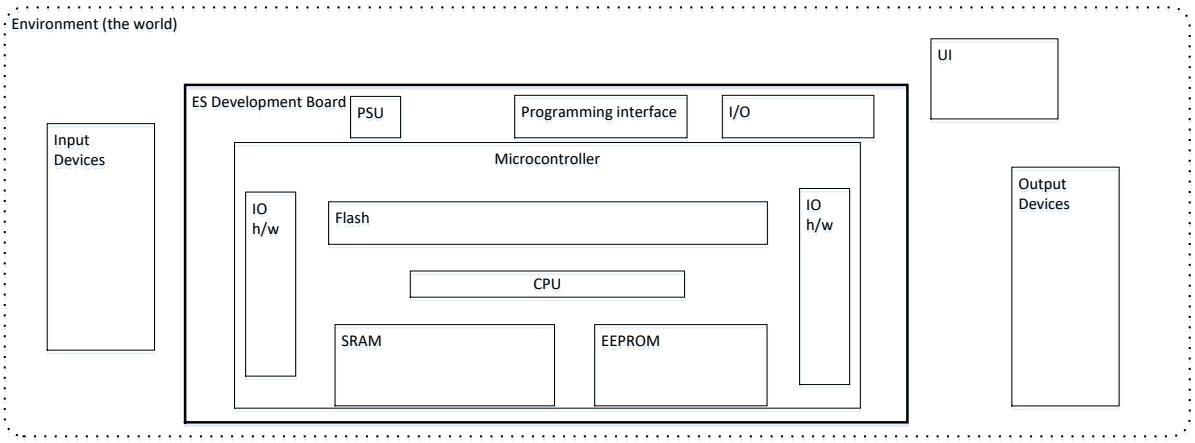
So far you have used a simulator to develop your ideas around simple AVR uC based ES's,

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





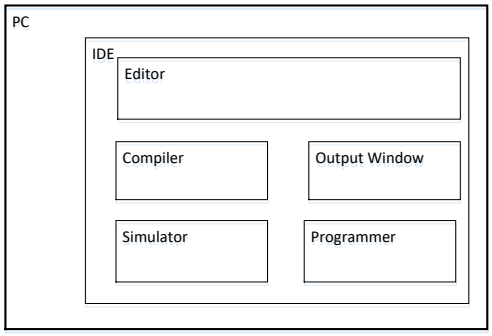


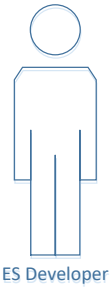
© Dr. Bill Collis 2021

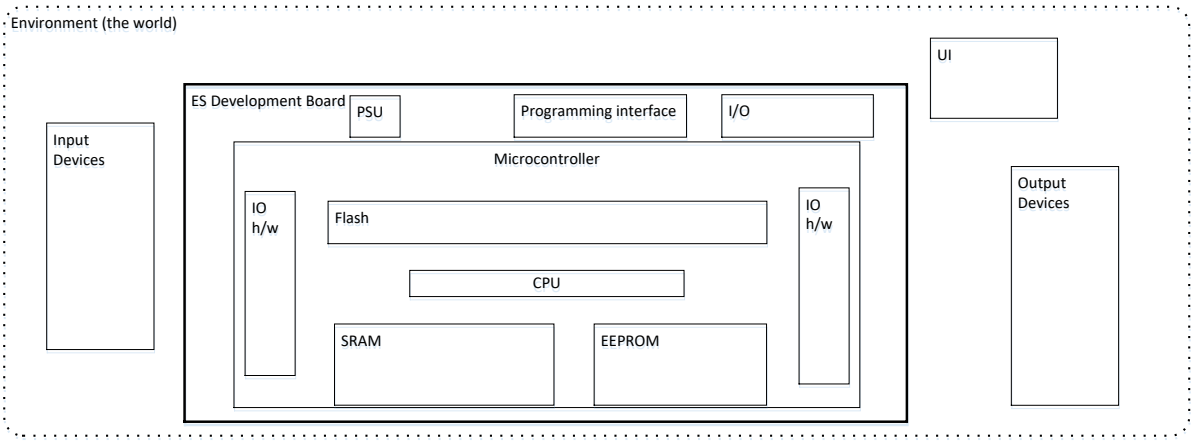
What about when the ES doesn't work quite right

These are called semantic or logic errors

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





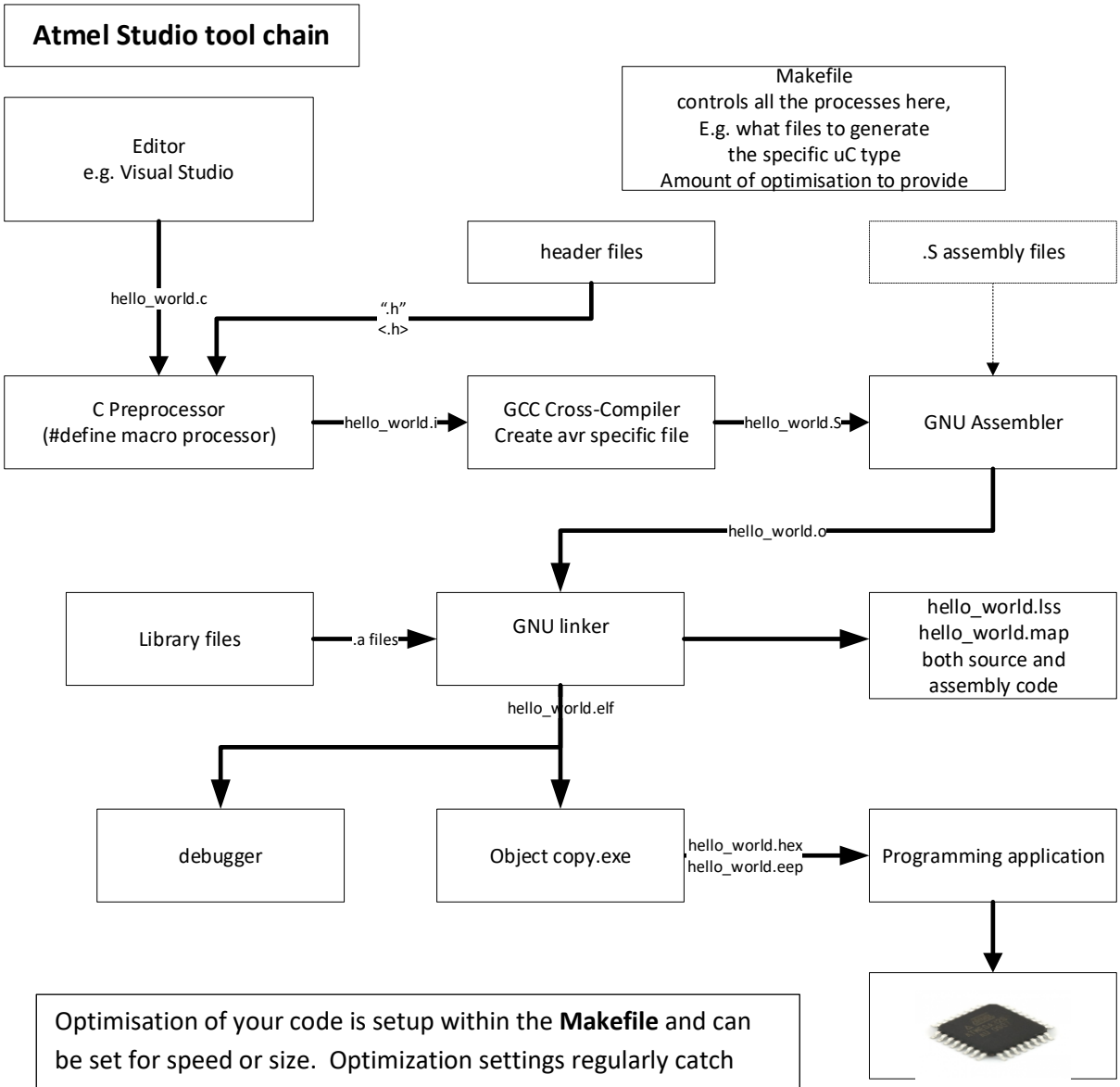


What could we add to an ES to help assist debugging?

© Dr. Bill Collis 2021

How we get from C to hex binary file in ATMEL Studio 7.

Describe the functions of an IDE - the process of compilation

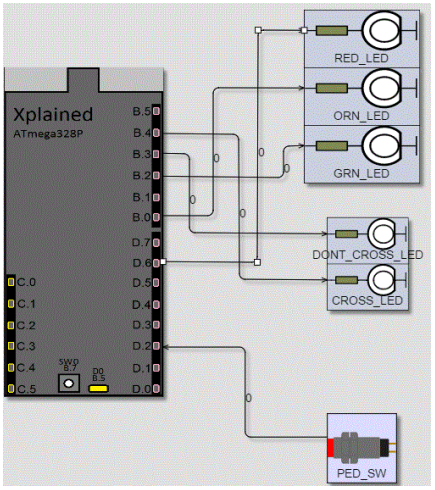


Optimisation of your code is setup within the **Makefile** and can be set for speed or size. Optimization settings regularly catch engineers out!!!

Later we will learn about the keyword _____, it can help avoid issues with optimization.

© Dr. Bill Collis 2021

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



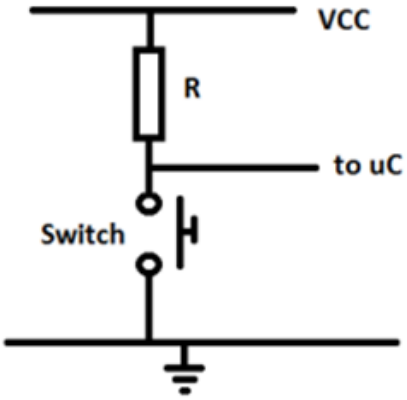

```
/***** Hardware macros *****/
//Hardware macros for outputs
1.   #define SET_RED_LED   PORTD |= (1<<PD6)
2.   #define CLR_RED_LED   PORTD &= ~(1<<PD6)
3.   #define SET_ORN_LED   PORTB |= (1<<PB0)
4.   #define CLR_ORN_LED   PORTB &= ~(1<<PB0)
5.   #define SET_GRN_LED   PORTB |= (1<<PB2)
6.   #define 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 all pins are automatically configured as inputs
20.      // make pins B4,B3,B2,B0,D6 outputs
21.      DDRB |= (1 << PB4);  //B4 set as output
22.      DDRB |= (1 << PB3);  //B3 set as output
23.      DDRB |= (1 << PB2);  //B2 set as output
24.      DDRB |= (1 << PB0);  //B0 set as output
25.      DDRD |= (1 << PD6);  //D6 set as output
26.      // Activate internal pull-up resistor
27.      PORTD |= (1 << PD2);
28.      /***** Run once code goes here *****/
29.      //setup state of all outputs
30.      //do not assume they are off
31.      SET_GRN_LED;
32.      CLR_ORN_LED;
33.      CLR_RED_LED;
34.      SET_DONT_CROSS_LED;
35.      CLR_CROSS_LED;
36.      /***** Loop code *****/
37.      while (1) {
38.          if(PB_SW_1_IS_LOW){
39.              SET_ORN_LED;
40.              CLR_GRN_LED;
41.              _delay_ms(ORANGE_DELAY);
42.              SET_RED_LED;
43.              CLR_ORN_LED;
44.              _delay_ms(RED_DELAY);
45.              SET_CROSS_LED;
46.              CLR_DONT_CROSS_LED;
47.              _delay_ms(PED_DELAY);
48.              CLR_CROSS_LED;
49.              SET_DONT_CROSS_LED;
50.              SET_GRN_LED;
51.              CLR_RED_LED;
52.          }
53.      } //end while(1)
```

Which line(s) of code is/are part of making the ES responsive to its environment?

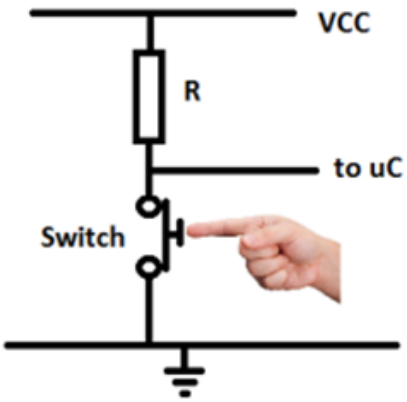
© Dr. Bill Collis 2021

The pullup resistor is a critical component in making the input responsive. When the switch is:

- **not pressed**,



pressed,



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

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

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<<PD2)

if ( SWITCH_IS_HIGH ){

}
```

Draw a pulldown circuit

© Dr. Bill Collis 2021

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

- *Apply a formal approach to writing program code*

Making code transparent with functions

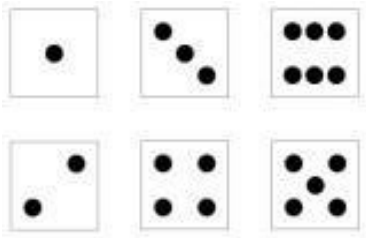
Functions have a single purpose

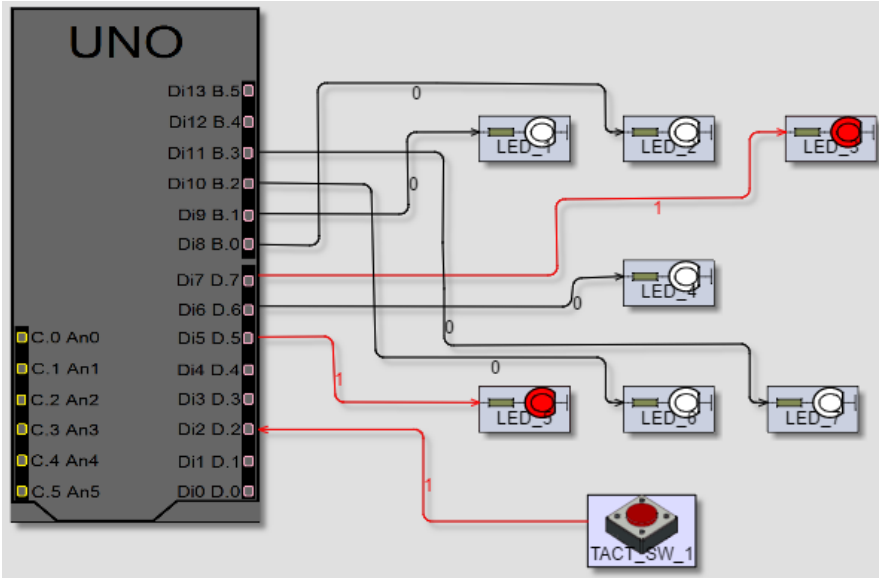
Functions have descriptive names

Functions fit in 1 page

Use function prototypes

```
/***** 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(); //function call
                    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
}
```





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?

© Dr. Bill Collis 2021

2.8.2. LO: Understand the importance of transparent software practices

- **Apply a ~~spaghetti~~ formal approach to writing program code**

www.freefoodphotos.com

When a colleague opens up your code, what does it look like inside?

```
/***** 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 *****/
    DDRB = 0xff; //setup all pins as outputs
    DDRD = 0xff; //setup all pins as outputs
    /***** Loop code *****/
    while (1) {
        PORTD = 0b00000001; //turn on firs LED
        _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 documents.

© Dr. Bill Collis 2021

2.8.3. Apply a formal approach to writing program code

At the core of professionalism in programming is **transparency**. Code must be *readable, logical, understandable and maintainable*. In programs, indentation and macros are used to structure code and make it transparent!! 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. }
```

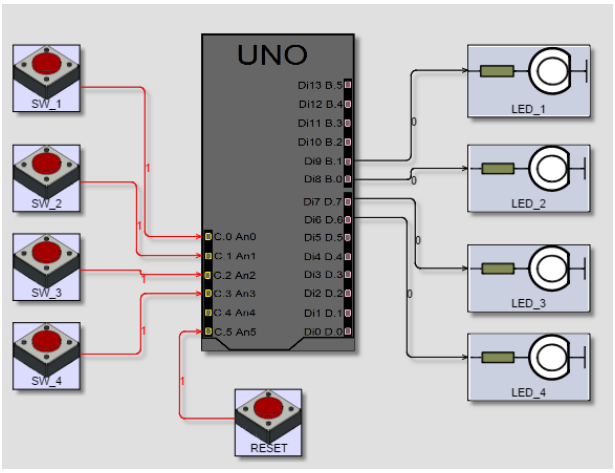
© Dr. Bill Collis 2021

2.8.4.**Polling**

Describe how an ES operates with the illusion of concurrency

Polling refers to the concept of regularly checking something (e.g. voters before an election, SET student evaluations) to see what is happening. When a uC has to check multiple inputs or variables to see if they have changed, this means that the program has to **poll** or regularly check each pin. Critically polling must be at a rate that is fast enough not to miss any event. When polling input pins, the rate of polling relates to how fast the environment might change state. 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. If we do not poll fast enough then the state of an input might change and then change back before we have polled it, that's bad as we might miss an importanct change in the environment. 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 switch press is ever missed, which might give an unfair advantage to another contestant .

1www.instructables.com



How does a quiz game controller function? Describe its function here. This is its algorithm, it can be drawn in diagram form or written as plain language or written in pseudo-code.

```
while (1) {
```

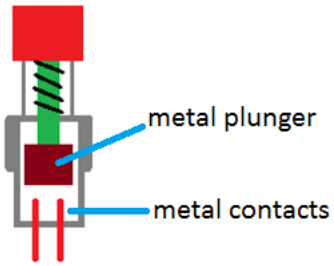
A 'cheat' for this game might be to favour a particular switch by polling it more often!!!

© Dr. Bill Collis 2021

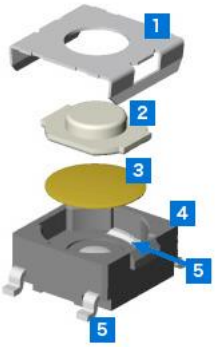
2.8.5. Hardware topic: Switch issues

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

- Anatomy of a cheap push-button switch: When the button is pressed 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 _____

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?

© Dr. Bill Collis 2021

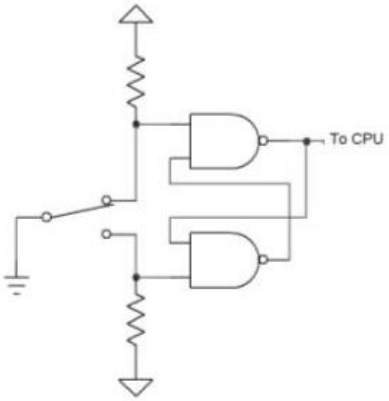
De-bouncing switches

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

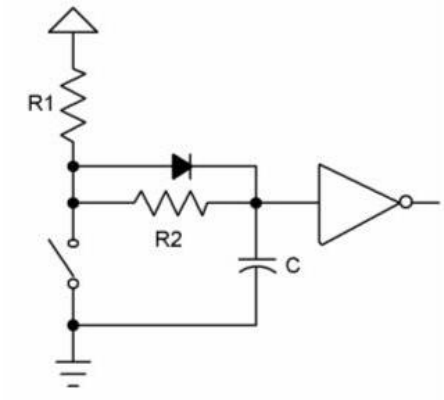
Hardware solutions

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.

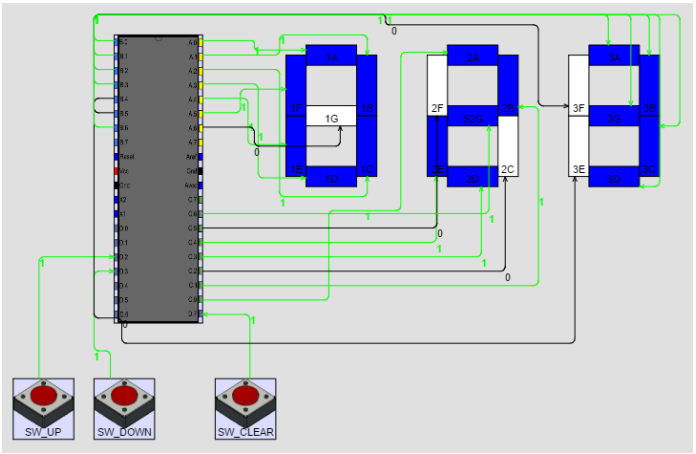
© Dr. Bill Collis 2021

2.9. Memory mapped IO and Pointers

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

In ENGGEN131 pointers were introduced. Think about what pointers are for, to store the address of a variable. In an ES, one powerful use for pointers is to refer to the address of hardware registers. We can do this because IO in a uC is memory-mapped – appears to be just like SRAM or FLASH memory. When you think about this, it makes sense because IO and memory are just locations with addresses .

In this 3 digit counter the 3 displays are connected to 3 different ports, but wired alike.



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 = …

We **could** write three different display functions

```
void disp(int16_t number){
    //split the number into 3 digits
    disp_hundreds (hundreds);
    disp_tens (tens);
    disp_units (units);
}
void disp_hundreds(uint8_t digit){
    if(digit == 0){
        PORTA = 0b11000000; //PORT A
    }
    if(digit == 1...
    if(digit == 2...
}
void disp_tens(uint8_t digit){
    if(digit == 0){
        PORTC = 0b11000000; //PORT C
    }
    if(digit == 1...
    if(digit == 2...
}
void disp_units(uint8_t digit){
    if(digit == 0){
        PORTB = 0b11000000; //PORT B
    }
    if(digit == 1...
    if(digit == 2...
} (but this is very wasteful of our FLASH memory)
```

© Dr. Bill Collis 2021

An efficient use of flash memory is to write one function not three. To do this the common and unique aspects of the functions must be identified. In this case the unique aspects are the 3 addresses of the port registers (A,B,C). The rest of the functions are identical. We can then write one function and pass it a port address. We can do this because the IO is memory-mapped in the AVR.

```
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
    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		

2.9.1. 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.

© Dr. Bill Collis 2021

2.10. 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. As commands are processed sequentially, and can take various paths in software, due to conditional (if) statements, it can be impossible to predict how long even a simple program loop can take; making it impossible to create accurate timing.

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

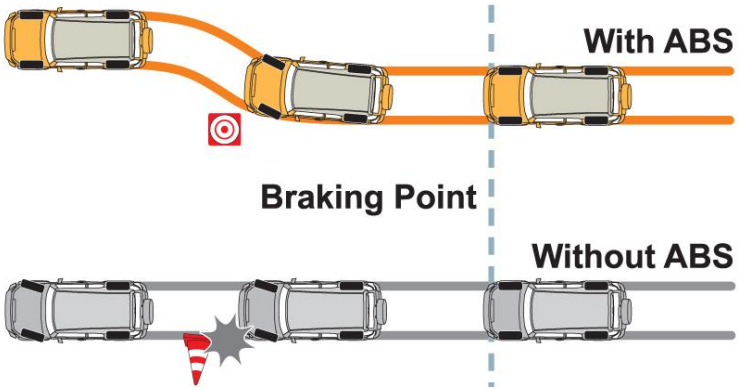
Some events in the real world need to be accurately controlled within tight timing constraints- sometimes in the order of 10^{-6} or 10^{-9} seconds. An example is a petrol engine, controlling the timing of the explosion of fuel in the cylinder is critical for power, fuel efficiency and emission reduction.

A microcontroller is required to control the timing of output devices such as this not only with great accuracy but reactively (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 the speed of each wheel of a vehicle needs to be measured accurately for the ABS (anti-lock braking system) to work. A microcontroller can measure the speed of each wheel then determine whether one has a different speed to the others to see if the car is at risk of loosing traction (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 and measure the wheel speed, meaning the uC will never miss an event.



http://simnascar.com/tag/abs/

© Dr. Bill Collis 2021

2.10.1. 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 binary counters based upon flip flops, but we use the words slightly differently to describe their different function.

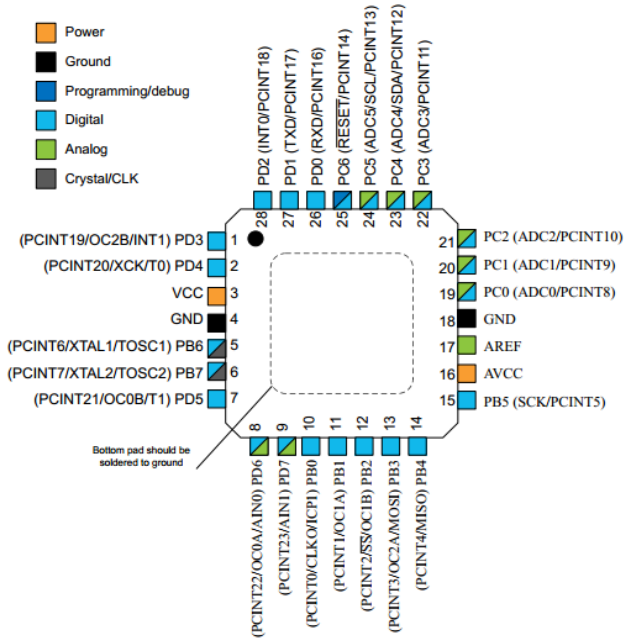
Timers are complex - and allow complex relational problems to be solved

- 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 some timers

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



© Dr. Bill Collis 2021

2.10.2. Timer example – 'heart beat' led flash

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

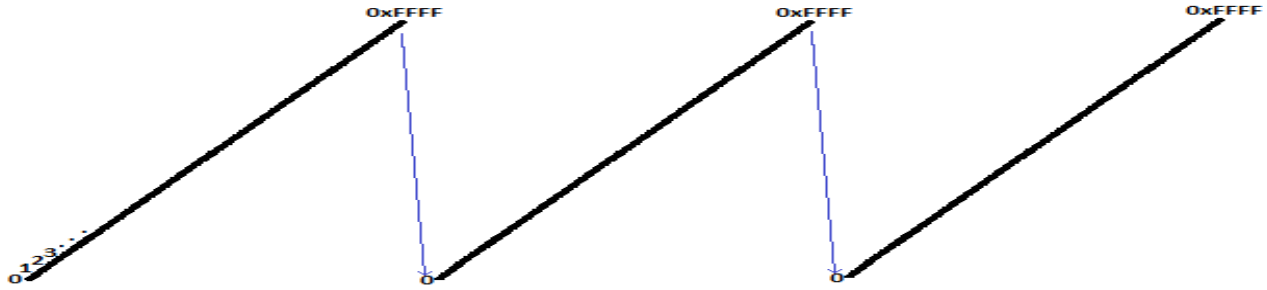
How to accurately control the timing of an external device without involving counting in the programs main loop.

One example of this 'heart beat' LED on the front of a piece of equipment. Users need to feel confidence that a piece of equipment is functioning, a steady lit LED is often used to indicate that power is on; a flashing LED makes users think that the equipment is actually working (like its alive, so we use the term heat-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. This diagram is used to show the counting process from 0 to 65536 and then overflow back to 0 in a 16 bit timer.

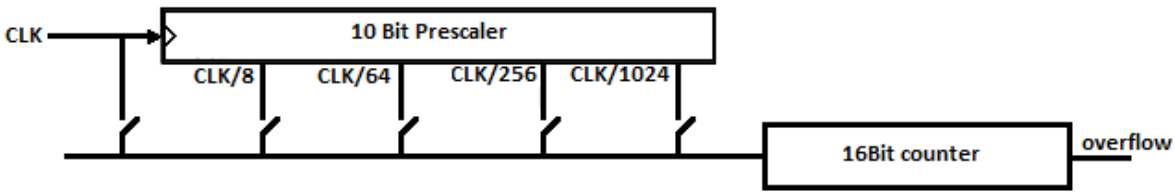


The rate of the counting is dependent upon F_CPU, the clock frequency of an Xplained mini, Arduino Nano, or Arduino Uno is 16MHz (if you are using another development board it may be different – so watch out)

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 (another counter/divider) for dividing the crystal frequency down before it goes to the main timer counter.



16Bit timer counter

Clock	No prescale	/8	/64	/256	/1024
TCNT1 input	16Mhz	2MHz			
Time period of 1 count	0.0625 uS 62.5 nS	0.5uS 500nS			
Overflow (**65536** counts)	4,096 mS	32.768mS			

© Dr. Bill Collis 2021

2.10.3. 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)					

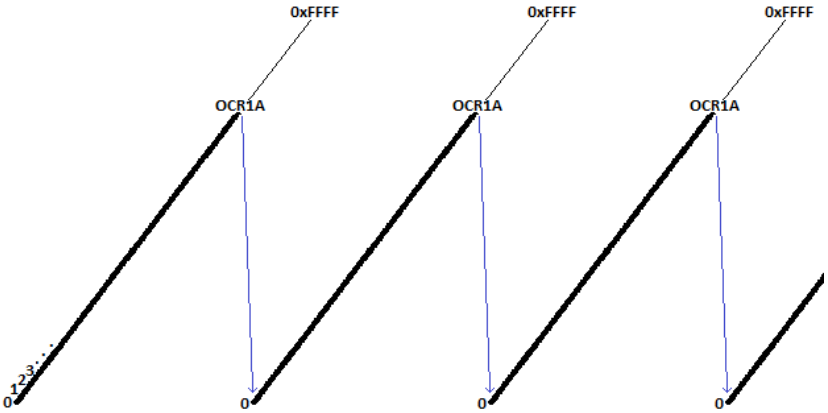
Can you think of a way to use an 8 bit timer to get a delay of about 1 second?

© Dr. Bill Collis 2021

2.10.4. 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 to 0, but to some lesser value _____and then overflows to 0

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
```

© Dr. Bill Collis 2021

2.10.5. 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 hardware 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…

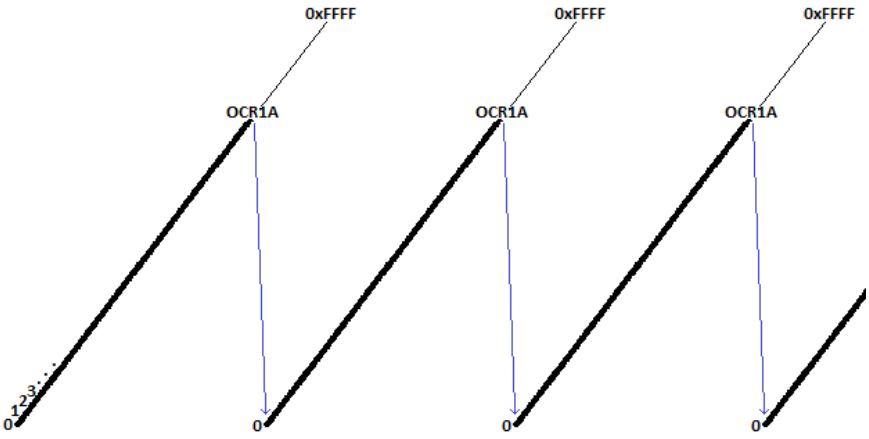
What would OCR1A need to become if we wanted 500mS on and 500mS off, and which prescale value should we use?

- 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 = _____

© Dr. Bill Collis 2021

2.10.6. 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

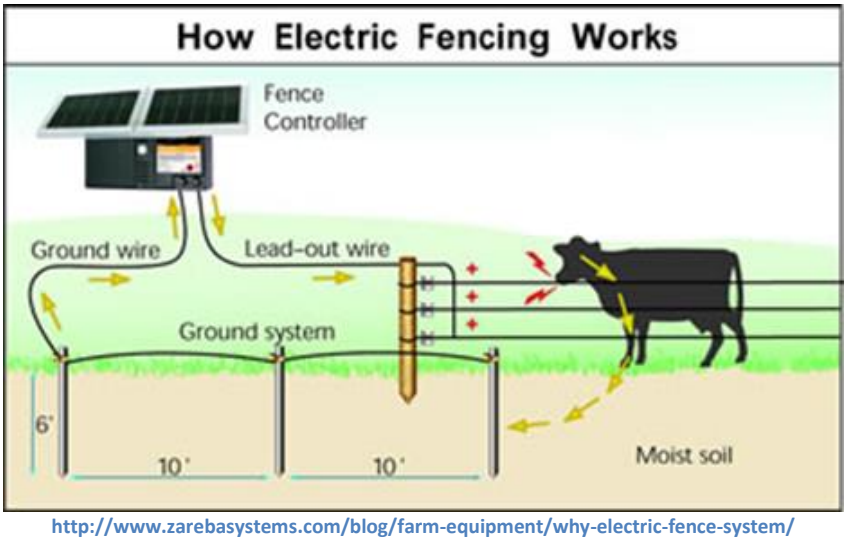



```
/***** Includes *****/
#include <avr/io.h>
#include <avr/interrupt.h>
/***** Hardware macros *****/
#define SET_LED_1   PORTD |= (1<<PD3) //could be any pin
#define CLR_LED_1   PORTD &= ~(1<<PD3) //could be any pin

/***** 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.

© Dr. Bill Collis 2021



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

Write code that uses two switches to enter hours and minute for a 24 hour timer

© Dr. Bill Collis 2021

2.10.7. Timer Registers

Each timer in the ATmega328P has a set of registers- these are described in the ATmega328 datasheet (sections 15,16,17,18)

- 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.

© Dr. Bill Collis 2021

Setting up Timer1 registers (datasheet 16.11.8)

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 ask questions on PIAZZA e.g.

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

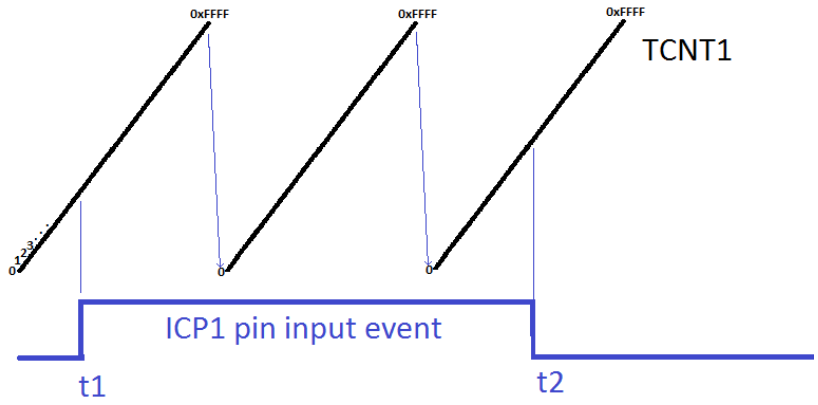
© Dr. Bill Collis 2021

2.10.8. 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 … 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.

How long is the pulse and what is the algorithm for measuring the pulse length?



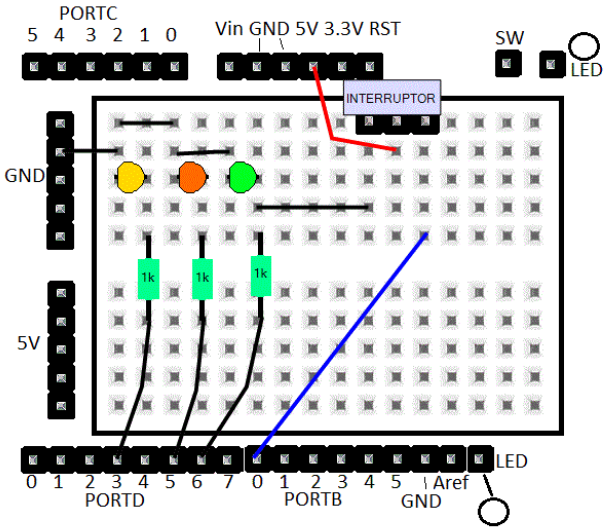
The timer is free-running (always counting) so the positive edge of ICP1 could be at any count of the timer. In the diagram above if the two counts were 15345 and 32071 in the above diagram what would the total pulse width be?

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

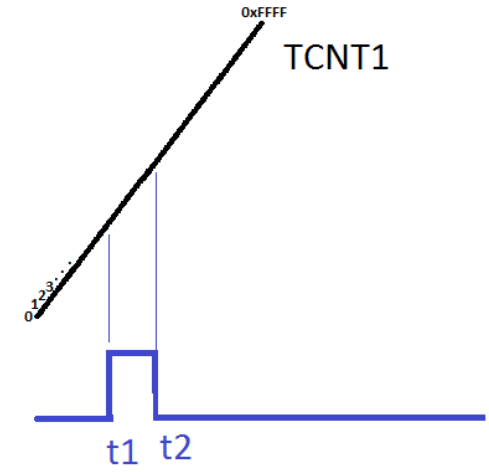
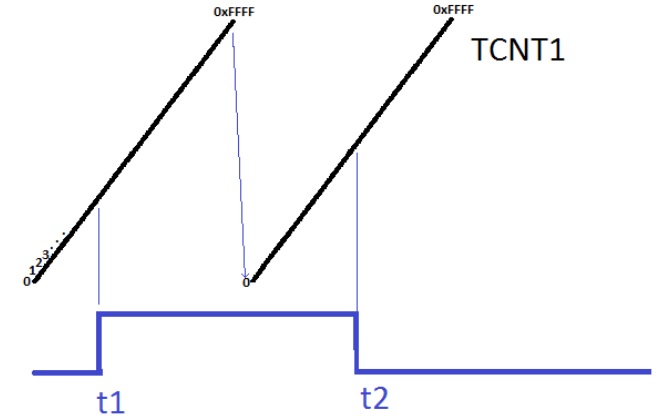
e.g. 13.7 seconds would be 1 flash of the first LED, 3 flashes of the second LED, and 7 flashes of the third LED.

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



© Dr. Bill Collis 2021

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 algorithm that works for both situations.

© Dr. Bill Collis 2021

Lab setup - Timer1 prescale = 1024

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

Positive edge detect with 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

© Dr. Bill Collis 2021

```
#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
}
```

© Dr. Bill Collis 2021

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 resolution we can get using that prescaler with the 16MHz clock

How much resolution is enough resolution?

This depends entirely upon the situation.

What determines the accuracy of uC timers?

© Dr. Bill Collis 2021

External peripherals	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.

© Dr. Bill Collis 2021

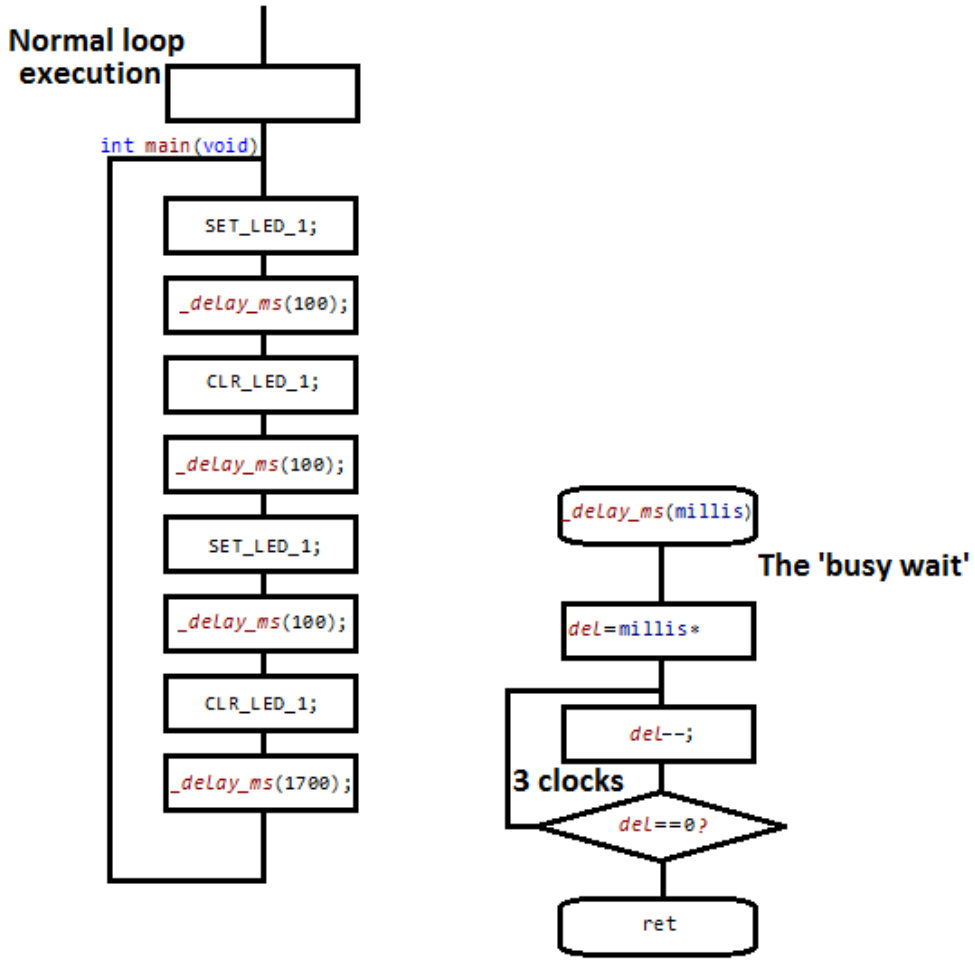
2.11. External Interrupts

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

Microcontrollers have another peripheral called an external interrupt controller, this allows the ES to react immediately to untimed but urgent things that occur around in its environment, without having to poll for them or set up a timer to check them regularly.

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.

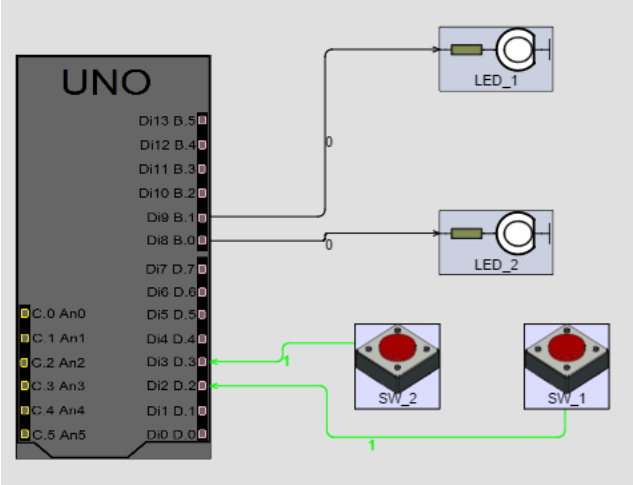
Also delays like _delay_ms() are sometimes essential in ES software because we need to slow outputs down so that slow environments are given the time to react to the system, or so that a UI component can be recognised by a user (e.g. a flashing LED). While the uC is waiting however it cannot do anything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



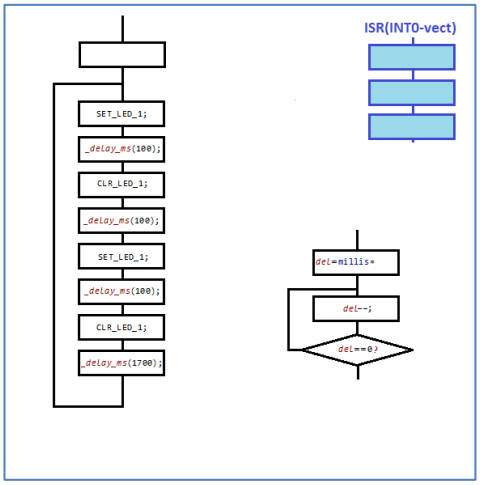
In response to this manufacturers now implement special hardware features in uC's called external interrupts. There are several different interrupt facilities in the AVR. By connecting an input to an external interrupt pin we don't have to poll the pin but let the interrupt do it for us. A hardware interrupt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

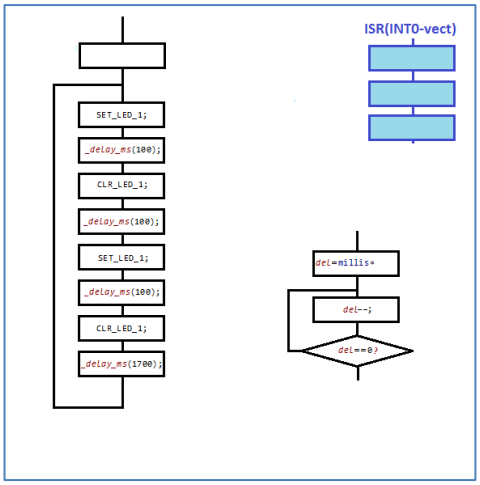
© Dr. Bill Collis 2021

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

© Dr. Bill Collis 2021

External interrupts setup

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

- 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?

© Dr. Bill Collis 2021

Setting up External Interrupts INT0 & INT1

```
#define F_CPU 16000000//crystal
/***** 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<<PB1)
#define CLR_LED_2   PORTB &= ~(1<<PB1)
#define TOGGLE_LED_1   PORTB ^= (1<<PB3)
//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) {
    // make these pins outputs
    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) {
           SET_LED_2;
           _delay_ms(100);
           CLR_LED_2;
           _delay_ms(100);
           SET_LED_2;
           _delay_ms(100);
           CLR_LED_2;
           _delay_ms(1700);
           if (SW_1_IS_LOW){//poll the switch
                 TOGGLE_LED_1;
           }     } //end while(1)
} //end of main
```

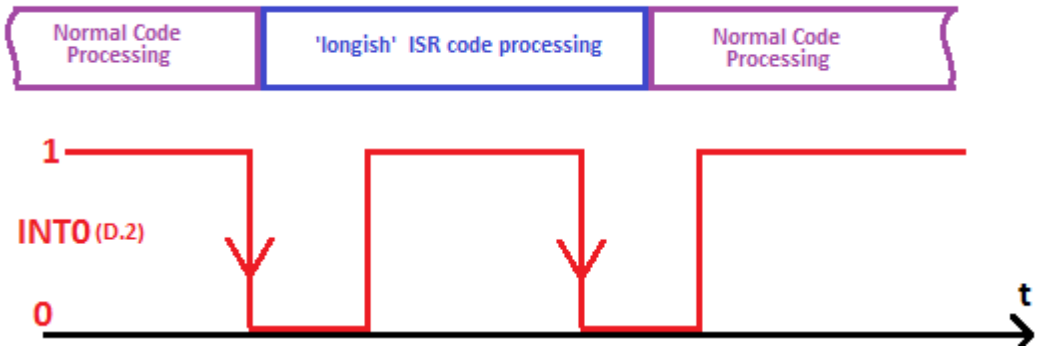
© Dr. Bill Collis 2021

2.11.1. Interrupt complications

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

Lengthy interrupts

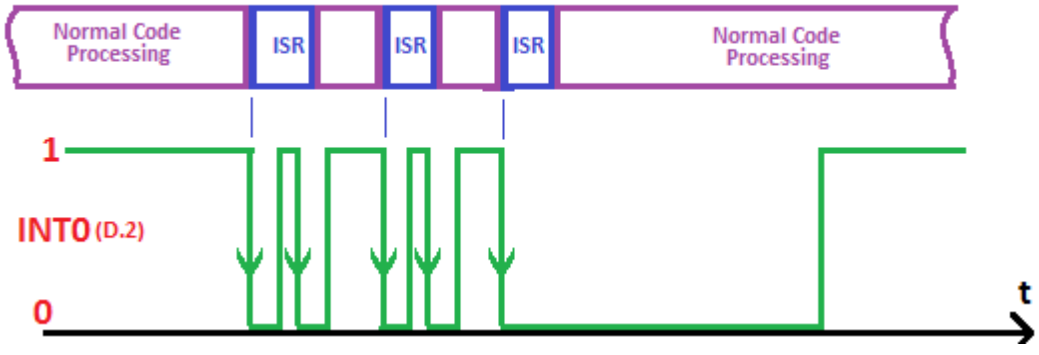
Once an interrupt is triggered and the ISR begins to execute, the global interrupt bit in the AVR's SREG is disabled, this means that if the same interrupt occurs during this period of time the AVR will not sense it and it will not be detected (other, different types of interrupts however can be queued).



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

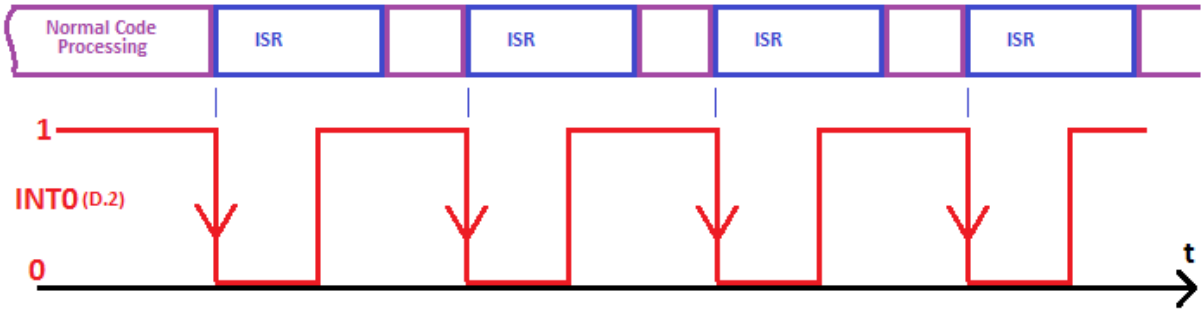


So how short should an ISR be?

© Dr. Bill Collis 2021

Too many interrupts

When your interrupt is executing your main program is not,
Excessive interrupts can…



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
}
```

Is atomicity a problem when using only 8 bit types?

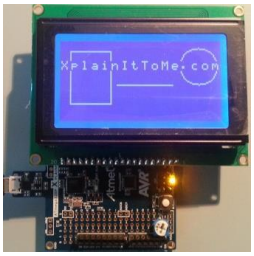
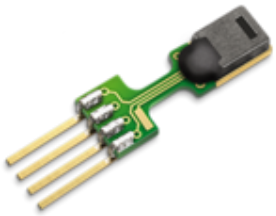
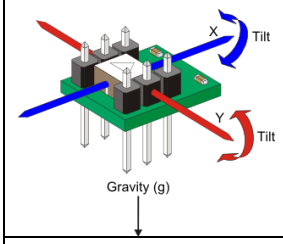
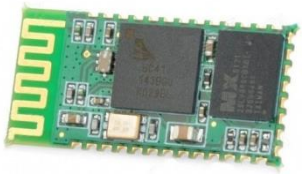
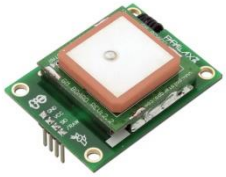
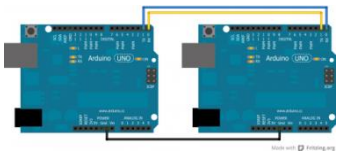
© Dr. Bill Collis 2021

2.12. Serial Data Communications

- describe the different communication protocols used in the Atmel AVR
- explain the uC USART serial data communication protocol and its parameters
- setup the registers for the uC USART peripheral

2.12.1. External serial and parallel peripherals

Some devices are more complex than a single transition on a uC's binary I/O pin or an analog input pin can deal with. In these cases data is transferred using either serial or parallel communication protocols.

		
Serial data TO a graphics LCD.	Parallel data TO / FROM a graphics LCD.	Serial data FROM a SHT11 humidity sensor
		
Serial data FROM an accelerometer	Serial data FROM a DS18B20 temperature sensor	Serial data FROM a magnetic card reader
		
Serial data TO / FROM a Bluetooth transceiver	Serial data TO / FROM a WiFi transceiver	Serial data FROM a GPS receiver
		
Serial data TO / FROM an external SRAM	Serial data FROM an RFID reader	Serial data between uC's

These peripherals require a great deal of information to be sent and received. They use either serial or parallel communications to achieve this. Most of them use serial communications, (serial is preferred because of the fewer number of connections required). The difference between serial and parallel communications can be viewed here https://www.youtube.com/watch?v=PJ_bS7meE7s

© Dr. Bill Collis 2021

Data communications needs to be completely reliable – i.e. the receiver must get the exact same information that was transmitted. To achieve the required reliability with such complex information, highly detailed protocols have been developed with very specific parameters.

- A protocol describes the rules and procedures for communication.
- A parameter is a numerical or other measureable aspect that clearly defines all rules and elements of the protocol, this includes acceptable tolerances (boundary conditions)

2.12.2. Serial communication protocols

There are several different common serial communication protocols. The AVR serial hardware circuits are the:

- USART - Universal Synchronous Asynchronous Receiver Transmitter
- TWI - two wire interface – compatible with Philips I2C or I^2C- Inter-Integrated Circuit
- SPI – serial peripheral interface

Protocol parameters

The parameters defined within a protocol might include such things as:

- Number of signal and control lines (wires)- including ground
- Signal level e.g. active high/low, 5V/3V3, -12V/+12V, and with % or values of tolerances
- Signal Timing /data rates– with tolerances around such things as shape (clock skew) and frequency
- Signal sequencing (MSB or LSB first)
- Codes that have specific meanings.

Communication channel types or modes

An important aspect of sending and receiving information reliably is that there must be an agreement between the transmitter and receiver as to which direction data is moving and when.

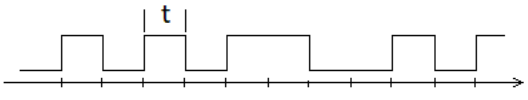
These are the different options that describe the direction in which data is transferred

- Simplex - data goes in only one direction TO or FROM the uC
- Half-duplex - data goes both ways but not simultaneously - only one way at a time
- Duplex – data goes both ways simultaneously
- This helpful animation shows the differences between them
    - http://www.iec-usa.com/Browse05/DTHFDUP.html

© Dr. Bill Collis 2021

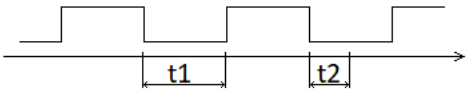
Transmission timing

The next stage in setting up a reliable transfer of information is to define the data transmission or bit rate. The bit rate is the number of bits (1's or 0's) sent per second.



Number of bits per second = 1 / t

Bit rate is very important; the receiver must align with the transmitter settings. The receiver must know the timing of a single bit so that it can determine exactly when to decode the data as 1 or 0. Here is an example of a data stream



1. If the transmitter timing of the above signal was t1 then the data would be _____
2. If the transmitter timing of the above signal was t2 then the data would be _____

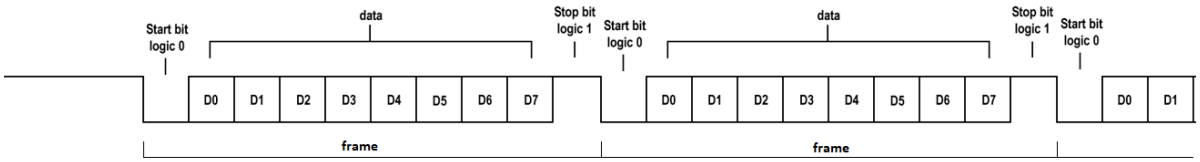
The receiver needs to be able to interpret this signal correctly; the two different ways of achieving this are synchronous and asynchronous methods:

- Synchronous:
  - A clock signal is sent along with the data on another wire
- Asynchronous
  - The receiver is setup with the timing in advance - a number of well-known bit rates exist, two are: 9600bps, 115200bps,
  - We could send an initial sequence of 10101010 so that the receiver could get a lock first

Information transfer

Information is more than single 1's or 0's. It requires groups of 1's and 0's (codes), which have meaning. If we wanted to represent numbers 0 to 9 we would need 4 bits, to send letters of the alphabet a 7 bit code has been created called ASCII (American Standard Code for Information Interchange), the letter 'A' is 0x41 = _____ This has been further extended into 8-bit and 16-bit codes called Unicode.

In asynchronous data transfer groups of bits or codes are sent with extra bits as part of a _____ to signal the beginning and end of each frame. The parameters for the frame below are 8 data bits, with 1 start bit (0) and 1 stop bit (1).

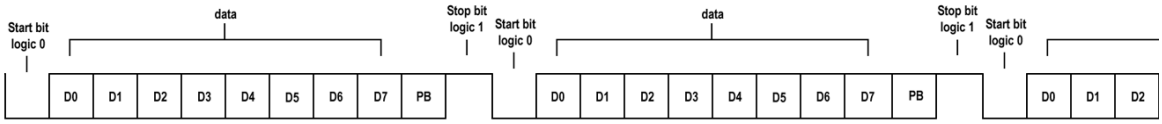


The receiver knows that the first frame has begun when the _____ is detected on the data line. The parameters for the data rate and the frame must be set up the same in the transmitter and receiver so the receiver knows how long a bit takes and can accurately tell when to measure the next bit in the sequence.

© Dr. Bill Collis 2021

Error checking using a parity bit

Sometimes electrical noise in the environment can interfere with data transmission. In this case a particular bit might get corrupted (changed). One simple method of checking for errors involves sending an extra 'parity' bit inside each frame.



In a system with a parameter setting for even parity the total number of 1's must be even, if the parameter setting is for odd parity then the total number of 1's must be odd.

	Received data frame	Parity	Is the data correct?
A	**0101010111**	even	
B	**0101111111**	odd	
C	**0111110111**	odd	
D	**0101101111**	even	
E	**0110010111**	odd	

2.12.3. **ATMega328 internal communication hardware datasheet section 20**

USART options from the datasheet

- **Full Duplex Operation (Independent Serial Receive and Transmit Registers)**
- **Asynchronous or Synchronous Operation**
- **Master or Slave Clocked Synchronous Operation**
- **High Resolution Baud Rate Generator**
- **Supports Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits**
- **Odd or Even Parity Generation and Parity Check Supported by Hardware**
- **Data OverRun Detection**
- **Framing Error Detection**
- **Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter**
- **Three Separate Interrupts on TX Complete, TX Data Register Empty and RX Complete**
- **Multi-processor Communication Mode**
- **Double Speed Asynchronous Communication Mode**

© Dr. Bill Collis 2021

2.12.4. Serial communications example code

Setup an AVR running at 8Mhz using 8 data bits, 1 start bit, 1 stop bit, 9600 baud

```
#define FOSC 8000000 // Clock Speed
#define BAUD 9600
#define UBRR_PARAM FOSC/16/BAUD-1 //Asynchronous normal mode
//*************************************************************
//usart_init: initializes the USART system
//*************************************************************
void usart_init() {
    UCSRB = 0x08;      //enable transmitter No magic numbers!!
    UCSRB = (1 << RXEN) | (1 << TXEN); // Enable Tx and Rx
    UCSRC = 0x86;      //async, no parity, 1 stop bit, 8 data bits
    UCSRC = (1 << UCSZ00) | (1 << UCSZ01)
    UBRRH = 0x00;
    UBRRL = 0x40;
    UBRR = UBRR_PARAM;
}
//*************************************************************
//usart_transmit: transmits single byte of data
//*************************************************************
void usart_transmit(unsigned char data) {
    while((UCSRA & 0x20)==0x00)  //wait for UDRE flag
    {
    ; //DO NOTHING
    ]
    UDR = data;
}
```

Receiving serial data using the USART

What might be wrong with developing this sort of code?

```
void main (void) {
    while(1){
        while((UCSRA & 0x80)==0x00)
        { ; }//wait for RXC flag
        data = UDR;
        … //the rest of your code
    }
}
```

It is better to use interrupts …

```
//*************************************************************
//usart ISR
//*************************************************************
ISR(USART_RXC_vect)
{
    rxdata = UDR;
    sei():
}
```

set up the serial receive interrupt (add to usart_init function this)

```
void usart_init(void)
{
    UCSRB |= (1 << RXCIE);
}
```

xtremely useful reference: **http://www.fourwalledcubicle.com/AVRArticles.php**

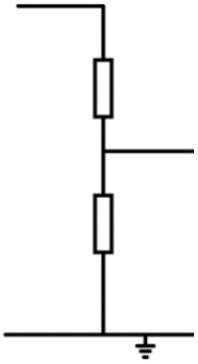
© Dr. Bill Collis 2021

2.13. The ADC

The real world is not digital, it is analog. So for an ES to be responsive and reactive to real world events it must be able to convert information from the analog world to the digital one. This is the function of the internal ADC peripheral

2.13.1. The voltage divider

At the core of converting energy in the real world into the electrical world is this fundamental electrical circuit, the voltage divider.



Working in context with voltage dividers is at the core of using ADCs and sensor subsystem design, and this involves understanding their model

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

Develop the models for these expressions
Vout =
Vin =
R1 =
R2 =

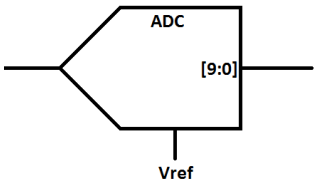
© Dr. Bill Collis 2021

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 model 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 =

© Dr. Bill Collis 2021

ADC reference select

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

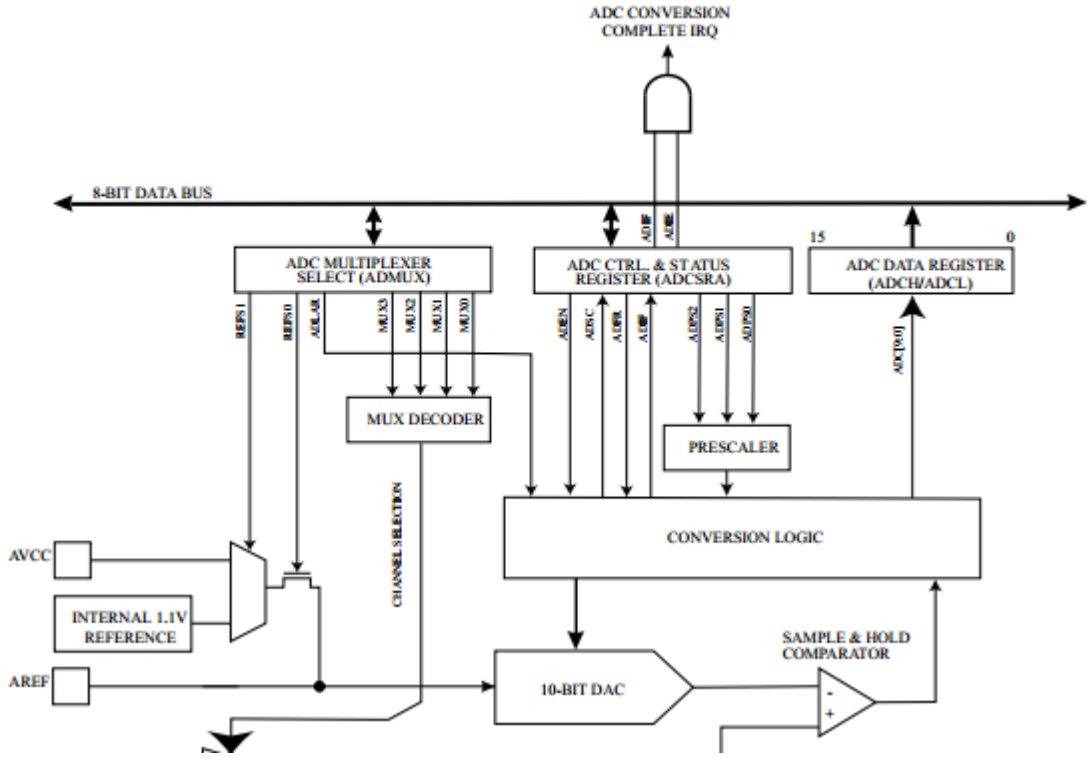


Table 28-3. ADC Voltage Reference Selection

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,

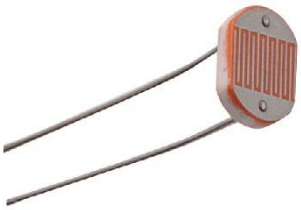
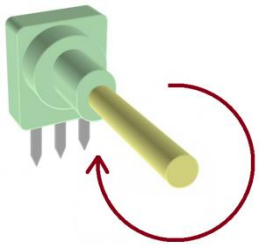
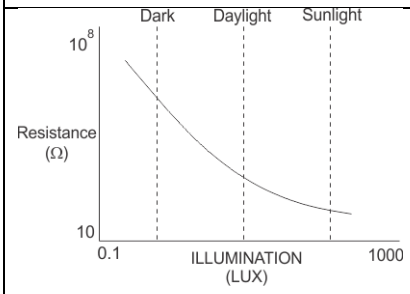
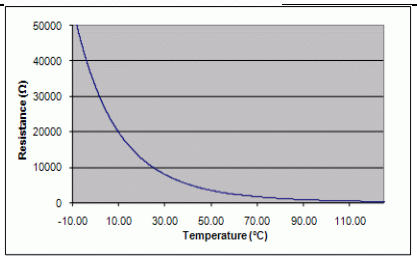
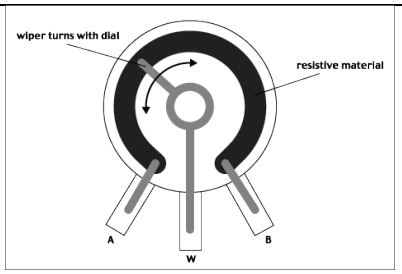
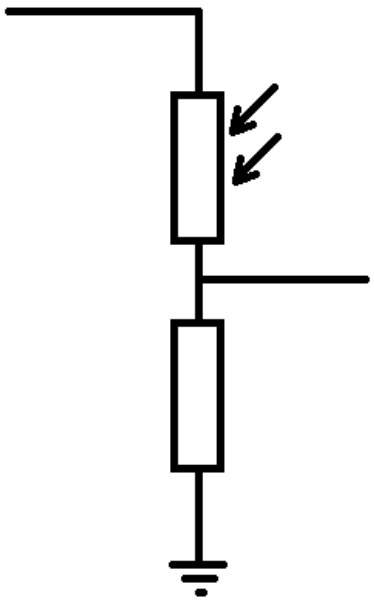
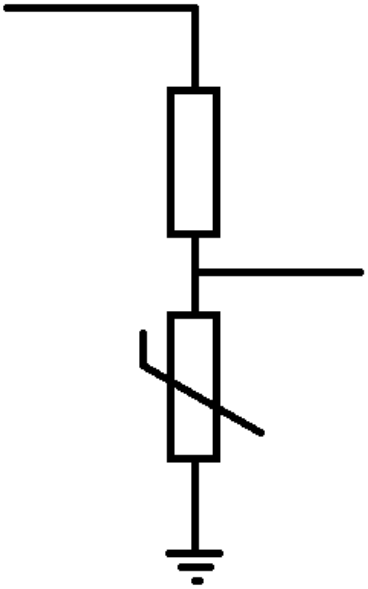
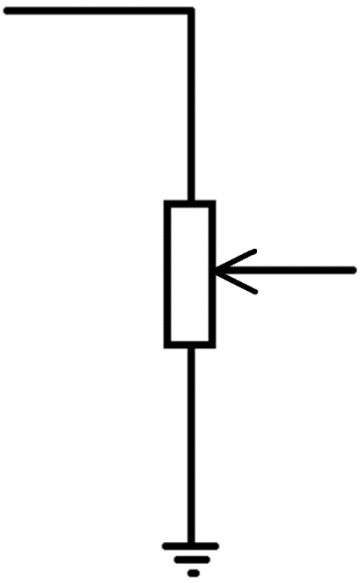
what would be the command(s) to select AVcc as the reference?

© Dr. Bill Collis 2021

2.13.2. **ES analog sensors**

Describe the use of a sensor in a voltage divider circuit

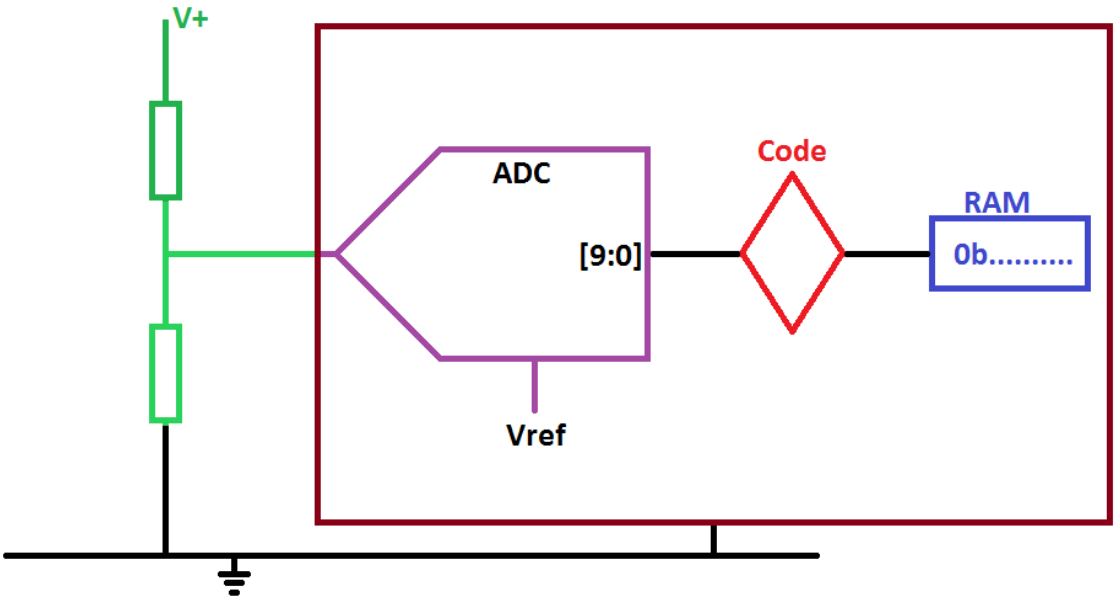
In an ES voltage divider circuit there are not two fixed value resistors but one component which has variable resistance and one which is fixed. Either can be the upper or lower resistor in the circuit.

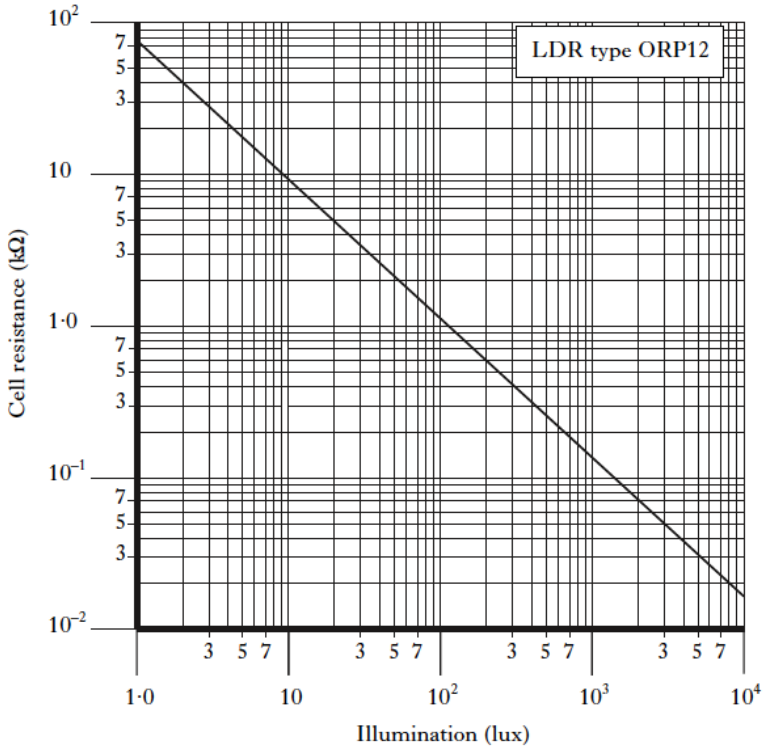
Light measurement	Temperature measurement	Displacement (potentiometer)
		
		 (linear or lagarithmioc over 270 degrees range)
		
As the light level goes up the output voltage goes _____	As the temperature goes up the output voltage goes _____	As the displacement of the object changes the output voltage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

© Dr. Bill Collis 2021

2.13.3. **The conversion process**





ORP12 Datasheet, ww.alldatasheet.com

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:

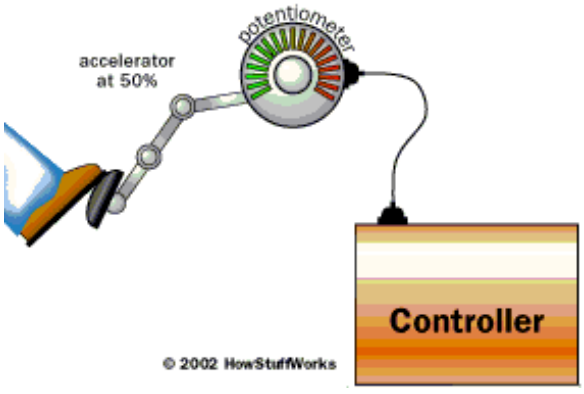
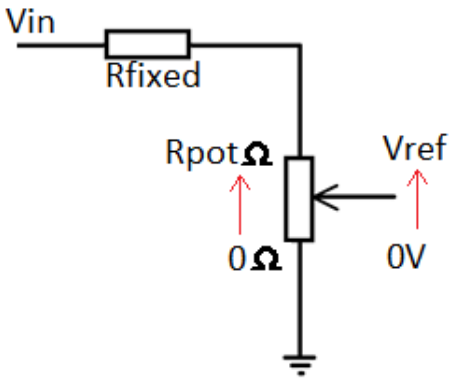
© Dr. Bill Collis 2021

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 = ??	

© Dr. Bill Collis 2021

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'?

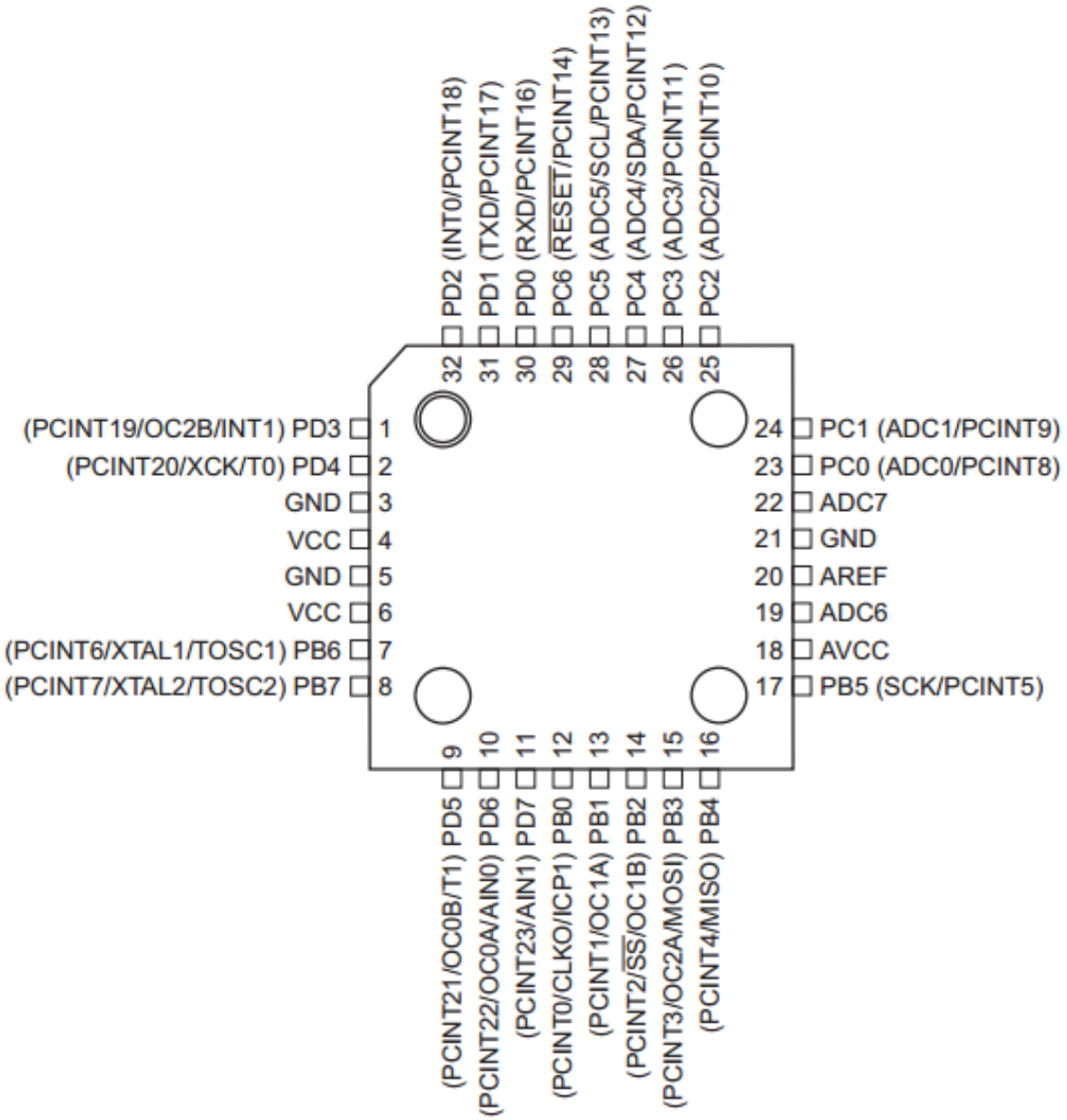
'better' relates to context and the system being fit-for-purpose. In an abstract or de-contextualised example like this there is no 'better' or 'best'.
We need to know the situation that an ADC will be used in to make a decision about what are the 'best' or minimum attributes for it.

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.

© Dr. Bill Collis 2021

2.13.4. **AVR ADC inputs**

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



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

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

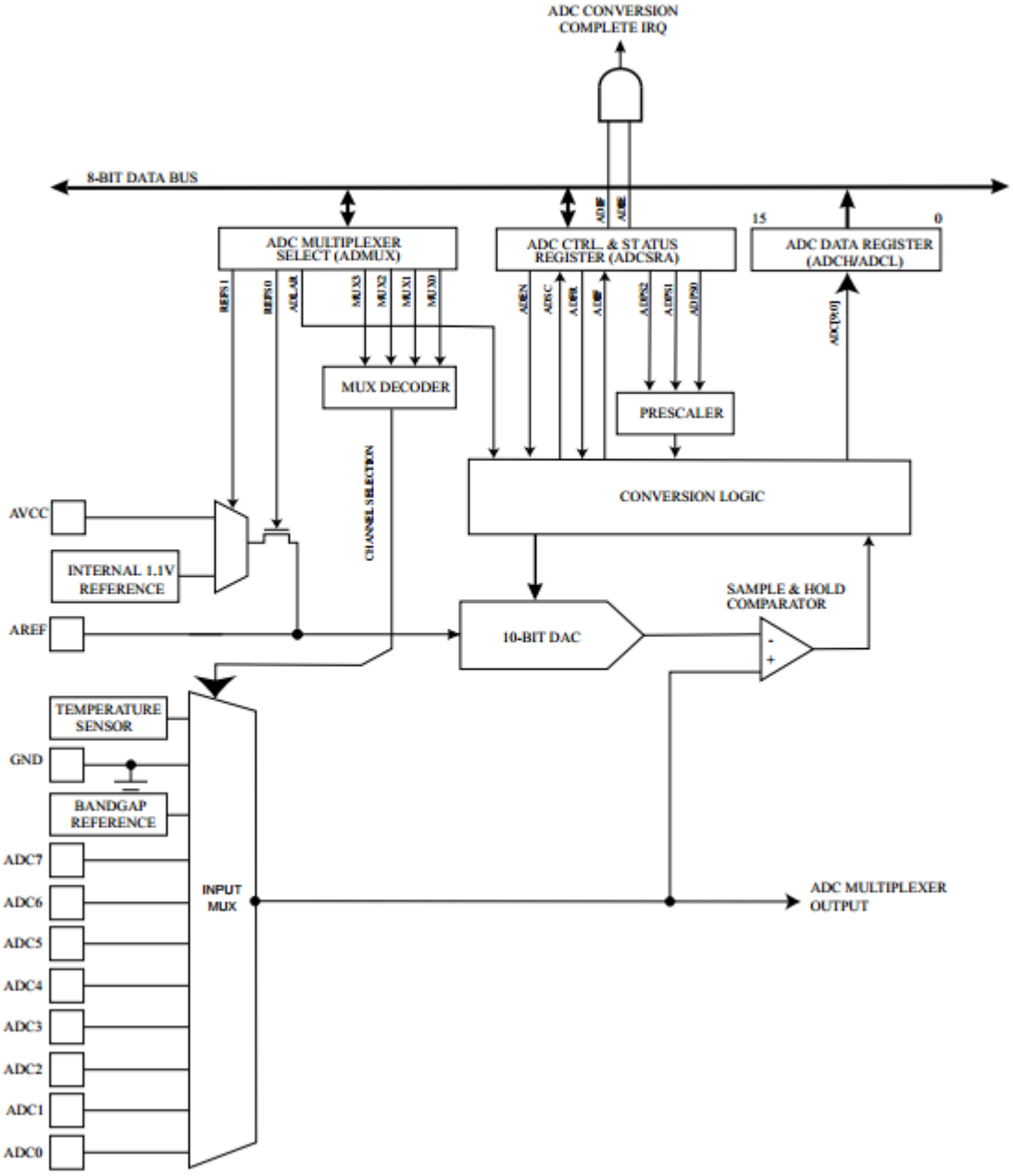
© Dr. Bill Collis 2021

2.13.5. **ADC Channel select**

The AVR's have _____ ADC pins but only ____ ADC converter. A _____ 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



© Dr. Bill Collis 2021

The selection of which channel is to be read, is made by selecting and clearing the appropriate bits in the ADMUX register, e.g if we wanted to read channel 3 (0011) then we need to set bits 1 and 0. If we wanted to read channel 8(1000) we need to set bit3. However we cannot just set the bits we need to

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 __ for an ADC conversion.

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

© Dr. Bill Collis 2021

2.13.6. **Sampling rate**

The process of making an analog to digital conversion inside an AVR is via the method known as ‘successive approximation’. This is reasonably complex, involving OpAmp comparisons and electronic switching of resistor networks, so ADC conversions are ‘slow’ compared to manipulating binary numbers. The ADC circuits take 13 clock cycles and requires a clock frequency slower than the CPU frequency e.g. 50kHz to 200kH. The clock rate is provided by the ADC prescaler which is setup using three bits in register ADCSRA.

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 division factor would meet the specification of 50kHz - 200kHz?

- 10,000,000 / 2 = 5,000,000
- 10,000,000 / 4 =
- 10,000,000
- 10,000,000
- 10,000,000
- 10,000,000
- 10,000,000

What is the command to set this division factor into the prescaler?

© Dr. Bill Collis 2021

2.13.7. 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, this makes 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

Appreciating that these different modes exist, means that you can design a conversion process for any system that will make it responsive and reactive to its environment.

Datasheet

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.

In addition to

Table 26-6. 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

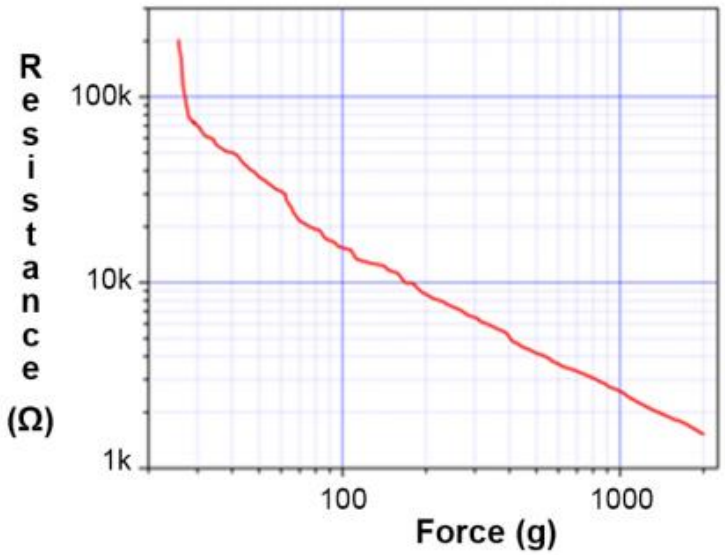
(In register ADCSRB)

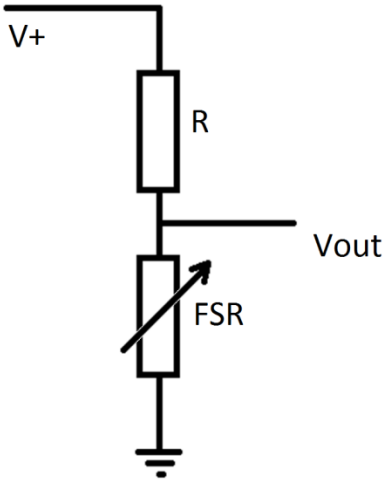
Example code to trigger ADC conversion to start on Timer/Counter0 overflow

© Dr. Bill Collis 2021

Sensor - FSR Force Sensitive resistor

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

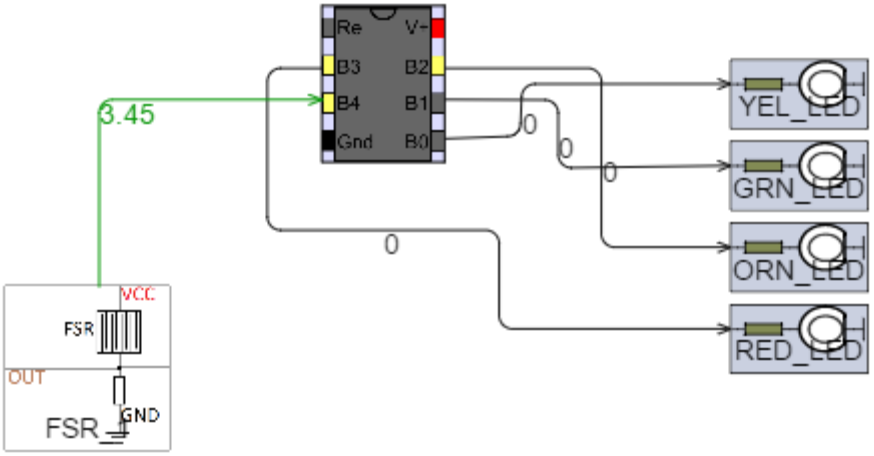




V+ = 5V, Vref = 2.56V, R= 10K

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

FSR test setup



© Dr. Bill Collis 2021

```
#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
    //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); //call the function
        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
```

© Dr. Bill Collis 2021

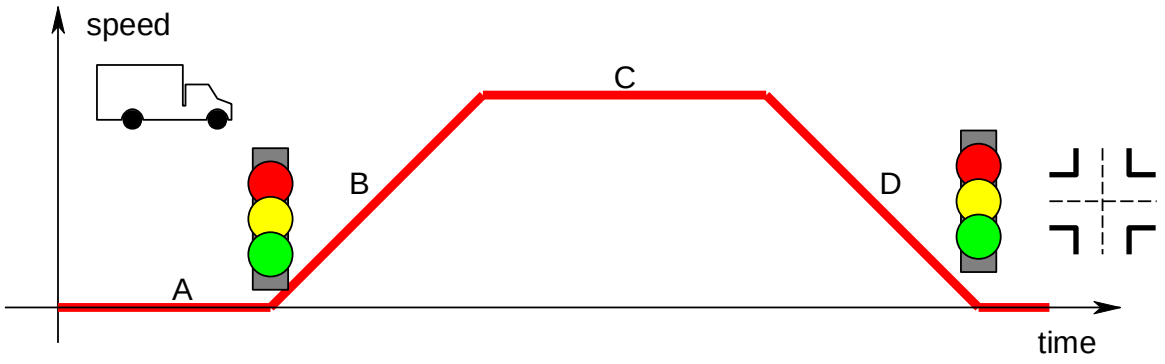
2.14. State machine programming

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. State machines are a very useful tool for non-trivial programs.

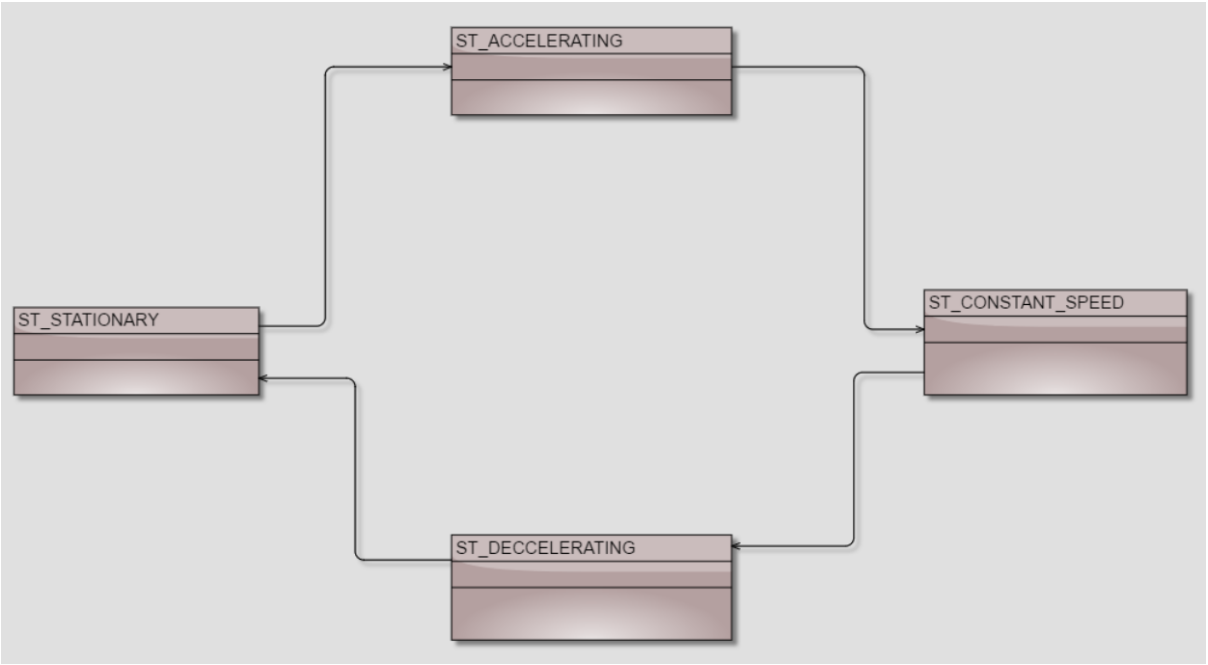
Think of the truck in the graph below as having 4 states.

- 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



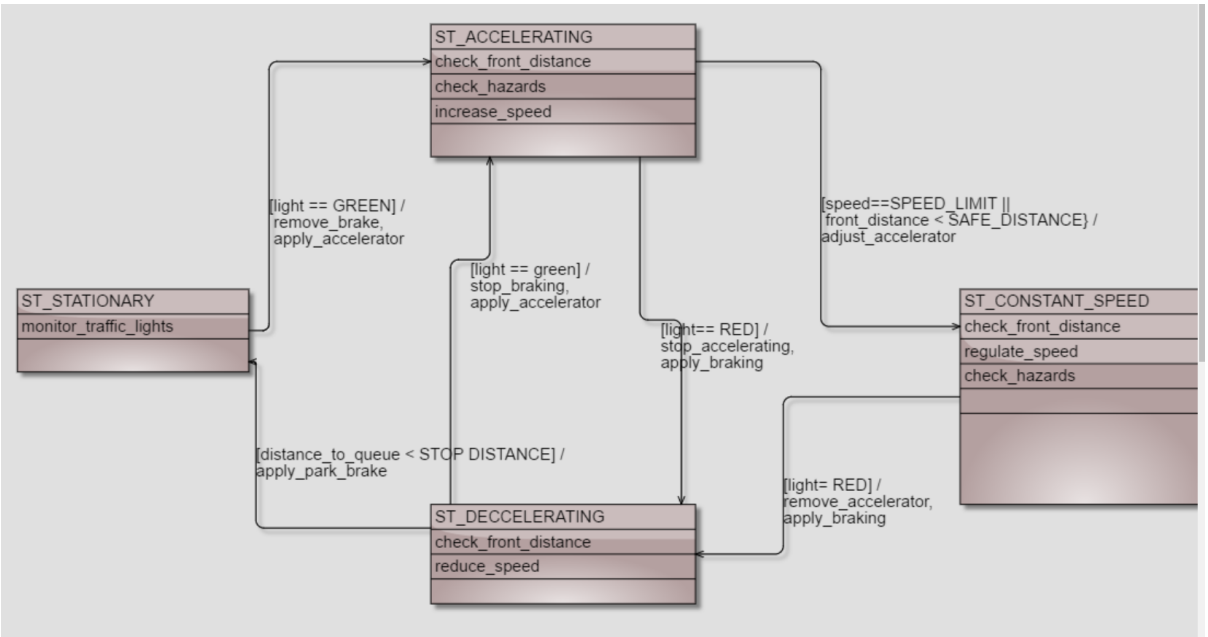
A first state diagram for the truck could look like this



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

© Dr. Bill Collis 2021

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

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

- Pacemaker
- Stopwatch
- Calculator
- Greenhouse
- Traffic lights

© Dr. Bill Collis 2021

2.14.1. 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

```
#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;
   }
}
```

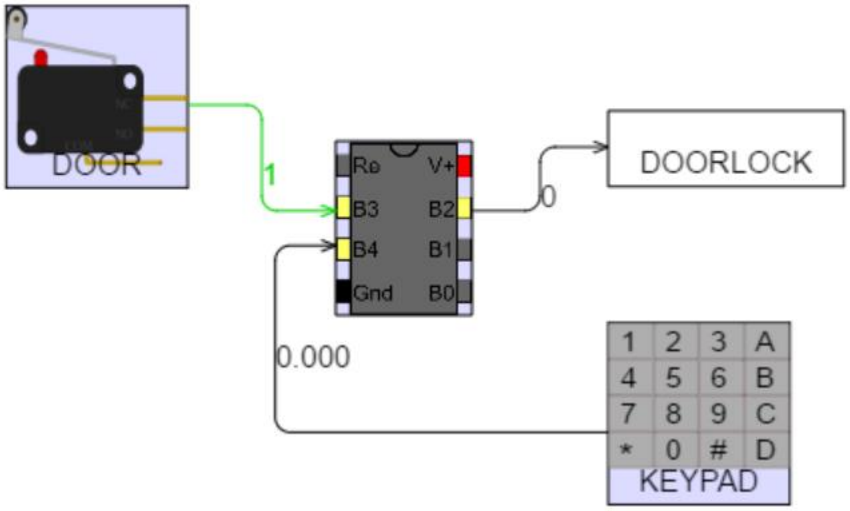
```
while (1){
    if (state == EW){
       //...
       if (PED_SW){
           state = PED;
       }
    }
    else if (state = NS){
        //...
    }
    else if (state == PED){
        //...
    }
}
```

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

A more comprehensive pattern uses a separate function for each state and calls the appropriate function from a table of function pointers, e.g. http://www.conman.org/projects/essays/states.html or http://johnsantic.com/comp/state.html

© Dr. Bill Collis 2021

2.14.2. **Electronic safe state machine**



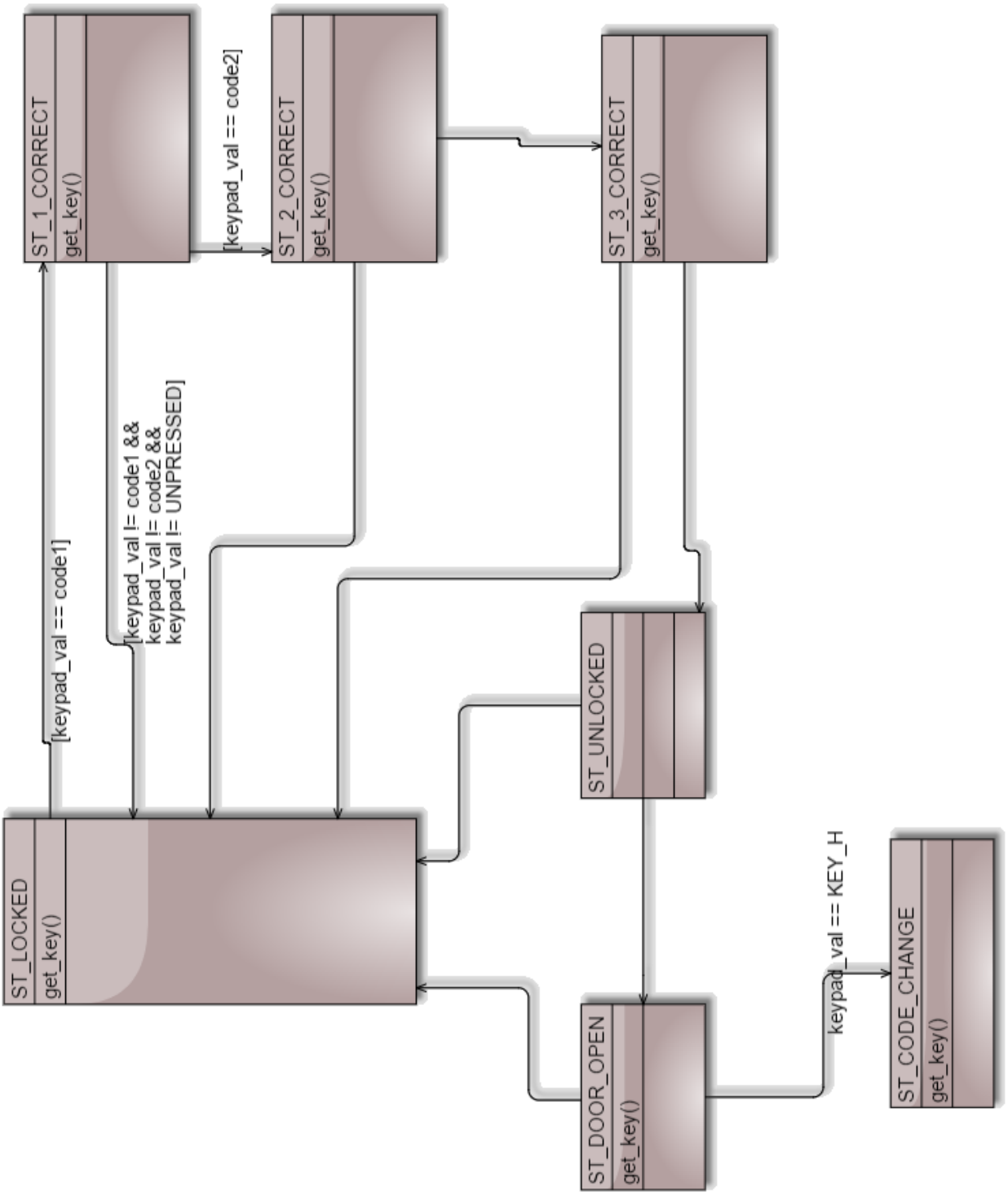
Operating the safe:

- 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 pressed 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);
}
```

© Dr. Bill Collis 2021



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();
        }
     }
```

© Dr. Bill Collis 2021

2.15. Driving output devices

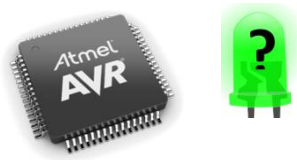
LO: list specifications of an uC that would inform choosing a uC for an ES

LEDs (Light Emitting Diodes)

Will it work?

Will it work?

Will it work?



How can we know?

we need to know how the characteristics of the devices interrelate with

each other, before we can engineer a solution – actually the equivalent circuit model for each

LED Specifications:

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

Vf typical: 2.0V at 20mA - this means it will draw 20mA and have approximately 2.0 V across it,

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?

Why does this interrelationship exist?

<u>Could / should we design circuits that use an over-clocked AVR?</u>

© Dr. Bill Collis 2021

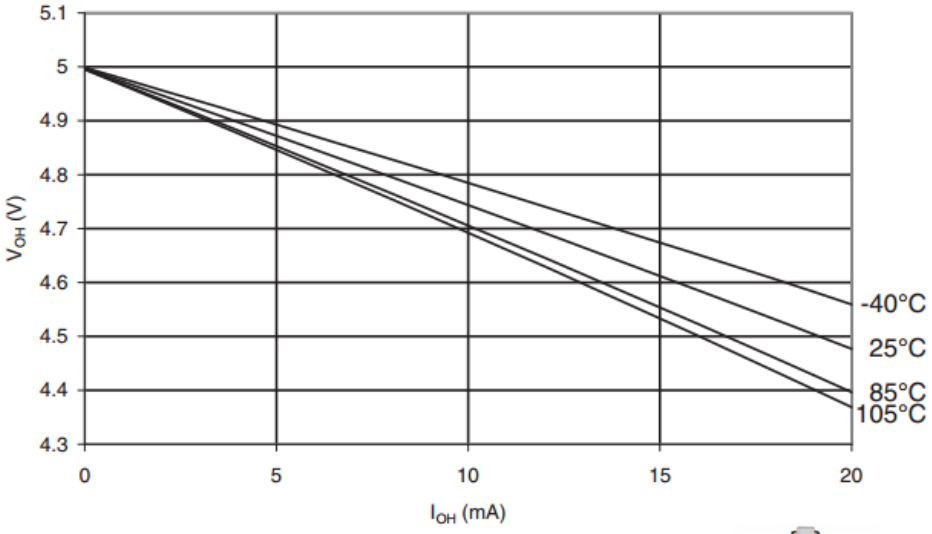
ATmega328P DC characteristics for -40 to +85 degC datasheet section 32.1

Absolute Maximum Ratings

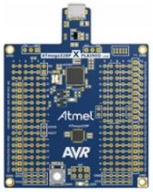
- DC Current per I/O Pin
- DC Current Vcc and GND pins

Although the absolute maximum specs are 40mA, the datasheet gives a graph of output voltage for different frequencies and output currents, that only goe 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.

© Dr. Bill Collis 2021

2.15.1. Driving devices with higher current / voltage requirements

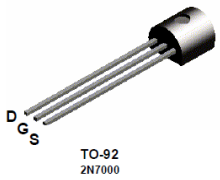
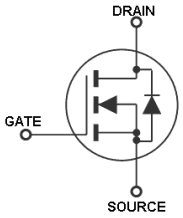
Many output devices require higher power than a microcontroller pin can typically provide.

The AVR can only supply $5V$ at a maximum current of $40mA$. This is too small to drive many output devices. We need an interface device that a microcontroller can switch that can control larger devices. One common device is the logic MOSFET (metal oxide silicon field effect transistor).

2.15.2. The logic MOSFET

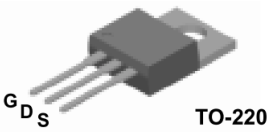
There are several different types of transistor –the BJT and the FET are two common ones. The only type considered in this course is the logic MOSFET, a specific type of FET that can be driven by voltages used in $3.3V$ and $5V$ logic circuits.

There are many different logic FETs made by very many different manufacturers, one common low power logic FET is the $2N7000$.

<table>
<tr><th>Physical appearance</th><th>Scematic symbol</th><th>Characteristics of interest</th></tr>
<tr>
<td>$2N7000$
 
</td>
<td> 
</td>
<td>
<table>
<tr><th>BV_{DSS} / BV_{DGS}</th><th>$R_{DS(ON)}$ (max)</th><th>$I_{D(ON)}$ (min)</th></tr>
<tr><td>60V</td><td>5.0Ω</td><td>75mA</td></tr>
</table>
<table>
<tr><th>I_D (continuous)*</th><th>I_D (pulsed)</th><th>Power Dissipation @ T_C = 25°C</th></tr>
<tr><td>200mA</td><td>500mA</td><td>1W</td></tr>
</table>
$2N7000$ datasheet
</td>
</tr>
</table>

One higher power logic FET is the $FQP30N06L$

$FQP30N06L$

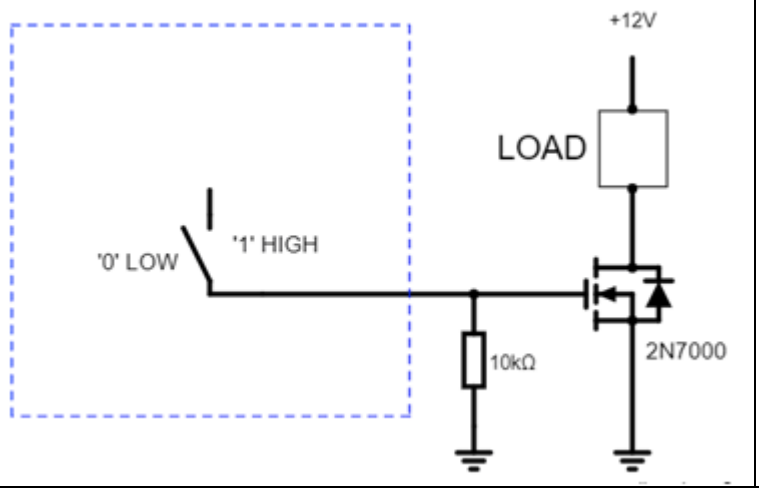


Symbol	Parameter		FQP30N06L	
V_{DSS}	Drain-Source Voltage		60	V
I_D	Drain Current	- Continuous (T_C = 25°C)	32	A
		- Continuous (T_C = 100°C)	22.6	A
I_{DM}	Drain Current	- Pulsed	128	A

Symbol	Parameter	Min	Typ	Max	
$V_{GS(th)}$	Gate Threshold Voltage	1.0	--	2.5	V
$R_{DS(on)}$	Static Drain-Source On-Resistance	-- --	0.027 0.035	0.035 0.045	Ω

© Dr. Bill Collis 2021

We connect a Logic FET to a microcontroller like this:

Schematic	Equivalent circuit
	

The 10k resistor is not necessary for the circuit to work, it is there because when the microcontroller turns on, it takes a finite amount of time for the software to set the pin as an output. During that time the pin is automatically setup to be an input. It could 'float' in potential, making the gate of the FET float as well. This means stray electric fields from the circuit around it could turn on the load.

Connecting a $150mW$ load to the circuit above.

What is the current in the load?

What is the voltage V_{DS}?

What is the power dissipated by the FET?

Replacing the $2N7000$ with a high power FET, such as the $FQP30N06L$, and changing to a $150W$ load - what is the current in the load?

What is the voltage V_{DS}?

What is the power dissipated by the FET?

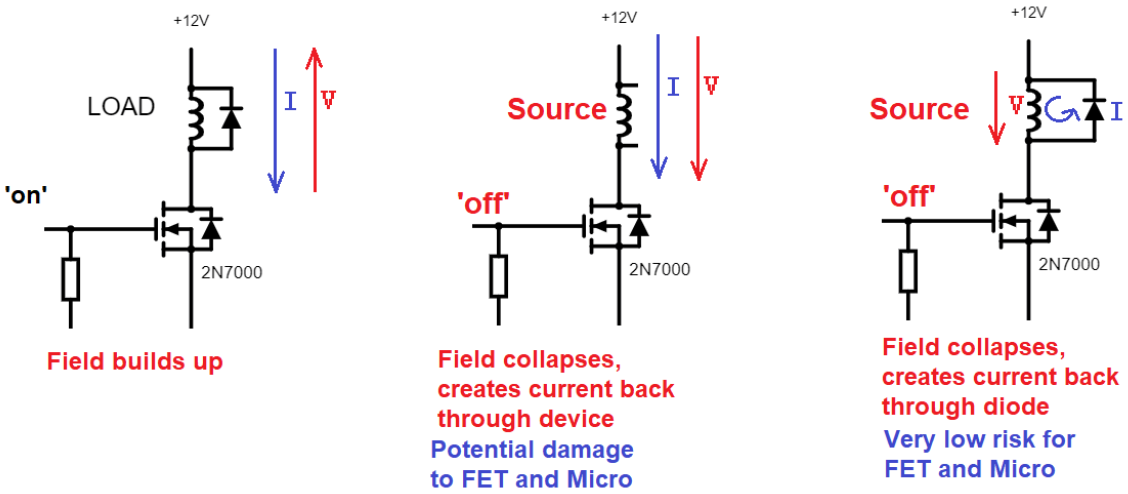
© Dr. Bill Collis 2021

2.15.3. **Inductive Loads**

Faraday's Law states that when the magnetic flux linking a circuit changes, an electromotive force (EMF) is induced in the circuit proportional to the rate of change of the flux linkage (this is the principle behind how generators work).

This however can have catastrophic effects in electronic circuits. If we connect a motor or solenoid (an inductor) to a microcontroller using a FET, when the FET is turned on and there is current in the inductive device and a magnetic field builds up (left diagram below). When the device is turned off the magnetic field collapses (rapidly) and creates an EMF. The effect is that the current tries to be maintained, this turns the inductor from a load to a source, and the voltage across the inductor reverses polarity – we use the term back-EMF to describe this reversed polarity. The inductive device becomes a small 'battery' for a short period of time – enough time to create enough current at a high enough voltage to damage the FET and the microcontroller - sometimes spectacularly! Centre diagram below.)

We always connect a 'flyback' diode across the load. A diode conducts when the polarity across it is in one direction and does not conduct when the polarity across it is in the other direction, right diagram below)



© Dr. Bill Collis 2021

2.16. Fitness-for-purpose

LO: recognise 'fitness-for-purpose' factors that guide uC choice for an ES

Understanding the nature of an ES (the interrelatedness of hardware and software, the unattended automaton, and as reactive and responsive to the environment) allows the engineer a mindset of what will make an ES 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 – and in the process 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

© Dr. Bill Collis 2021

2.17. Useful resources

'Atmel AVR microcontroller primer programming and interfacing' - Barrett & Pack
An excellent resource on how to setup AVR timers, ADC, Serial communications

'A Baker's Dozen' Bonnie Baker
An excellent book written by an IC designer

'Trouble shooting Analog Electronics' Bob Pease
This is an excellent book written by a IC designer from National Semiconductor Corporation

'Practical AVR Microcontrollers'
Trevennor has some interesting projects and excellent coverage of AVRDude

You can download the above books from the UoA library

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

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

AVR Tutorials
extremeelectronics.co.in/category/avr-tutorials/

Interrupts
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

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

Thanks for coming to the uC section of CS201; I have enjoyed teaching it and I hope you find this knowledge useful in your future as engineers.

© Dr. Bill Collis 2021