CG2271

Real-Time Operating Systems

Lecture 3

Microcontroller Programming

colintan@nus.edu.sg

NUS
National University
of Singapore

School of Computing

NUS National University of Singapore
School of Computing

Learning Objectives

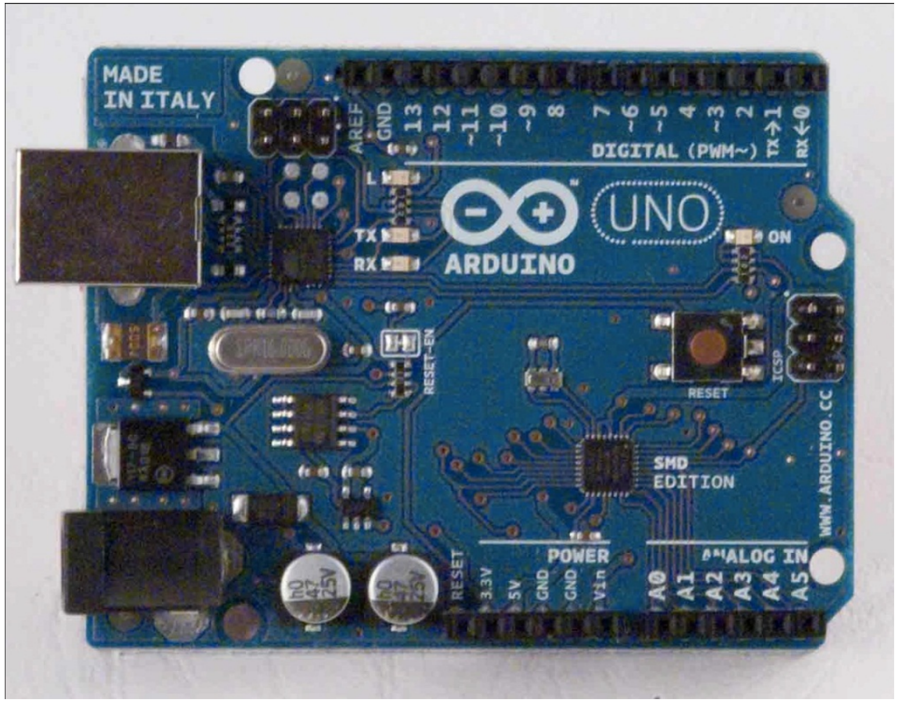
- **By the end of this lecture you will be able to:**
  - Understand the general architecture of a simple microcontroller.
  - Read and write digital devices using the General Purpose I/O (GPIO) ports.
  - Read analog devices using the Analog-to-Digital Conversion (ADC) ports.

Learning Objectives

- **By the end of this lecture you will be able to:**
  - Control programs using the on-chip timers.
  - Write to analog devices using the Pulse-Code-Modulation (PCM) ports.
- **Why?**
  - You will be able to write the device layer of an RTOS OR customize an RTOS for the AVR.
- **Why AVR?**
  - Cheap, relatively simple to program.
  - Concepts learnt here are applicable to other microcontrollers, like the Atmel AT91SAM7 and STR91 ARM-based microcontrollers.

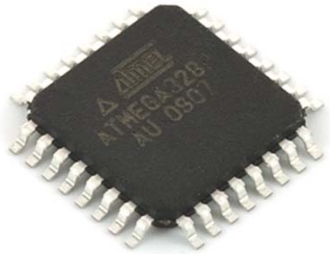
Introduction

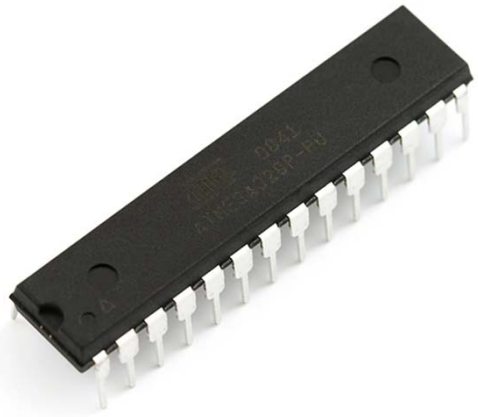
- **In this lecture you will be introduced to microcontroller programming on an AVR microcontroller.**
  - We will be using the Atmel Atmega328, a tiny 16 MHz microcontroller with 32KB of flash memory and 2 KB of RAM.
  - Very commonly used in real-time systems, and in the Sparkfun Inventor Kits that will be issued to each team. ☺



Lecture Outline

- **Preliminaries.**
  - The binary number system.
  - Bit manipulation in C.
- **The Atmel Atmega328 AVR Microcontroller.**
  - Architecture and Specifications.
  - General Purpose I/O.
  - Analog to Digital Conversion.
  - Programming the on-chip timers.
  - Programming PWM Ports.





AVR Programming

PRELIMINARIES

Preliminaries
Binary

- **Since some of you have no computer organization background, we will cover the basics of binary and hexadecimal, and masking.**
  - In computers, for reliability reasons, all data is represented using 0-volts (0) and 5-volts (1). This is binary.
  - Conversion from decimal to binary is through a process called “repeated divide”.
  - Using this technique, a value like 213 in binary is 0b11010101.
    - ✓**The “0b” indicates a binary number. Can be written 11010101b, which means the same thing.**

NUS National University of Singapore
School of Computing

Preliminaries
Hexadecimal

- **A long binary string is often difficult to deal with. So we deal with a different number system called "hexadecimal", which is a base-16 system.**
  - Divide the binary bits into groups of 4, from right to left.
  - Use the table on the right to convert each group into a hexadecimal digit.

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Preliminaries
Hexadecimal

- **In our previous example, we convered 213 to 0b11010101.**
  - Split into groups of 4 bits from the right. 1101 0101.
  - Use the table and get: 1101=D, 0101=5.
  - So 213 = 0b11010101 = 0xD5.
    - ✓**The "0x" indicates a hexadecimal number. Can also be written D5h which means the same thing.**

NUS National University of Singapore
School of Computing

Preliminaries
Bit-wise Operators in C

- **C has the following bitwise operators:**
  - AND (&), OR (|) and NOT (~)
- **The table below shows how these work.**

A	B	A&B	A|B	~A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Preliminaries
Masking

- **The bit-wise operators are very useful for setting or clearing a particular bit.**
  - To clear a bit, create a “mask” where that bit is 0 and all others are 1, and do a bit-wise AND.
  - To set a bit, create a “mask” where that bit is 1 and all others are 0, and do a bit-wise OR.
- **We will consider a variable “value” set to 0b1101 1110 (0xDE). We want to clear bit 3 and set bit 5.**
  - Bits are numbered 0 to 7 from right to left.

NUS National University of Singapore
School of Computing

Preliminaries
Masking

- **Clearing bit 4:**

Value	1	1	0	1	1	1	1	0
	&	&	&	&	&	&	&	&
Mask	1	1	1	1	0	1	1	1
Result	1	1	0	1	0	1	1	0

Our mask here is 0b1111 0111 or 0xF7

- **Setting bit 2:**

Value	1	1	0	1	0	1	1	0
	|	|	|	|	|	|	|	|
Mask	0	0	1	0	0	0	0	0
Result	1	1	1	0	0	1	1	0

Our mask here is 0b00100000 or 0x20

NUS National University of Singapore
School of Computing

Preliminaries
Masking

- **The C code to accomplish this is shown below:**

```
int value=0xde;

…
// Clear bit 4
value  &= 0xF7;
// Set bit 2
value |= 0x20;
…
```

AVR Programming

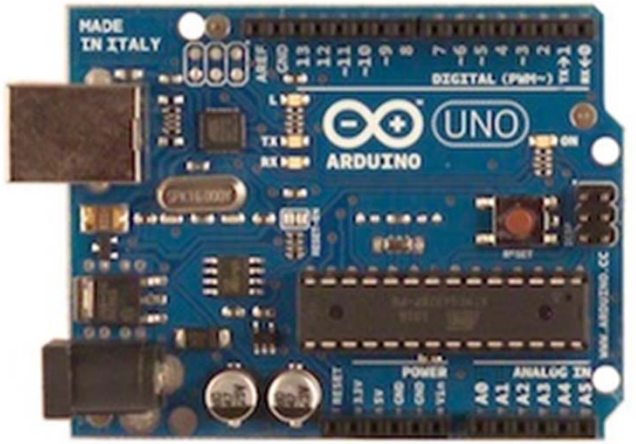
THE ATMEL AVR AND ARDUINO

The Atmel AVR

- **The Atmel AVR (Advanced Virtual RISC, or Alf-Vegard RISC) processor is a family of 8 and 32 bit microcontrollers.**
  - ATTiny
    - ✓**20 MHz clock.**
    - ✓**8-bit core.**
    - ✓**0.5-8KB of memory.**
  - ATMega (which you are using)
    - ✓**Like ATTiny, but with 4-256KB of memory and more I/O options and larger instruction set.**
  - AVR32
    - ✓**66 MHz clock**
    - ✓**32-bit core.**
    - ✓**512 KB of memory.**

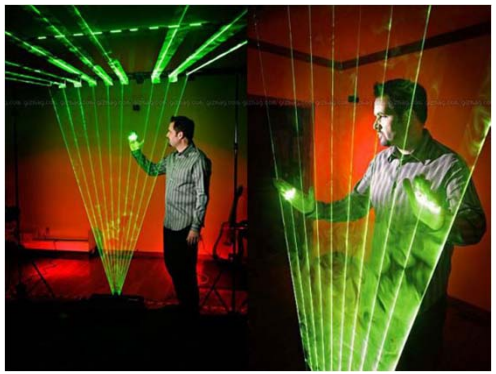
AVR and Arduino

- **For this course you will be using a platform called "Arduino", based on the Atmel Atmega328 microcontroller.**
  - Open-source hardware and software platform aimed at designers, hobbyists, artists.
    - ✓**Comes with a simple-to-use Arduino library, which you will use for the first few labs.**
    - ✓**Library will be slowly replaced by programs you write yourself, as you learn to program "bare-metal".**
  - Hardware consists of 13 digital I/O pins, 6 analog input pins, 5 PWM analog output pins and a USART port.



AVR Applications

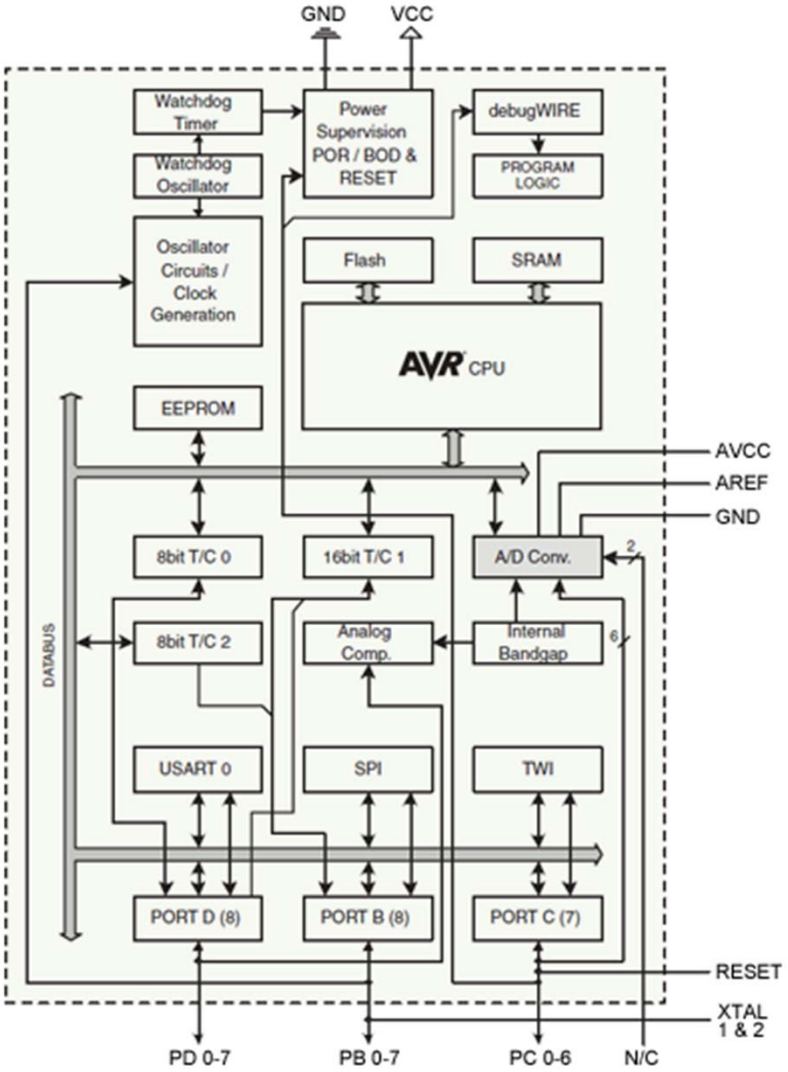
- **AVR is used in industrial automation. Unfortunately information is hard to come by, so here are some cool Arduino projects instead. ☺**
  - Laser Harp
  - Autopilot Systems
  - Plant Twitter System
  - High Speed Photography



NUS National University of Singapore
School of Computing

Atmel Atmega Architecture

- **Specifications:**
  - 20 MHz 8-bit microcontroller
  - 32 single-byte registers.
  - I/O Options:
    - ✓**23 general purpose I/O (GPIO) lines.**
    - ✓**6 analog-digital converter channels (analog in).**
    - ✓**3 PWM channels (analog out).**
    - ✓**2 8-bit counters and 1 16-bit counter.**
    - ✓**USART (RS232-like interface)**
    - ✓**Two-wire Interface (TWI).**
    - ✓**Serial Peripheral Interface (SPI).**

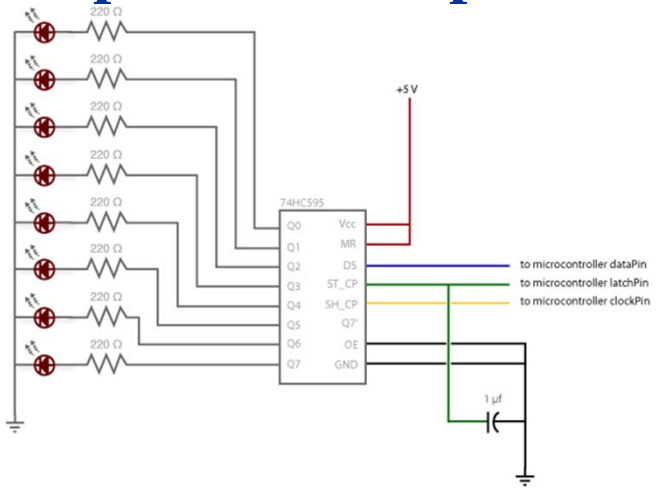


AVR Programming

GPIO PROGRAMMING

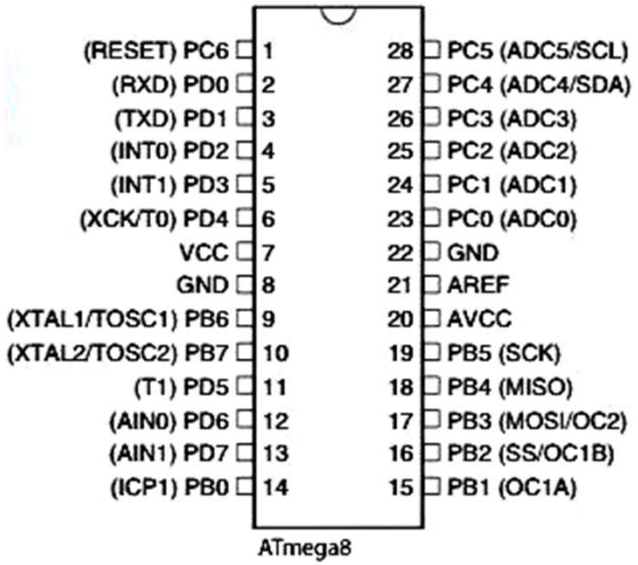
GPIO Programming

- **The AVR General Purpose Input Output (GPIO) ports are used for communicating with digital inputs and outputs:**
  - Switches.
  - LEDs.
  - Digital circuits.
  - Etc. etc.



GPIO Programming

- **There are three GPIO ports labeled PORTB, PORTC, and PORTD, corresponding to pins on the AVR.**
  - Pins are labeled PB0-7 (8 lines), PC0-6 (7 lines) and PD0-7 (8 lines), totaling 23 pins.
  - These pins are also shared with other functions:
    - ✓**E.g. PD0 and PD1 are also used by the receive (RXD) and transmit (TXD) lines for the USART.**



GPIO Programming

- **To program a GPIO port, we must first set the direction of the individual pins. This is done using the DDRx registers:**
  - Example shown is for PORTB:

DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
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	

  - A “1” in bit 6 will set DB6 to be output, a “0” will set it to be input.

GPIO Programming

- **Once the pin directions have been configured, you can use the PORTx registers to read or write the pins.**

PORTB – The Port B Data Register

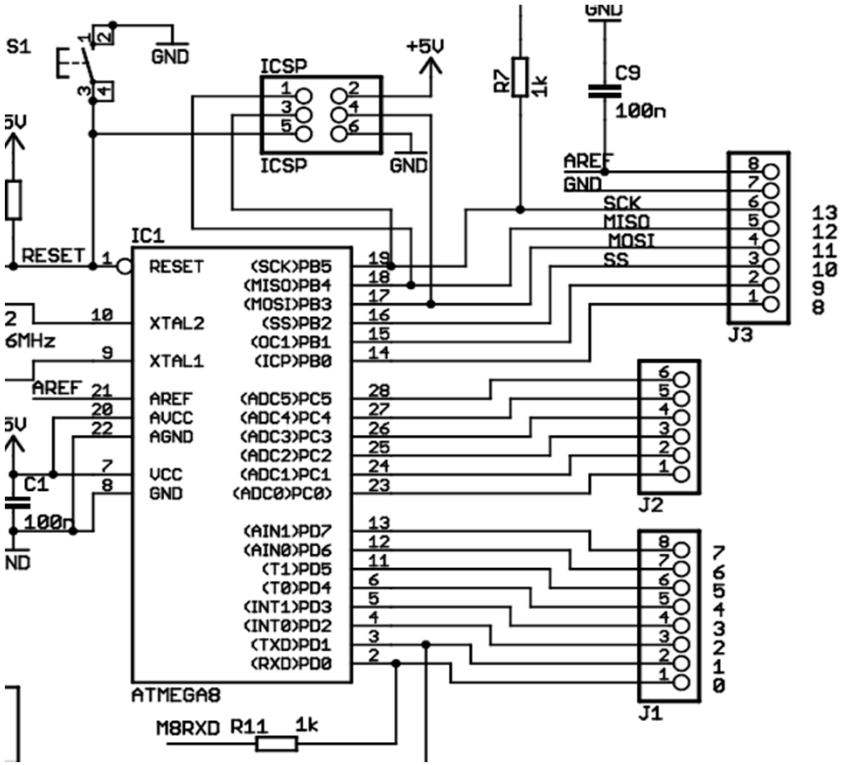
Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- If bit 5 in DDRB was set to 0, and a device was asserting pin PB5 HIGH, then reading register PORTB will result in a "1" being read in bit 5.
- If bit 3 in DDRB was set to 1, and if your program wrote a "1" to bit 3 in PORTB, then pin PB3 will be asserted HIGH.

GPIO Programming

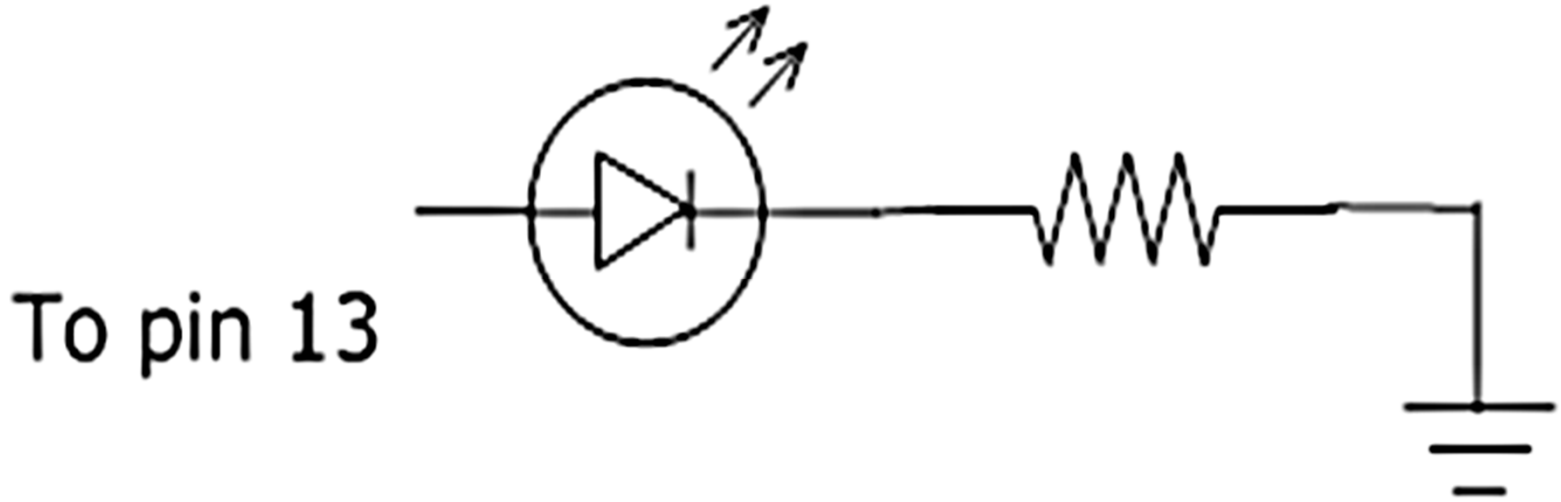
- **The GPIO ports on the Atmega are mapped to Arduino digital pins using the following table:**

Arduino Pin	Atmega 328 port and pin number
0	Port D, pin 0
1	Port D, pin 1
2	Port D, pin 2
3	Port D, pin 3
4	Port D, pin 4
5	Port D, pin 5
6	Port D, pin 6
7	Port D, pin 7
8	Port B, pin 0
9	Port B, pin 1
10	Port B, pin 2
11	Port B, pin 3
12	Port B, pin 4
13	Port B, pin 5



GPIO Programming Example 1

- **In this example we will assume that an LED has been connected to Arduino pin 13, using the following circuit.**

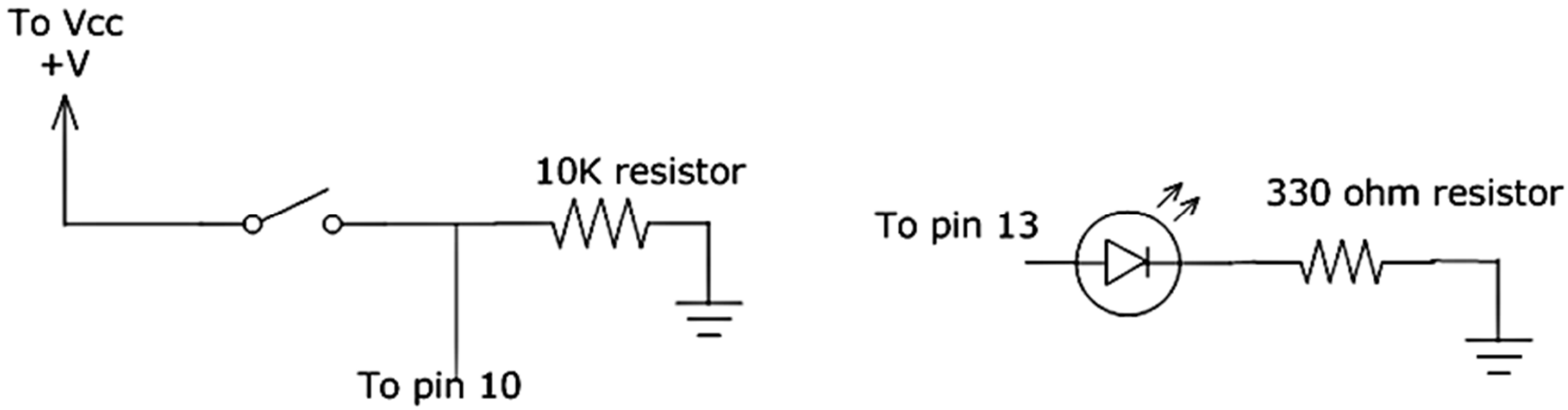


GPIO Programming Example 1

```
#include <avr/io.h>
#include <util/delay.h>

void ledOn()
{
        PORTB|=0b00100000;
}
 void ledOff()
{
        PORTB&=0b11011111;
}
 int main()
{
        DDRB |= 0b00100000;
        while(1)
        {
                ledOn();
                _delay_ms(1000);
                ledOff();
                _delay_ms(1000);

        }
}
```

GPIO Programming Example 2

- **Same program as before, except that this time we toggle the LED on and off using a push button switch connected to pin 10 using this circuit:**



GPIO Programming Example 2

```
#include <avr/io.h>
#include <util/delay.h>

int buttonVal=0;

void ledOn()
{
        PORTB|=0b00100000;
}

void ledOff()
{
        PORTB&=0b11011111;
}
```

GPIO Programming Example 2

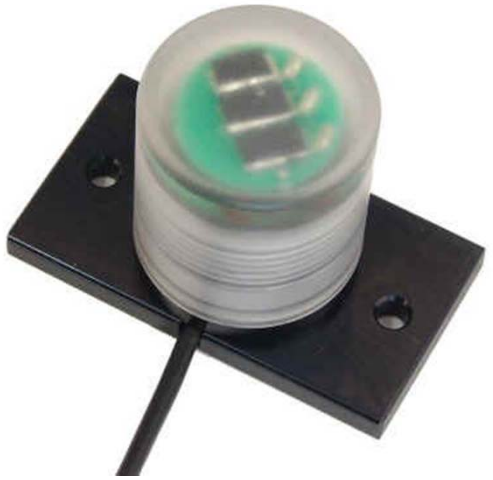
```
int main()
{
        DDRB |= 0b00100000;
        DDRB &= 0b11111011;

        while(1)
        {
                buttonVal=PORTB & 0b00000100;

                if(buttonVal)
                        ledOn();
                else
                        ledOff();
                _delay_ms(100);
        }
}
```

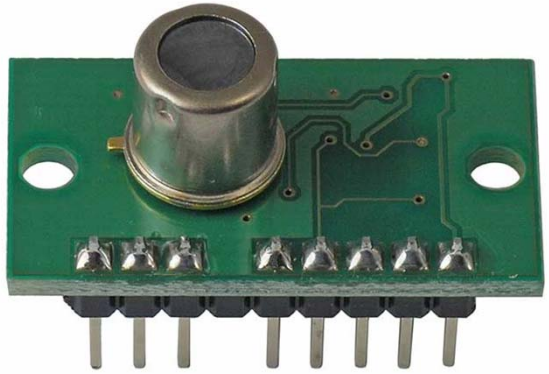
AVR Programming

ANALOG INPUT PROGRAMMING

Analog Input Programming

- **GPIO Ports:**
  - Great for reading digital (0 and 1) inputs or writing to devices that are either only fully on or off.
  - Not great for inputs that vary continuously over a range.
  - Examples:
    - ✓**Thermometers whose readings vary within a temperature range.**
    - ✓**Light sensors whose outputs vary according to light conditions.**
    - ✓**Voltage sensors.**



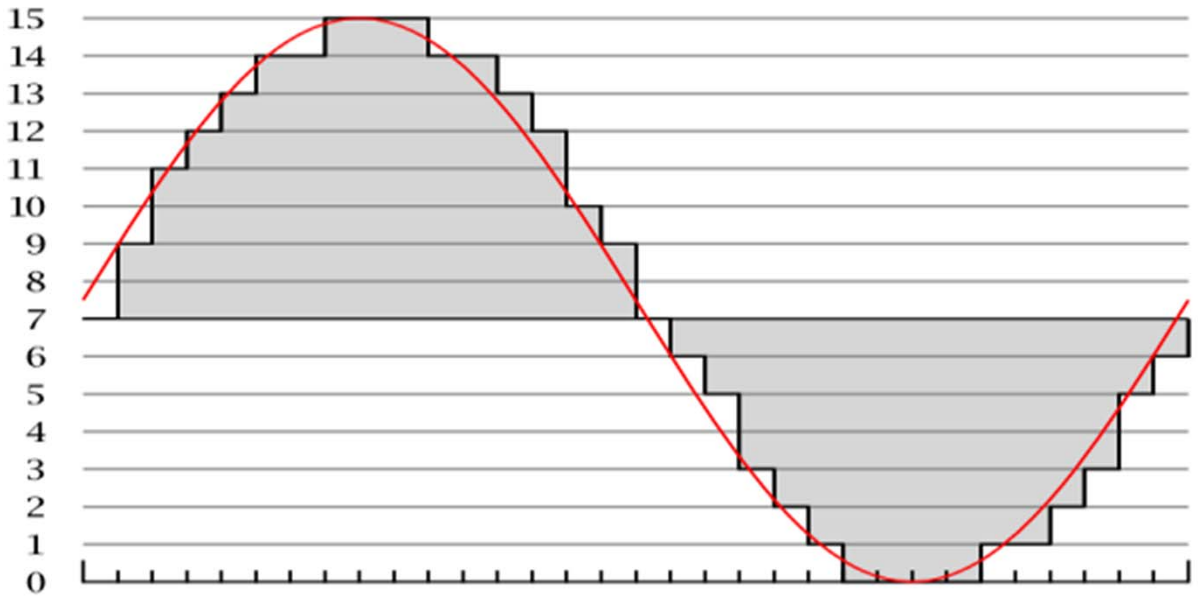


Analog Input Programming

- **To deal with analog inputs that produce values that vary within a range, the AVR provides 6 analog-to-digital converters.**
  - These use pulse-code-modulation (PCM) to convert the analog signal into a number:
    - ✓**Input voltage range: 0 to Vcc (either 3.3V or 5V)**
    - ✓**Output Range: 0 to 1023.**
  - We will briefly look at PCM to understand what it is.

Analog Input Programming
Pulse Code Modulation

- **In PCM:**
  - A sample voltage is taken from the input every *t* seconds.
    - ✓**Typical value for *t* is 0.000625 for a 16 KHz sample rate.**
  - The voltage read in is then converted to a digital number, using a curve similar to the diagram below (for a 4-bit ADC)



Analog Input Programming

- **To program the ADC on the AVR, the following steps need to be taken:**
  1. Activate power to the ADC:
     - ✓**Write a "0" to bit 0 (ADC) of the Power Reduction Register PRR.**
  2. Switch on the ADC:
     - ✓**Write a "1" to bit 7 (ADEN) of the ADC Control and Status Register ADCSRA.**
  3. Choose which channel you want to read from, and the reference voltage source.
     - ✓**This is done in the ADC Multiplexer Register ADMUX.**

Analog Input Programming

4. Start the conversion:
   - ✓**Write a "1" to bit 6 (ADSC) of ADCSRA.**
5. Wait until the conversion ends.
   - ✓**Poll bit 6 of ADCSRA until it becomes 0.**
6. Read in the converted value.
   - ✓**Read in bits 7-0 from register ADCL, and combine with bits 9-8 from register ADCH.**
7. GOTO 4 until desired number of values are converted.

Analog Input Programming
Sending Power to the ADC

- **The Power Reduction Register is used to turn off power to parts of the AVR, to conserve energy.**

Bit	7	6	5	4	3	2	1	0	
(0x64)	PRTWI	PRTIM2	PRTIM0	–	PRTIM1	PRSPI	PRUSART0	PRADC	PRR
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	

- **To turn on power, write a "0" to the bit corresponding to the device you want to switch on. In this case bit 0 (PRADC) corresponds to the ADC.**

```
PRR&=0b11111110;
```

Analog Input Programming
Switching on the ADC

- **Now that power is being supplied to the ADC, we must switch it on by writing a "1" to bit 7 of ADCSRA:**

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

- **We also need to set a prescalar value.**
  - This determines the sampling frequency, which is given by:

$$f_s = \frac{f_{clk}}{ps}$$

NUS National University of Singapore
School of Computing

Analog Input Programming
Switching on the ADC

- **The prescalar value *ps* is specified using bits *ADPS2:0* in ADSCRA, using the following table:**

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

Analog Input Programming
Switching on the ADC

- **The standard clock rate on the Arduino UNO is 16 MHz.**
- **We will use a pre-scale value of 0b111 (128), giving us a 16,000,000/128 = 125KHz sampling rate.**
- **For now we will ignore the ADSC, ADATE, ADIF and ADIE bits, setting these to 0.**
- **The C statement to set up ADSCRA is therefore:**

```
ADCSCRA = 0b10000111;
```

NUS National University of Singapore
School of Computing

Analog Input Programming
Setting up the ADMUX Register

- **The ADMUX register lets you choose your reference voltage, as well as which channel to convert:**

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
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	

- **The REFS1 and REFS0 bits tell the ADC which reference voltage to take, for conversion.**

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V Voltage Reference with external capacitor at AREF pin

Analog Input Programming
Setting up the ADMUX Register

- **We will use the AVR's own internal reference voltage (equal to Vcc), and will therefore set REFS1:0 to 01.**
- **We can choose the conversion channel using this table:**

Channel	MUX2	MUX1	MUX0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Analog Input Programming
Setting up the ADMUX Register

- **For converting on the Arduino Uno, we will ignore MUX3 and ADLAR, and set these bits to 0.**
- **The C statement to configure ADMUX to use the internal reference voltage and convert channel 2 is therefore:**

```
ADCMUX=0b01000010;
```

Analog Input Programming
Starting the Conversion

- **Now that we've set everything up, we need to start the conversion. To do this we set the ADSC bit (bit 6) in ADSCRA to a 1.**

```
ADCSRA |= 0b01000000;
```

- **Loop until the ADCS bit returns back to 0, signaling end of conversion.**

```
while(ADCSRA & 0b10000000);
```

Analog Input Programming
Clearing ADIF/Reading the Result

- **The ADIF (ADC Interrupt Flag) must be cleared when the ADC is in programmed I/O mode (like in this example). Clear the flag by writing a "1" (Yes, a ONE) to the ADIF bit (bit 4) in ADSCRA:**

```
ADCSRA |= 0b00010000;
```

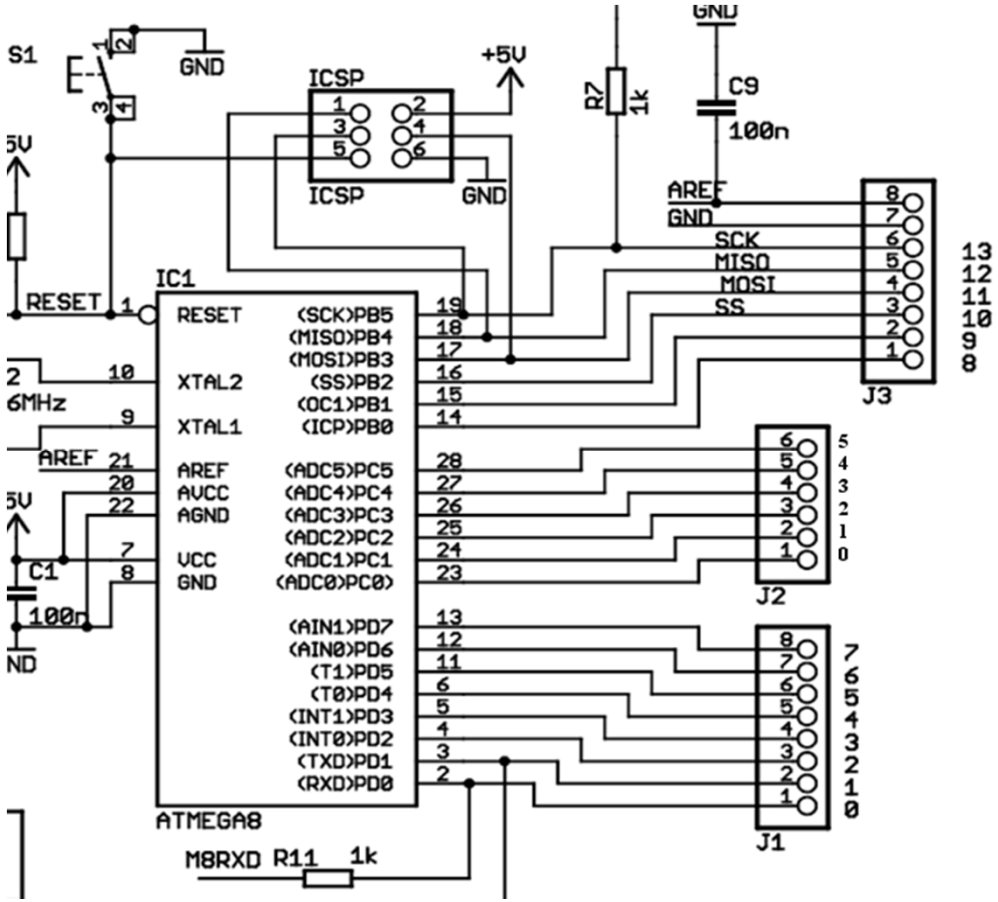
- **Read the ADCL and ADCH registers to get the converted value.**
  - IMPORTANT: Reading from ADCH causes the ADC to over-write both ADCL and ADCH with new values.
  - ALWAYS read ADCL first or you will lose the data there!

Analog Input Programming
Clearing ADIF/Reading the Result

```
loval=ADCL;
hival=ADCH;
adcval= hival * 256 + ADCL;
```

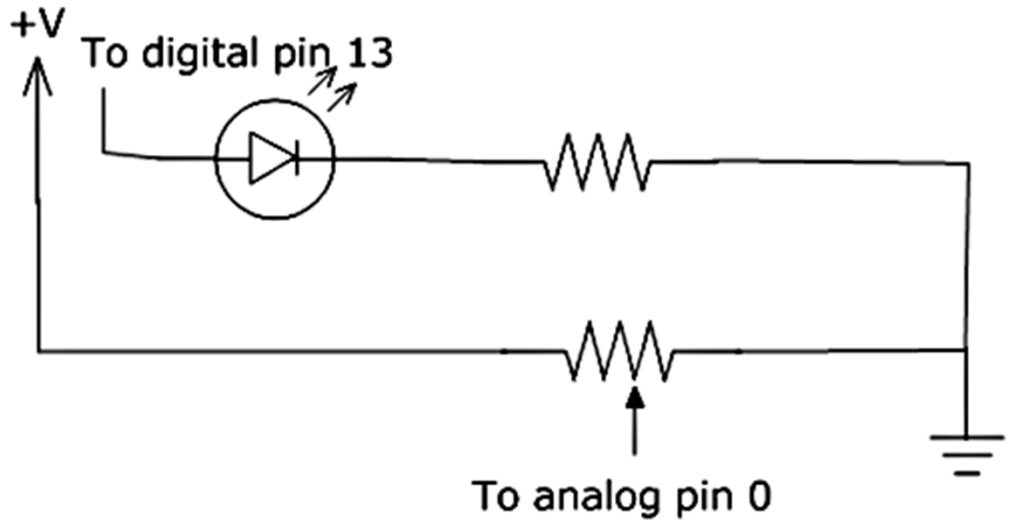
Mapping between AVR and Arduino ADC Channels

- **As shown in the reference circuit for the Arduino Uno, the mapping is direct**
  - ADC channel 0 maps to Arduino analog input 0, etc.



Analog Input Programming Example 1

- **We will now write a program that reads in a voltage reading from analog pin 0, and uses the read value to control the rate of blinking of an LED connected to digital pin 13.**



Analog Input Programming Example 1

```
#include <avr/io.h>
#include <util/delay.h>

void ledOn()
{
        PORTB|=0b00100000;
}

void ledOff()
{
        PORTB&=0b11011111;
}
```

Analog Input Programming Example 1

```
int main()
{
        unsigned adcvalue, loval, hival;

        // Set up ADC
        // Write 0 to PRR bit 0 to disable power reduction on the ADC
        PRR &= 0b11111110;

        // Enable ADC, don't start conversion, disable ADIF and ADIE and
        // set ADPS2-0 to 111 to set prescalar value of 128
        ADCSRA=0b10000111;

        // Set up ADMUX to convert from channel 0.
        ADMUX=0b01000000;

        // Set up LED on pin 13 (port B pin 5) for output.
        DDRB|=0b00100000;
```

Analog Input Programming Example 1

```
        while(1)
        {
                // Start ADC conversion by writing a 1 to ADSC bit.
                ADCSRA |= 0b01000000;
                // Wait for conversion to end.
                while(ADCSRA & 0x01000000);

                // Clear ADIF, Read ADC value
                ADSCRA |=0b00010000;
                loval=ADCL;
                hival=ADCH;
                adcvalue = hival*256+loval;

                ledOn();
                _delay_ms(adcvalue);
                ledOff();
                _delay_ms(adcalue);
        }
} // main
```

Analog Input Programming
Interrupt Programming

- **In the previous example we saw how to do ADC using programmed I/O.**
  - I.e. we polled the ADSC bit in ADSCRA until it became 0, signaling the end of conversion.
- **In this example we will look at how to do interrupt driven programming with the ADC.**
  - At the same time we will learn interrupt programming for the AVR.

Interrupt Programming on the AVR

- **To handle interrupts, we must first include the necessary macros:**

```
#include <avr/interrupt.h>
```

- **Now create an interrupt service routine by using the ISR macro**

```
ISR(irq_vect)
{
        ... ISR body ...
}
```

Interrupt Programming on the AVR

- **The *irq_vect* parameter specifies which interrupt this ISR should be tied to. Some example values are:**

Vector	Use
ADC_vect	To handle end-of-conversion interrupt from the Analog/Digital Converter.
INT0_vect	Handle IRQ0 (pin 3 on the Seeeduino)
INT1_vect	Handle IRQ1 (pin 4 on the Seeeduino)
BADISR_vect	Handle cases where an interrupt has no ISR.

- **The AVR allows you to disable ALL interrupts using a single flag in the status register SREG.**

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	

Interrupt Programming on the AVR

- **This register tells you important information like :**
  - Whether the previous computation resulted in:
    - ✓**An overflow (V), bit 3.**
    - ✓**A zero (Z), bit 1.**
    - ✓**…**
  - The important bit here is bit 7 (I), which must be set to “1” to ensure that interrupts are not globally switched off. You can do this by using the sei() macro.

```
sei();
```

  - Switching off interrupts is done with cli();.

Analog Input Programming
Interrupt Programming

- **ADC setup and use proceeds EXACTLY the same as before, EXCEPT:**
  - When configuring the ADCSRA, bit 3 (ADIE) must now be set to “1” instead of 0 to enable ADC interrupts.
  - We no longer have to poll bit 6 (ADSC) of the ADCSRA.
  - We no longer have to write a “1” to bit 4 of the ADCSRA to clear the ADIF.
    - ✓**This is automatically done when the ISR is triggered.**
- **We now look at Example 1, re-written to use interrupts.**

Analog Input Programming Example 2

- **The following changes were made to Example 1:**
  - We #include <avr/interrupt.h>
  - The “adcvalue” variable has been made a global variable, to allow sharing between main and the ISR.
  - An ISR has been added to handle end-of-conversion tasks.
    - ✓**Read ADCL and ADCH.**
    - ✓**Re-starts the conversion by writing a 1 to the ADSC bit in ADSCRA.**
  - The while loop inside main now consists solely of lines to turn the LED on and off.

NUS National University of Singapore
School of Computing

Analog Input Programming Example 2

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr.interrupt.h>

void ledOn()
{
        PORTB|=0b00100000;
}

void ledOff()
{
        PORTB&=0b11011111;
}
```

Analog Input Programming Example 2

```
ISR(ADC_vect)
{
        unsigned loval, hival;

        // Read the result from the registers
        // Read ADC value
        loval=ADCL;
        hival=ADCH;
        adcvalue = hival*256+loval;

        // Re-start the conversion
        ADCSRA |= 0b01000000;
}

unsigned adcvalue;
```

Analog Input Programming Example 2

```
int main()
{

        // Set up ADC
        // Write 0 to PRR bit 0 to disable power reduction on the ADC
        PRR &= 0b11111110;

        // Enable ADC, don't start conversion, enable ADIE (ADC
        // interrupt) and set ADPS2-0 to 111 to set prescalar value
        // of 128
        ADCSRA=0b10100111;

        // Set up ADMUX to convert from channel 0.
        ADMUX=0b01000000;

        // Set up LED on pin 13 (port B pin 5) for output.
        DDRB|=0b00100000;
```

Analog Input Programming Example 2

```
        // Start ADC conversion by writing a 1 to ADSC bit.
        ADCSRA |= 0b01000000;

        while(1)
        {
                ledOn();
                _delay_ms(adcvalue);
                ledOff();
                _delay_ms(adcalue);
        }
}// main
```

AVR Programming

TIMER PROGRAMMING

Timer Programming

- **Timers are important in real-time systems:**
  - They allow us to read sensors or write to actuators at precise times.
  - They ensure that time-sensitive algorithms (e.g. the PID control algorithm) run at the correct timing.
  - Timers are often needed to switch between tasks in a multi-tasking OS (future topic).
  - Timers are used to generate analog output signals through pulse-width modulation (PWM).

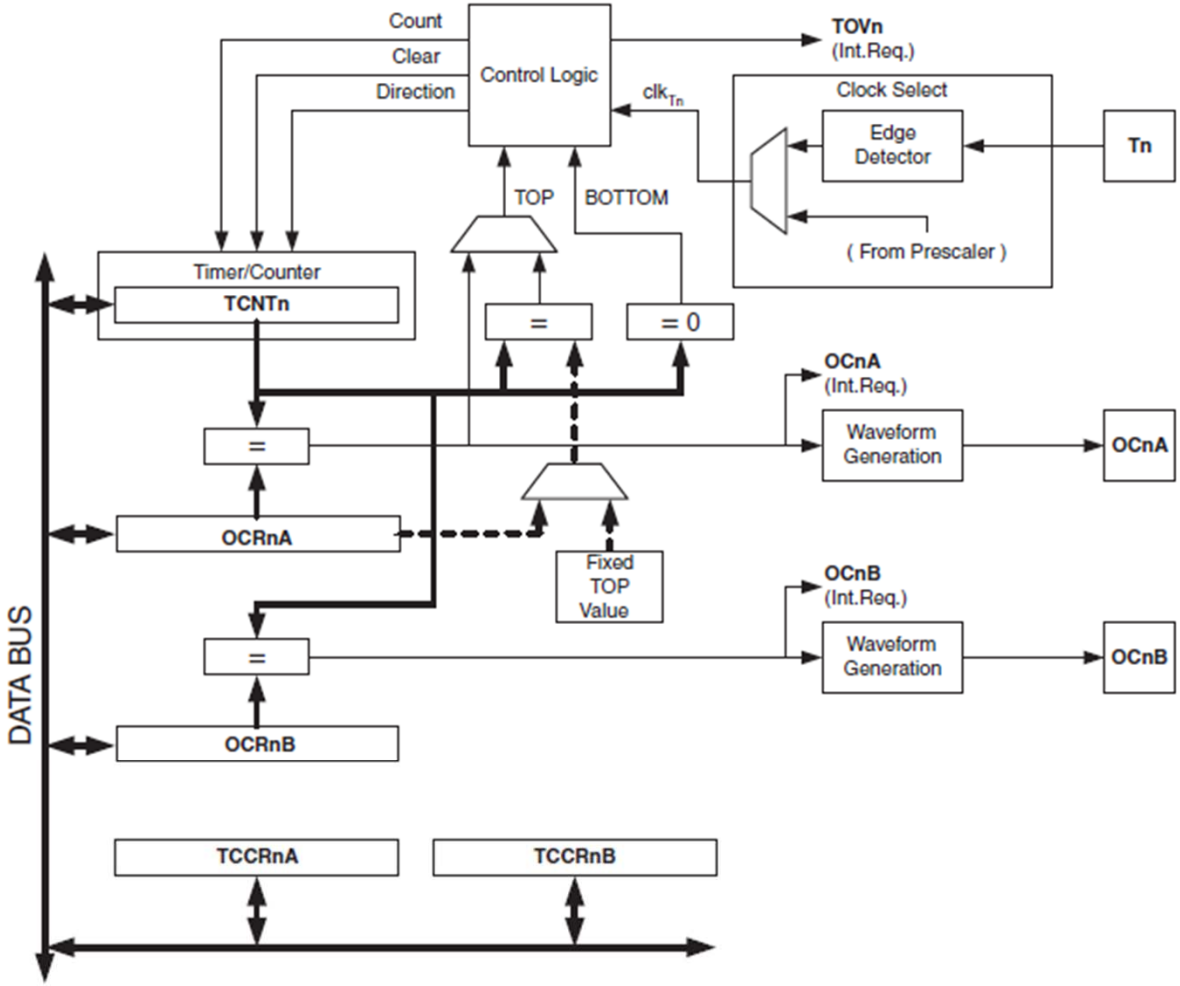
Timers in the Atmega AVR

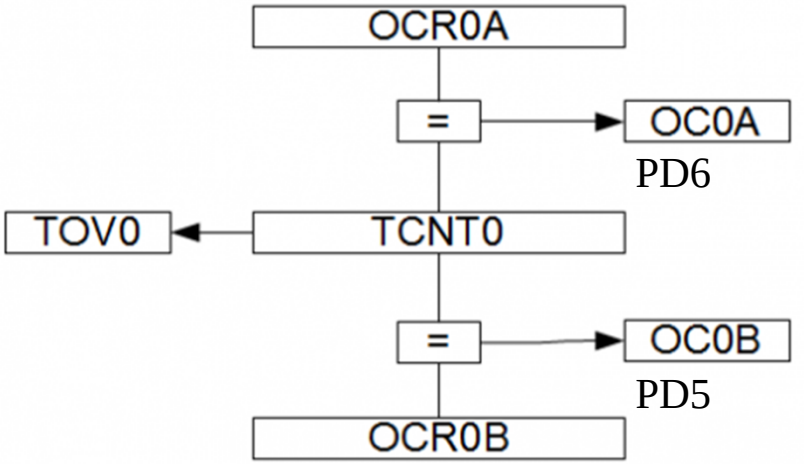
- **There are 3 timers available on the Atmega328:**
  - Two 8-bit timers, Timers 0 and 2.
    - ✓**Counts from 0 to 255, then back to 0, etc.**
    - ✓**Triggers an interrupt (TOV0 or TOV2) whenever the counter rolls over from 255 to 0. Also triggers interrupts if counter matches a value.**
    - ✓**One 16-bit timer, Timer 1.**
    - ✓**Counts from 0 to 65535 and back to 0, etc.**
    - ✓**Triggers an interrupt (TOV1) whenever the counter rolls over from 65535 to 0. Also triggers interrupts if counter matches a value.**
  - In this lecture we will focus on Timer 0.
    - ✓**Techniques for Timer 1 and 2 are identical, just with different registers.**

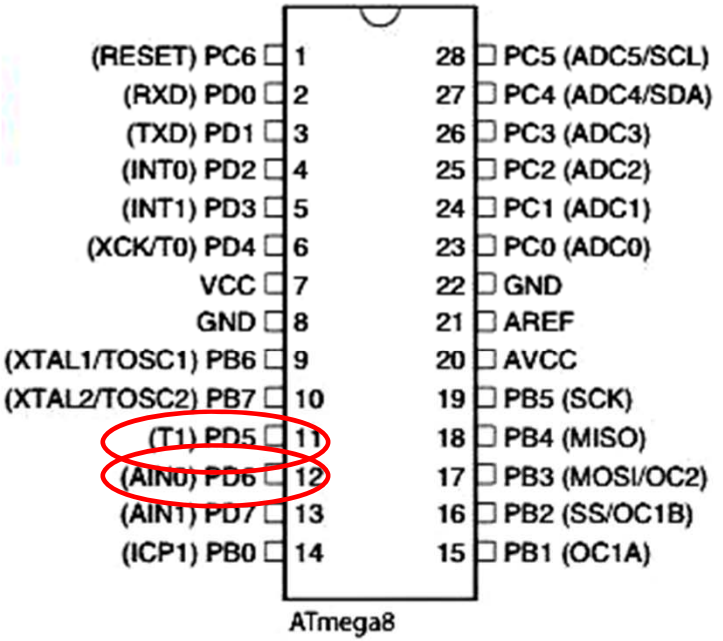
School of Computing

Timer 0/Timer 2 Block Diagram

Diagram Credits: Midity. http://www.midity.com







Timer 0 Programming
CTC Mode

- **The (second) simplest timer mode is CTC (Clear Timer on Compare) mode.**
  - Counter TCNT0 starts at 0, counts to a preset "top value" *V*, then rolls over back to the 0. Process repeats.
  - A "timer match" interrupt is triggered each time TCNT0 matches OCR0A or OCR0B.
- **This is very useful for generating an interrupt at fixed times, e.g. every 1 ms.**
  - Perfect for devices that need to be read at fixed times.
  - Perfect also for programs that need to "sleep" for fixed times.

Timer 0 Programming
CTC Mode

- **To program CTC mode in Timer 0, you need to:**
  - Decide on the time period that you want, e.g. 1 ms.
  - From this:
    - ✓**Decide on a prescaler value *P***
    - ✓**Decide on the count value *V*.**
  - Configure the registers.
    - ✓**Set up the ISR for OC0A (Timer 0 Output Compare Match A).**
    - ✓**Set the initial timer value of 0 into TCNT0.**
    - ✓**Set the top count value *V* into OCR0A.**
    - ✓**Set CTC mode in TCCR0A.**
    - ✓**Set the prescaler into TCCR0B to start the timer.**

Timer 0 Programming
CTC Mode

- Deciding on a prescaler value P
  - ✓**This determines the resolution *res* of the timer. I.e. the amount of time between "increments" of the counter.**
  - ✓**This is given by:**

$$res = \frac{F_{clock}}{P}$$

Timer 0 Programming
CTC Mode

- The prescalar is chosen using bits 2 to 0(CS02:00) of TCCR0B:

$$\text{Resolution} = \frac{1}{(F_{clk} / P)}$$

CS02	CS01	CS00	Prescalar P	Resolution (F_{clk}=16 MHz)
0	0	0	Stops the timer	-
0	0	1	1	0.0625 microseconds
0	1	0	8	0.5 microseconds
0	1	1	64	4 microseconds
1	0	0	256	16 microseconds
1	0	1	1024	64 microseconds
1	1	0	External clock on T0. Clock on falling edge.	-
1	1	1	External clock on T0. Clock on rising edge	-

Timer 0 Programming
CTC Mode

- **The TOV0 interrupt is triggered whenever TCNT0 reaches the top timer value *V* and rolls back to 0.**
- **We need to decide what *V* is!**

$$V = \frac{T_{cycle}}{res}$$

- **We want our TOV0 interrupt to be triggered every 1 ms, or 1000 microseconds. Using the formula above, the possible values are shown in the table on the next page.**

Timer 0 Programming
CTC Mode

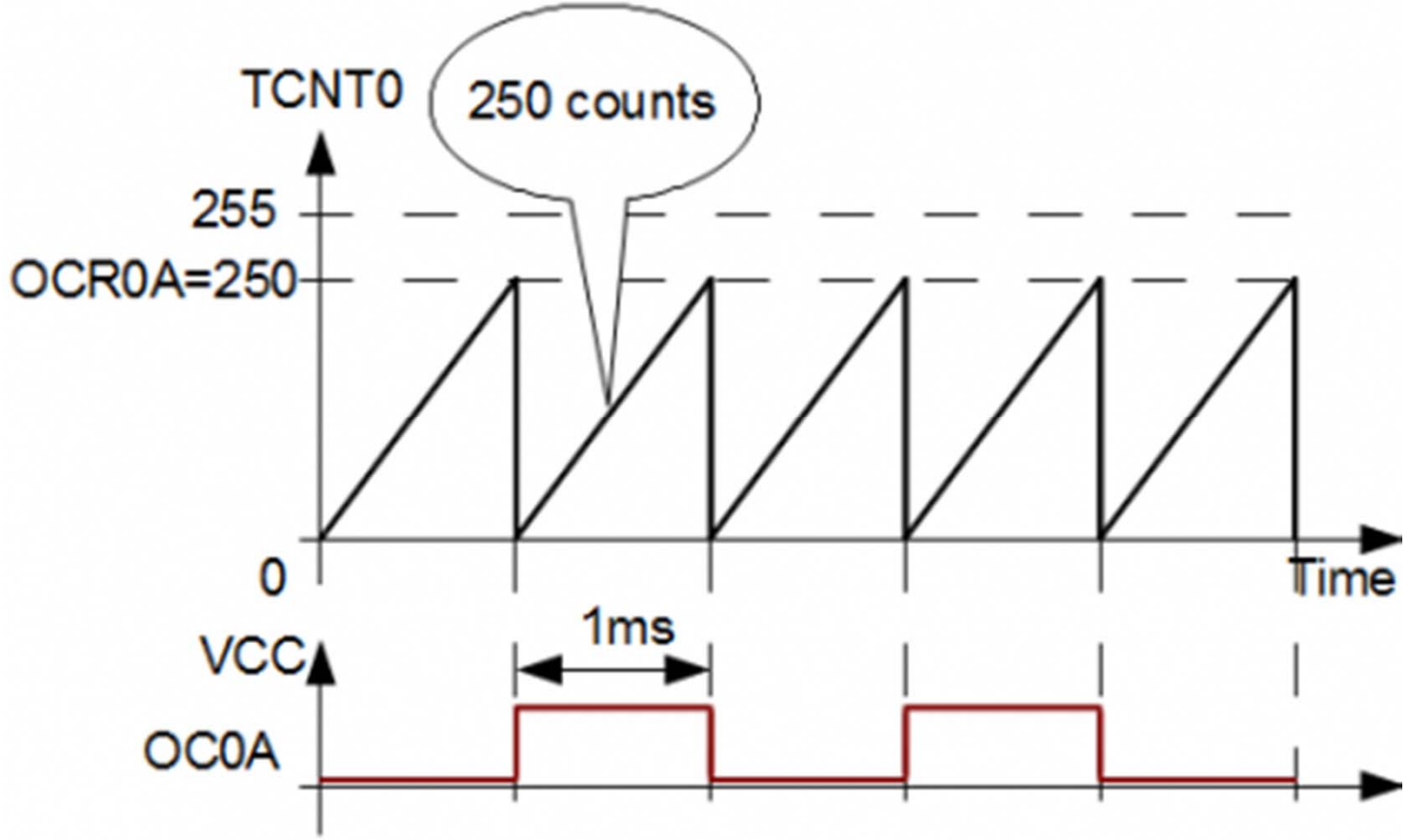
- **The possible *V* are:**

Prescaler	Resolution (16 MHz Clock)	Count *V* (for 1 ms or 1000 microseconds)
1	0.0625 microseconds	16000
8	0.5 microseconds	2000
64	4 microseconds	250
256	16 microseconds	62.5
1024	64 microseconds	15.63

- **It is not possible to load 16000 or 2000 into 8-bit registers. The possible values are 250, 62.5 and 15.63.**
  - We choose the largest value 250 because it gives us the best possible resolution of 4 microseconds.
    - ✓**Better resolution = more accurate timings.**
- **This gives us a prescaler value of 64.**

Timer 0 Programming
CTC Mode

Diagram Credits: Midity. http://www.midity.com/



Timer 0 Programming
CTC Mode

- **Now that we have chosen *P* and *V*, we can begin configuring the timer.**
  - Set the initial timer value in the Timer 0 Control register TCNT0. We use an initial value of 0.

```
TCNT0=0;
```

  - Set the timer value *V* into the Output Compare Register OCR0A. For 250 steps, *V=249*, since we start from 0.

```
OCR0A=249;
```

Timer 0 Programming
CTC Mode

- Set up Timer/Counter Control Register TCCR0A:

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

  - **COM0A1 and COM0A0 control what to do with OC0A each time we hit the top value *V* in OCR0A:**

Table 14-2. Compare Output Mode, non-PWM Mode

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	Toggle OC0A on Compare Match
1	0	Clear OC0A on Compare Match
1	1	Set OC0A on Compare Match

  - **We want to toggle OC0A on match, so we choose CM0A1 and COM0A0 to “01” respectively.**

NUS National University of Singapore
School of Computing

Timer 0 Programming
CTC Mode

- Set up Timer/Counter Control Register TCCR0A:
  - **The WGM00, WGM01 and WGM02 bits control the waveform generation:**

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

  - **We will use CTC mode so WGM01 and WGM00 are “01”. WGM02 is in TCCR0B.**

Timer 0 Programming
CTC Mode

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Taken together, we will program TCCR0A using the following statement:**

```
TCCR0A=0b01000010;
```

- **We need to enable the OCIE0A interrupt (output compare match A) by writing a 1 to bit 1 of the Timer/Counter Interrupt Mask Register TIMSK0:**

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

```
TIMSK0|=0b10;
```

Timer 0 Programming
CTC Mode

- **Finally we will start the timer by writing selecting the prescaler value *P* in TCCR0B, using bits CS02:CS00.**
  - We want a prescaler value of 64 so we use 011.
  - We also want WGM02 to be 0 (see previous slide).

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

```
TCCR0B=0b00000011;
```

  - Note: Setting CS02:CS00 bits to anything other than 011 automatically begins the timer.
  - We must also enable global interrupts using the sei(); function call.

Timer 0 Programming
CTC Mode

- **We will now look at an example program that:**
  - Increments a counter every millisecond.
  - On every 100th increment, it toggles an LED at pin 13 on or off.
    - ✓**Effectively causes the LED to blink on and off every 100 ms.**

Timer 0 Programming
CTC Mode Example 1

```
#include <avr/io.h>
#include <avr/interrupt.h>

int count;
//Initialize Timer0
void InitTimer0(void)
{
        //Set Initial Timer value
        TCNT0=0;
        //Place TOP timer value to Output compare register
        OCR0A=249;

        //Set CTC mode
        //      and make toggle PD6/OC0A pin on compare match
        TCCR0A=0b01000010;
        // Enable interrupts.
        TIMSK0|=0b10;
}
```

Timer 0 Programming
CTC Mode Example 1

```
void StartTimer0(void)
{
        //Set prescaler 64 and start timer
        TCCR0B=0b00000011;
        // Enable global interrupts
        sei();
}

void ledOn()
{
        PORTB|=0b00100000;
}

void ledOff()
{
        PORTB&=0b11011111;
}
```

Timer 0 Programming
CTC Mode Example 1

```
// Toggle the LED
void toggleLED()
{
        static state=1;

        if(state==1)
        {
                ledOn();
                state==0;
        }
        else
        {
                ledOff();
                state=1;
        }
}
```

Timer 0 Programming
CTC Mode Example 1

```
// Set up the ISR for TOV0A
ISR(TIMER0_COMPA_vect)
{
        count++;
        if((count % 100)==0)
                toggleLED();
}
int main(void)
{
        // Set up LED on pin 13 (port B pin 5) for output.
        DDRB|=0b00100000;
        count=0;
        InitTimer0();
        StartTimer0();
    while(1)
    {           // Do nothing.
    }
}
```

NUS National University of Singapore
School of Computing

Timer Programming in General

- **Programming the other times (timer 1 and 2) is exactly the same, but with different register names.**
  - E.g. use TCCR2A, TCCR2B, TCNT2, OCR2A, etc.
- **Timer 1 is a 16-bit timer, giving you better resolution.**

Prescaler	Resolution (16 MHz Clock)	Count *V* (for 1 ms or 1000 microseconds)
1	0.0625 microseconds	16000
8	0.5 microseconds	2000
64	4 microseconds	250
256	16 microseconds	62.5
1024	64 microseconds	15.63

  - We can load a value of 16000 into OCR1AH/OCR1AL to get a 1ms count, allowing use to use a prescaler of 1 and a resolution of 0.0625 microseconds.
    - ✓**This can potentially lead to more accurate timings.**

AVR Programming

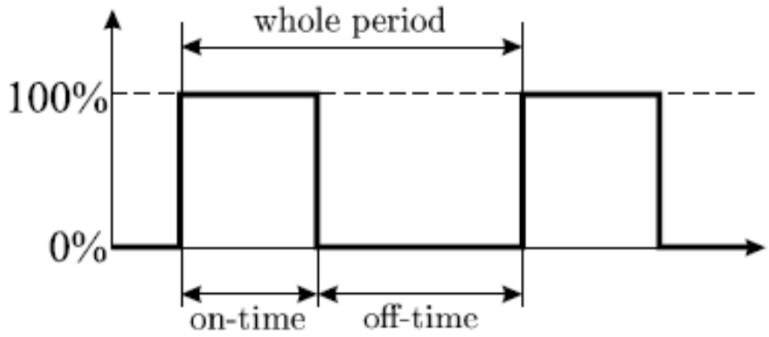
GENERATING ANALOG OUTPUTS

Generating Analog Output

- **Earlier we saw how to read in analog inputs using the ADC:**
  - Allows you to read in thermometers, light sensors, etc.
- **Now we will look at how to generate analog outputs.**
  - Control the intensity of LEDs. No more simple ON or OFF!
  - Control the speed of a motor.
  - Control the heating in a room.
  - Control the turn angle of a servo.

Generating Analog Output
Pulse Width Modulation

- **Actuators are often controlled using "pulse-width modulation"**
  - Digital bits are measured or encoded over a fixed period of time known.
  - Analog values are encoded by the proportion of "1" ("on") time to "0" ("off") time within this fixed time. This proportion is called a "Duty Cycle", usually denoted D.
    - ✓**E.g. if a PWM signal consists of 8-bits, and the analog signals are from 0 to 5v, then 11110000_2 corresponds to 2.5v. This is a 50% duty cycle.**



NUS National University of Singapore
School of Computing

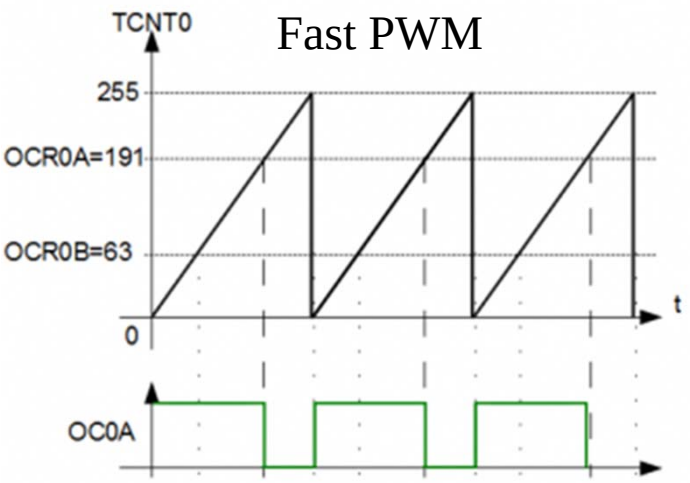
Generating Analog Output
Pulse Width Modulation

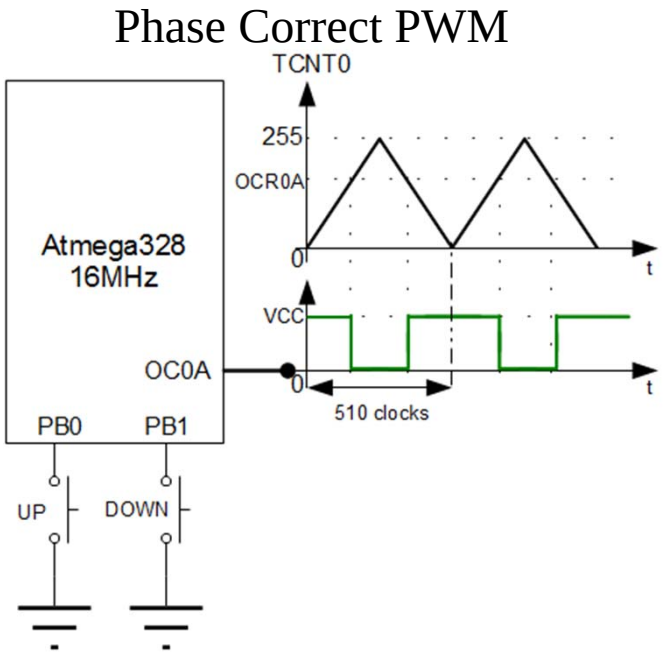
- **PWM is used in Digital to Analog conversion.**
  - If the processor/microcontroller outputs 11111100_2, this corresponds to 3.75v.
  - The actuators react proportionally to the D/A value produced.
- **PWM can also be used in A/D conversion:**
  - Given maximum and minimum analog values of 0v and 5v, an input of 1.25v can be encoded as 11000000_2, if there are 8 bits in a cycle. This is a 25% duty cycle.
- **Given sufficient bandwidth, any analog signal can be represented by PWM.**

Generating Analog Output PWM on the Atmega328

- **PWM is generated using the timers on the Atmega.**
- **There are two PWM modes on the Atmega:**
  - Fast PWM, allowing higher frequencies.
  - Phase Correct PWM, allowing symmetric wave-forms, better resolution at expense of maximum frequency.
    - ✓**More suited for motors.**
- **We will focus only on phase-correct PWM.**

Diagram Credits: Midity. http://www.midity.com





Generating Analog Output
PWM Generation on Timer 0

- **We first need to decide on our PWM frequency. Let's choose 500 Hz as a starting point. We need to choose our prescaler value *P* based on the following formula:**

$$F_{PWM} = \frac{F_{clk}}{510 \times P}$$

- **Assuming our F_{clk} is 16 MHz, substituting F_{pwm}=500 Hz, solving for *P* gives us:**

$$P = 62.745$$

- **The possible prescaler values are 1, 8, 64, 256 and 1024. The closest *P* is therefore 64, giving us F_{pwm}=*490 Hz*.**

NUS National University of Singapore
School of Computing

Generating Analog Output
PWM Generation on Timer 0

- **The duty cycle is decided by the value stored in OCR0A, using the following formula:**

$$D = \frac{OCR0A}{255} \times 100$$

- **The table shows duty cycles of 0%, 25%, 50%, 75% and 100%, corresponding OCR0A values and output voltages. Note that some OCR0A values are decimals and must be rounded.**

Desired D (OCR0A)	Desired V (Vcc=5V)	OCR0A	Actual D	Actual V (Vcc=5V)
0 (0)	0v	0	0	0v
25 (63.75)	1.25v	64	25.098	1.255v
50 (127.5)	2.5v	128	50.196	2.510v
75 (191.25)	3.75v	191	74.902	3.745v
100 (255)	5.00v	255	100	5v

Generating Analog Output
PWM Generation on Timer 0

- **Interrupts can be triggered as usual when TCNT0 rolls over to 0.**
  - Useful for setting new PWM duty cycles at the end of a PWM wave.

```
TIMSK0|=0b10; // Enables OCIE0A IRQ
```

- **To generate phase correct PWM:**
  - Load 0 into the initial count register TCNT0.

```
TCNT0=0;
```

  - Load the desired value into OCR0A. For example, to get a 50% duty cycle, we use:

```
OCR0A=128;
```

Generating Analog Output
PWM Generation on Timer 0

- Set up the PWGM2:0 bits in TCCR0A and TCCR0B.

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on[(1)(2)]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Generating Analog Output
PWM Generation on Timer 0

- **✓There are two phase correct PWM modes. We will choose mode 1.**
- **✓We also need to set COM0A1:0 to 0b10 to clear OC0A when TCNT0 counts up from 0 to OCR0A, and sets 0C0A when TCNT0 counts down from 255 to OCR0A, to give the phase correct waveform.**

Table 14-4. Compare Output Mode, Phase Correct PWM Mode(1)

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	WGM02 = 0: Normal Port Operation, OC0A Disconnected. WGM02 = 1: Toggle OC0A on Compare Match.
1	0	Clear OC0A on Compare Match when up-counting. Set OC0A on Compare Match when down-counting.
1	1	Set OC0A on Compare Match when up-counting. Clear OC0A on Compare Match when down-counting.

- We therefore use:

```
TCCR0A=0b10000001;
```

Generating Analog Output
PWM Generation on Timer 0

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

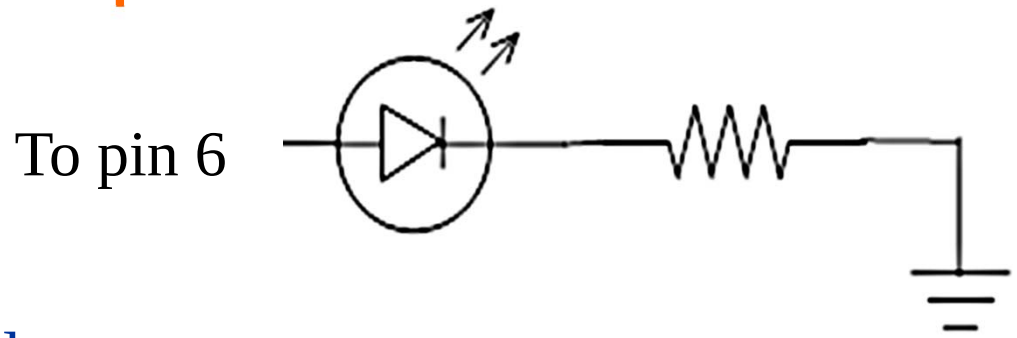
- **Finally we start the PWM signal by writing 0b011 to bits CS02:00 to of TCCR0B select a prescaler of 64. WGM02 should also be set to 0.**

```
TCCR0B=0b00000011;
```

CS02	CS01	CS00	Prescalar P	Resolution (F_{clk}=16 MHz)
0	0	0	Stops the timer	-
0	0	1	1	0.0625 microseconds
0	1	0	8	0.5 microseconds
0	1	1	64	4 microseconds
1	0	0	256	16 microseconds
1	0	1	1024	64 microseconds
1	1	0	External clock on T0. Clock on falling edge.	-
1	1	1	External clock on T0. Clock on rising edge	-

Generating Analog Output Example 1

- **We will now connect an LED to pin OC0A on the Atmega328.**
  - This is shared with pin PD6, which is pin 6 on the Arduino Uno.
    - ✓**You will notice that it reads "~6" on the board. The "~" indicates a PWM output.**



  - Our program will progressively light up the LED in steps of 10% duty cycles. I.e. 0%, 10%, 20%, etc.

Generating Analog Output Example 1

```
#include <avr/interrupt.h>
#include <avr/io.h>

// OCR0A values for 10, 20, .. 100% duty cycles.
int pwm_values[]={0, 26, 51, 77, 102, 128, 153, 179, 204, 230, 255};
int index=0;

void InitPWM()
{
        // Set initial timer value
        TCNT0=0;

        // Set the initial OCR0A values
        OCR0A=0;
```

Generating Analog Output Example 1

```
// Set TCCR0A to clear OC0A when we reach 255,
        // and choose mode 1 Phase correct PWM
        TCCR0A=0b10000001;

        // Enable compare interrupt
        TIMSK0 |= 0b10;
}

// Start PWM signal
void startPWM()
{
        // Set prescaler of 0b011, or 64.
        TCCR0B=0b00000011;

        // Set global interrupts
        sei();
}
```

Generating Analog Output Example 1

```
// ISR
ISR(TIMER0_COMPA_vect)
{
        // Increment the index to go to next duty cycle value.
        index=(index+1)%11;
        OCR0A=pwm_values[index];
}

int main()
{
        InitPWM();
        StartPWM();

        while(1) { // Do nothing // }
}
```

Generating PWM on other timers

- **Generating PWM waveforms is exactly the same on Timers 1 and 2.**
  - Exception: Timer 1 is a 16-bit timer.
    - ✓**Uses two 8-bit OCR1AH and OCR1AL registers instead of a single 8-bit register.**
    - ✓**PWM resolution is much better on Timer 1. 7.63 x 10^{-5} volts vs 0.196 volts on timers 0 and 2.**
    - ✓**Likewise, resolution on timer 1 is 0.002% duty cycle vs 0.392% on timers 0 and 2.**
  - This means that speed/intensity control on Timer 1 is 100x better than Timers 0 and 2.

Summary

- **In this lecture we looked at microcontroller programming on the Atmel Atmega328 AVR.**
  - Used in the Sparkfun Inventor Kits issued to each team.
  - Small, cheap, slow, small memory.
  - But quite representative of how microcontrollers are programmed.
- **We looked at:**
  - Digital input and output using the GPIOs.
  - Analog input using the ADCs.
  - Producing regular interrupts with timers.
  - Analog PWM output using timers.