AMERICAN INTERNATIONAL UNIVERSITY – BANGLADESH(AIUB)
Where leaders are created

Introduction to microcontroller and programming with a microcontroller.

By Tahseen Asma Meem

Department of EEE

Example of a microcontroller : ATmega328.

ATmega-328 is basically an Advanced Virtual RISC (AVR) micro-controller. ATmega-328 has **32KB internal built in memory**.

- ATmega328 supports the data up to **eight (8) bits and 28 Pins** AVR Microcontroller, manufactured by Microchip, follows RISC Architecture and **has a flash type program memory of 32KB**.

- ATmega 328 has **1KB Electrically Erasable Programmable Read Only Memory (EEPROM)**. For a EEPROM, if the electric supply is removed from the micro-controller it can store the data. After providing electric supply it can provide previous results.

- ATmega-328 has **2KB Static Random Access Memory** (SRAM).

- It has **8 Pin for ADC operations**, which all combines to form Port A (PA0 – PA7).

•It also has 3 built in Timers, two of them are 8 Bit timers while the third one is 16-Bit Timer.

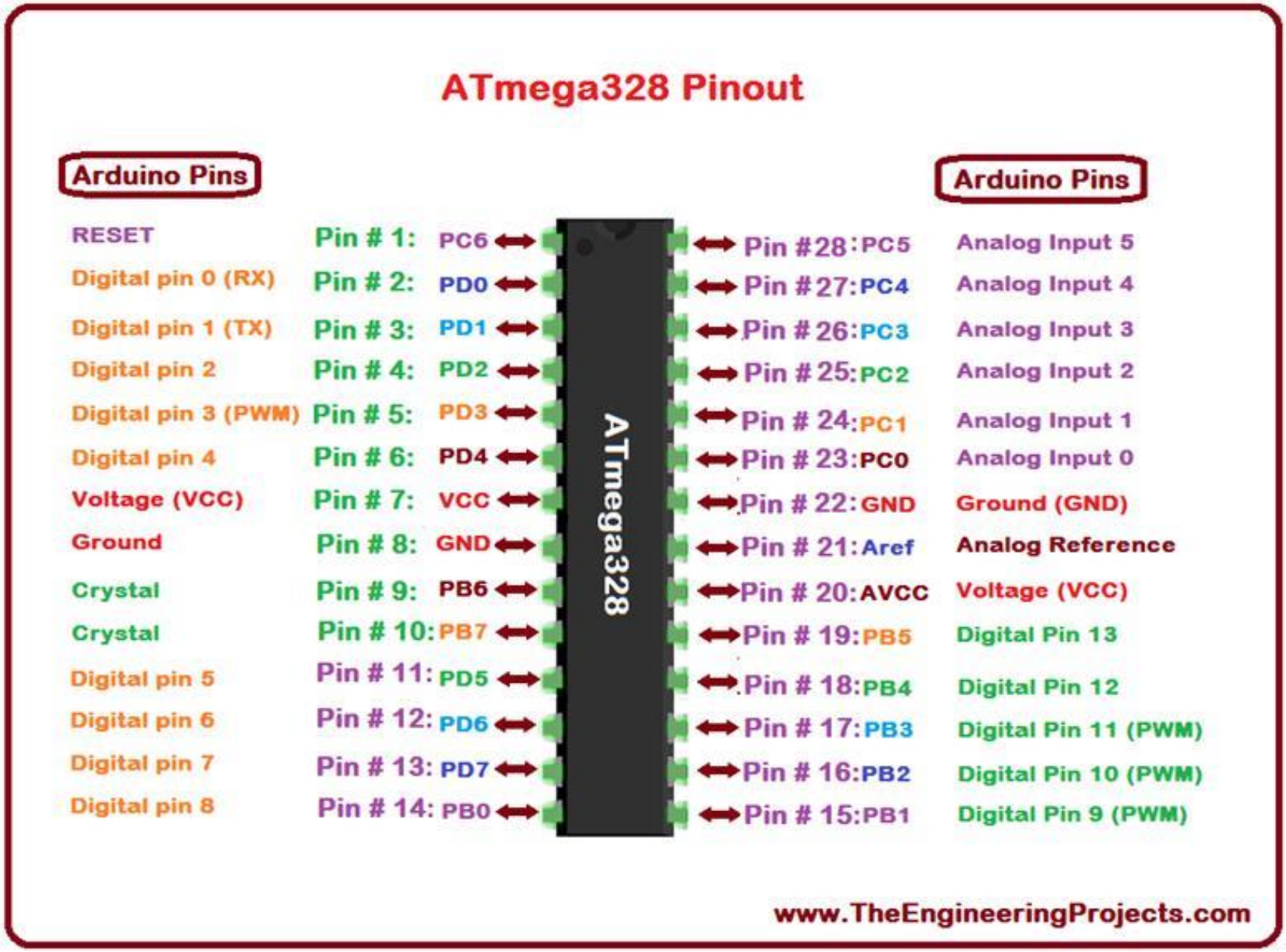
•Arduino UNO is based on **Atmega328** Microcontroller. It's UNO's heart.

•It operates ranging from **3.3V to 5.5V** but normally we use 5V as a standard.

•Its excellent features include the **cost efficiency, low power dissipation, programming lock for security purposes, real timer counter** with separate oscillator.

•It's normally used in Embedded System applications.

•Moreover, ATmega 328 has several different features which makes it the most popular device in today's market. These features consist of advanced RISC architecture, good performance**, 6 PWM pins, programmable Serial USART, programming lock for software security, throughput up to 20 MIPS** etc.


ATmega328 Pinout
Arduino Pins
Arduino Pins
RESET
Digital pin 0 (RX)
Digital pin 1 (TX)
Digital pin 2
Digital pin 3 (PWM)
Digital pin 4
Voltage (VCC)
Ground
Crystal
Crystal
Digital pin 5
Digital pin 6
Digital pin 7
Digital pin 8
Pin # 1: PC6
Pin # 2: PD0
Pin # 3: PD1
Pin # 4: PD2
Pin # 5: PD3
Pin # 6: PD4
Pin # 7: VCC
Pin # 8: GND
Pin # 9: PB6
Pin # 10: PB7
Pin # 11: PD5
Pin # 12: PD6
Pin # 13: PD7
Pin # 14: PB0
ATmega328
Pin #28: PC5
Pin #27: PC4
Pin #26: PC3
Pin #25: PC2
Pin # 24: PC1
Pin # 23: PC0
Pin # 22: GND
Pin # 21: Aref
Pin # 20: AVCC
Pin # 19: PB5
Pin # 18: PB4
Pin # 17: PB3
Pin # 16: PB2
Pin # 15: PB1
Analog Input 5
Analog Input 4
Analog Input 3
Analog Input 2
Analog Input 1
Analog Input 0
Ground (GND)
Analog Reference
Voltage (VCC)
Digital Pin 13
Digital Pin 12
Digital Pin 11 (PWM)
Digital Pin 10 (PWM)
Digital Pin 9 (PWM)
www.TheEngineeringProjects.com

ATmega328 Pins Description

Functions associated with the pins must be known in order to use the device appropriately:

- VCC is a digital voltage supply.
- AVCC is a supply voltage pin for analog to digital converter.
- GND denotes Ground and it has a 0V.
- ATmega-328 pins are divided into different ports which are given in detail below:

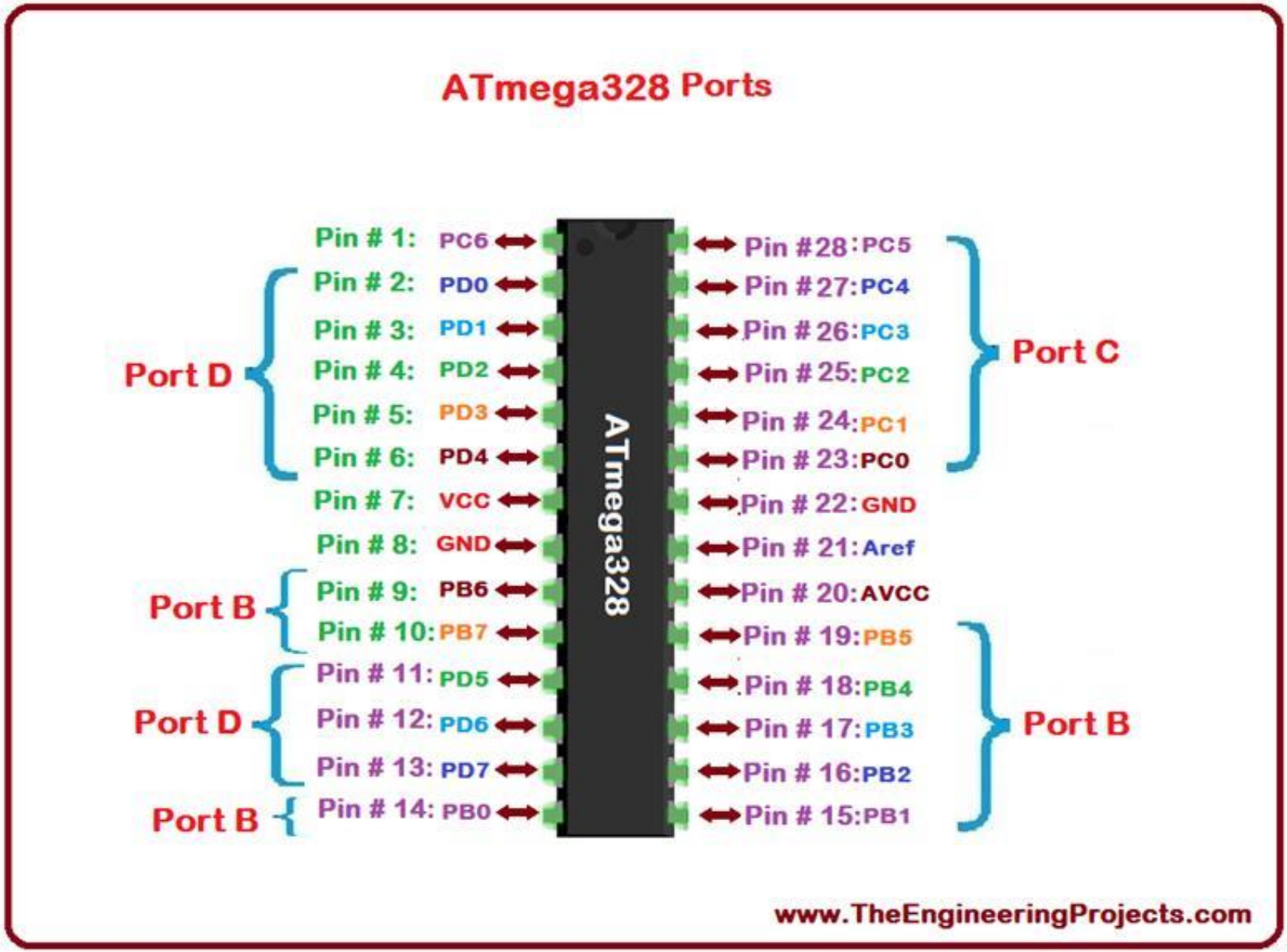
❖ Port A consists of the pins from PA0 to PA7. **These pins serve as analog input to convert from analog to digital converters.** If analog to digital converter is not used, port A acts as **an eight (8) bit bidirectional input/output port.**

❖ Port B consists of the pins from PB0 to PB7. This port is an **8 bit bidirectional port having an internal pull-up resistor.**

❖ Port C consists of the pins from PC0 to PC7. The **output buffers of port C has symmetrical drive characteristics with source capability as well high sink.**

❖ Port D consists of the pins from PD0 to PD7. It is also **an 8 bit input/output port having an internal pull-up resistor.**

❖ AREF is an **analog reference pin for analog to digital converter**.

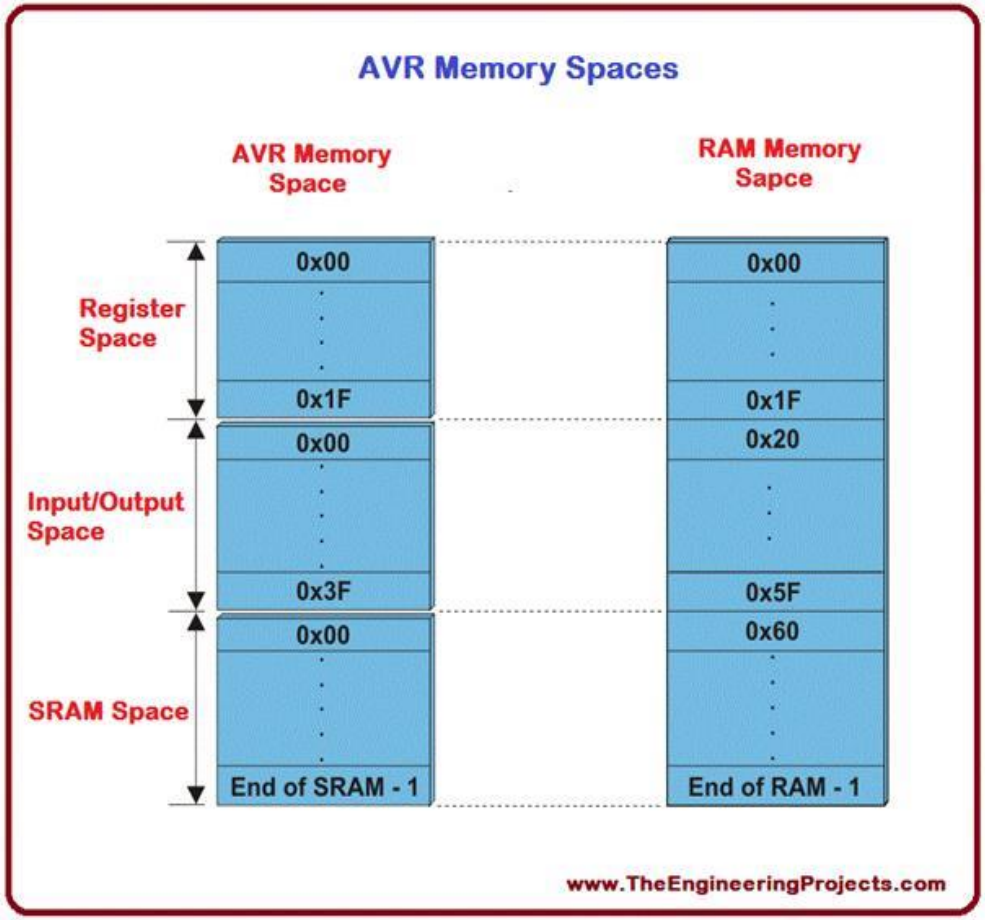

ATmega328 Ports
Pin # 1: PC6
Pin # 2: PD0
Pin # 3: PD1
Pin # 4: PD2
Pin # 5: PD3
Pin # 6: PD4
Pin # 7: VCC
Pin # 8: GND
Pin # 9: PB6
Pin # 10: PB7
Pin # 11: PD5
Pin # 12: PD6
Pin # 13: PD7
Pin # 14: PB0
Pin # 28: PC5
Pin # 27: PC4
Pin # 26: PC3
Pin # 25: PC2
Pin # 24: PC1
Pin # 23: PC0
Pin # 22: GND
Pin # 21: Aref
Pin # 20: AVCC
Pin # 19: PB5
Pin # 18: PB4
Pin # 17: PB3
Pin # 16: PB2
Pin # 15: PB1
ATmega328
Port D
Port B
Port D
Port B
Port C
Port B
www.TheEngineeringProjects.com

ATmega328 Memory

- ATmega 328 has **three types of memories** e.g. EEPROM, SRAM etc.
- The capacity of each memory is explained in detail below:
- Flash Memory has **32KB capacity. It has an address of 15 bits. It is a Programmable Read Only Memory (ROM) and** non volatile memory.

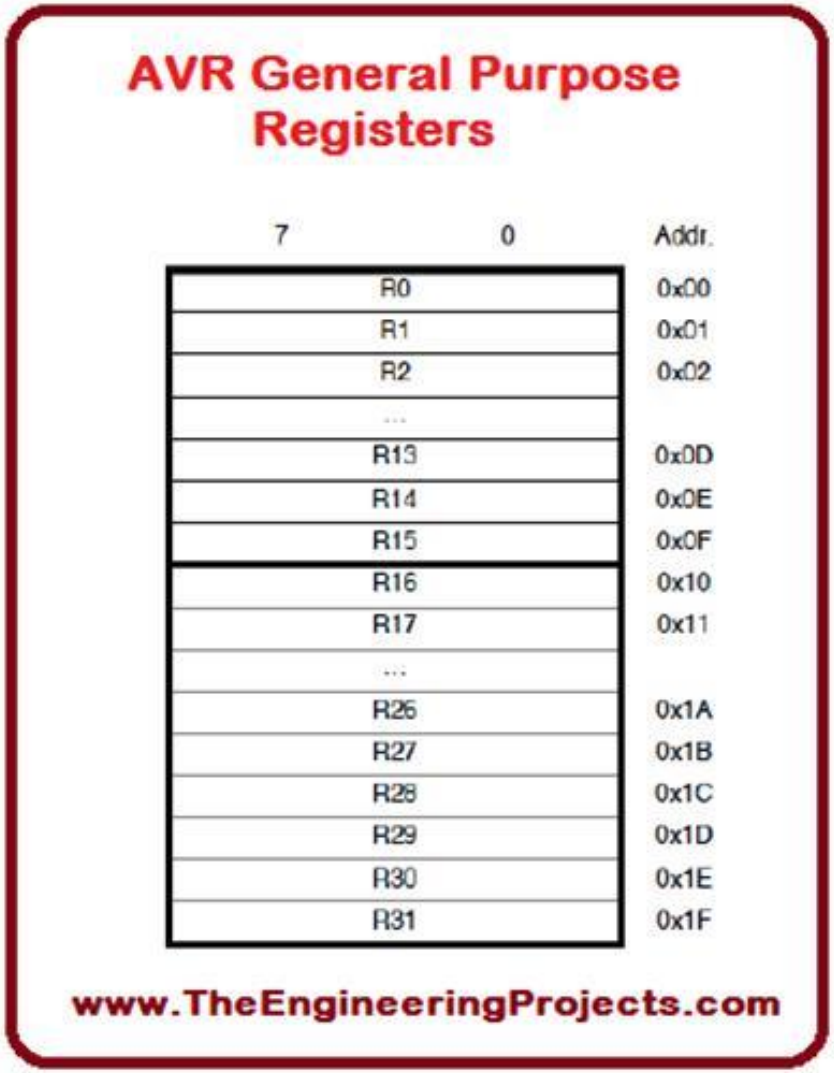
- SRAM stands for **Static Random Access Memory. It is a volatile memory i.e. data will be removed after removing the power supply.**

- EEPROM stands for Electrically Erasable Programmable Read Only Memory. **It has a long term data.**



ATmega328 Registers

- ATmega-328 has thirty two **(32) General Purpose (GP) registers.**
- These all of the registers are the **part of Static Random Access Memory (SRAM).**

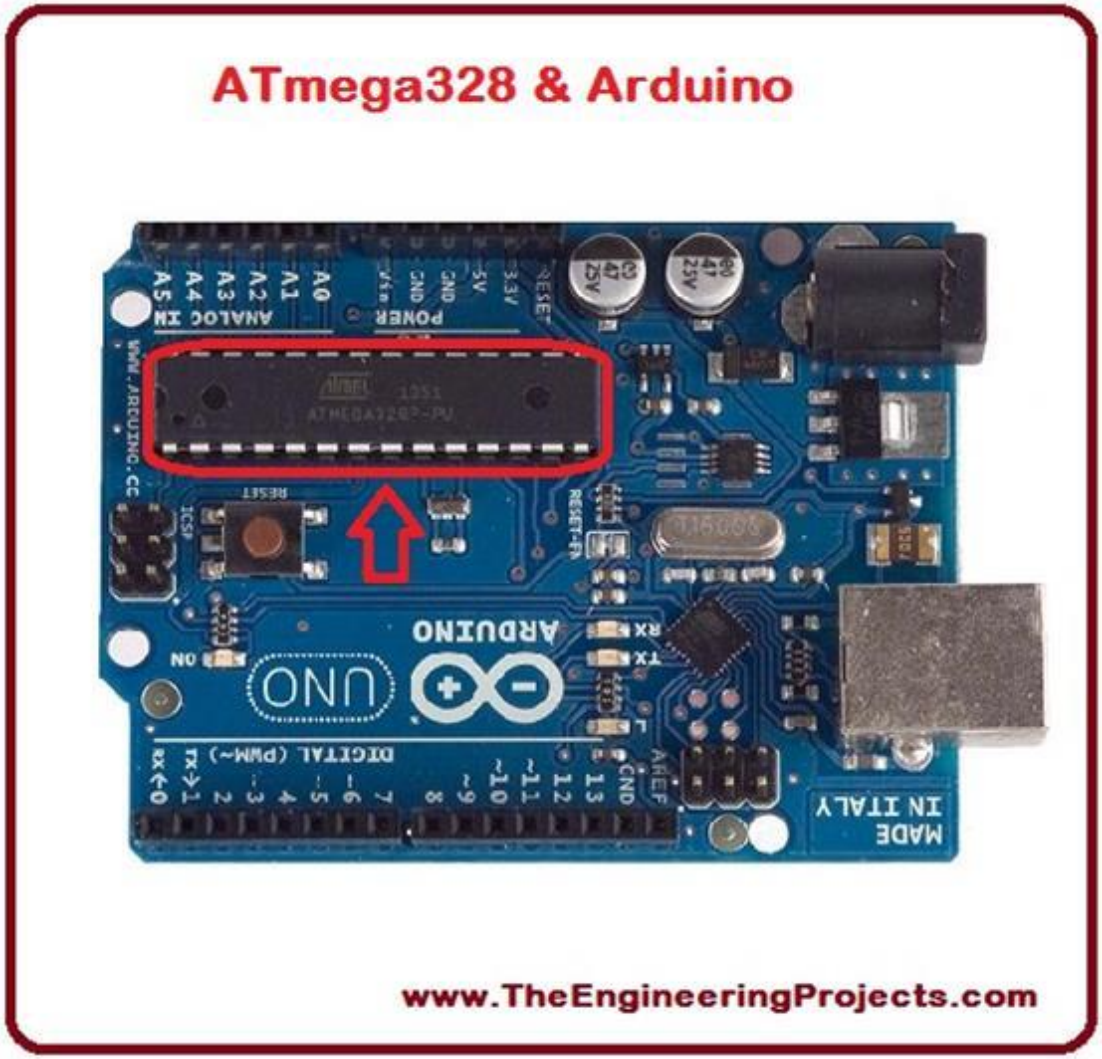


.ATmega328 and Arduino

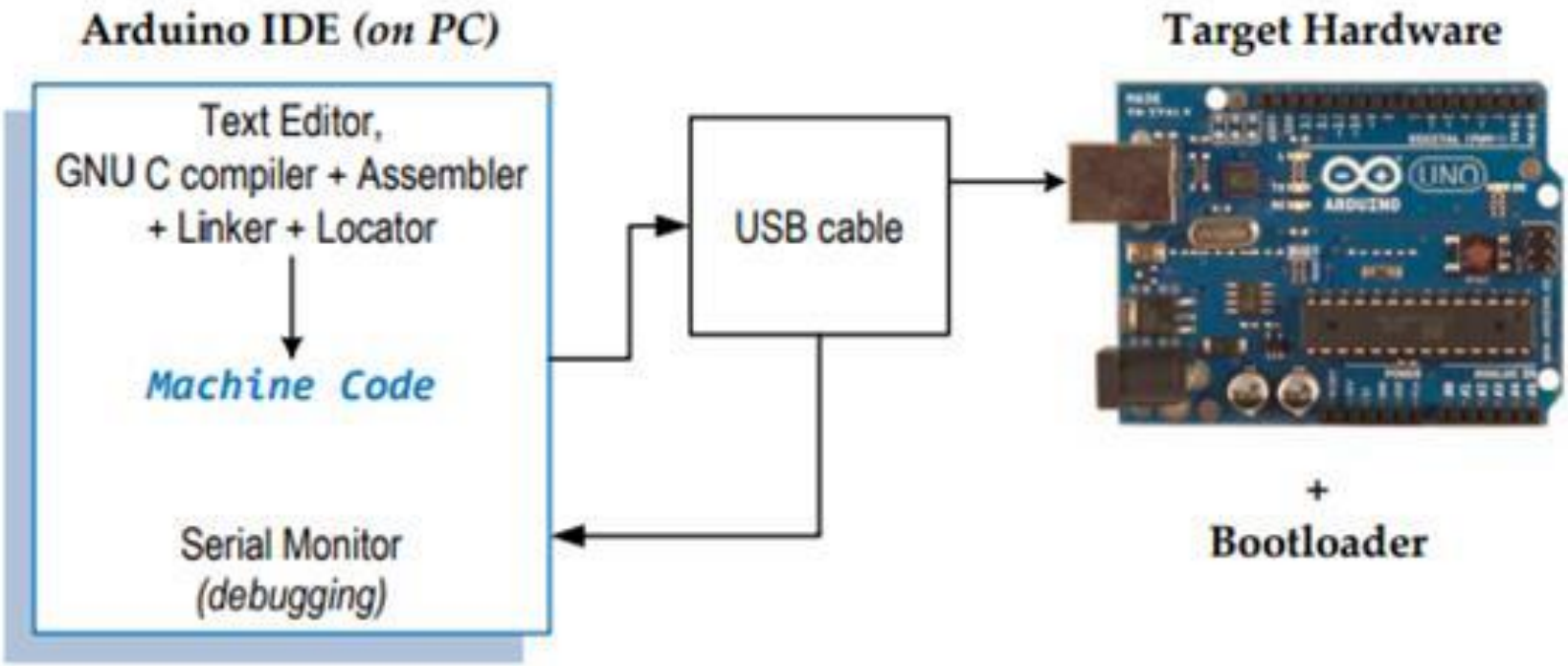
ATmega-328 is the most important micro-controller that is used while designing.

ATmega 328 is the most important part of Arduino.

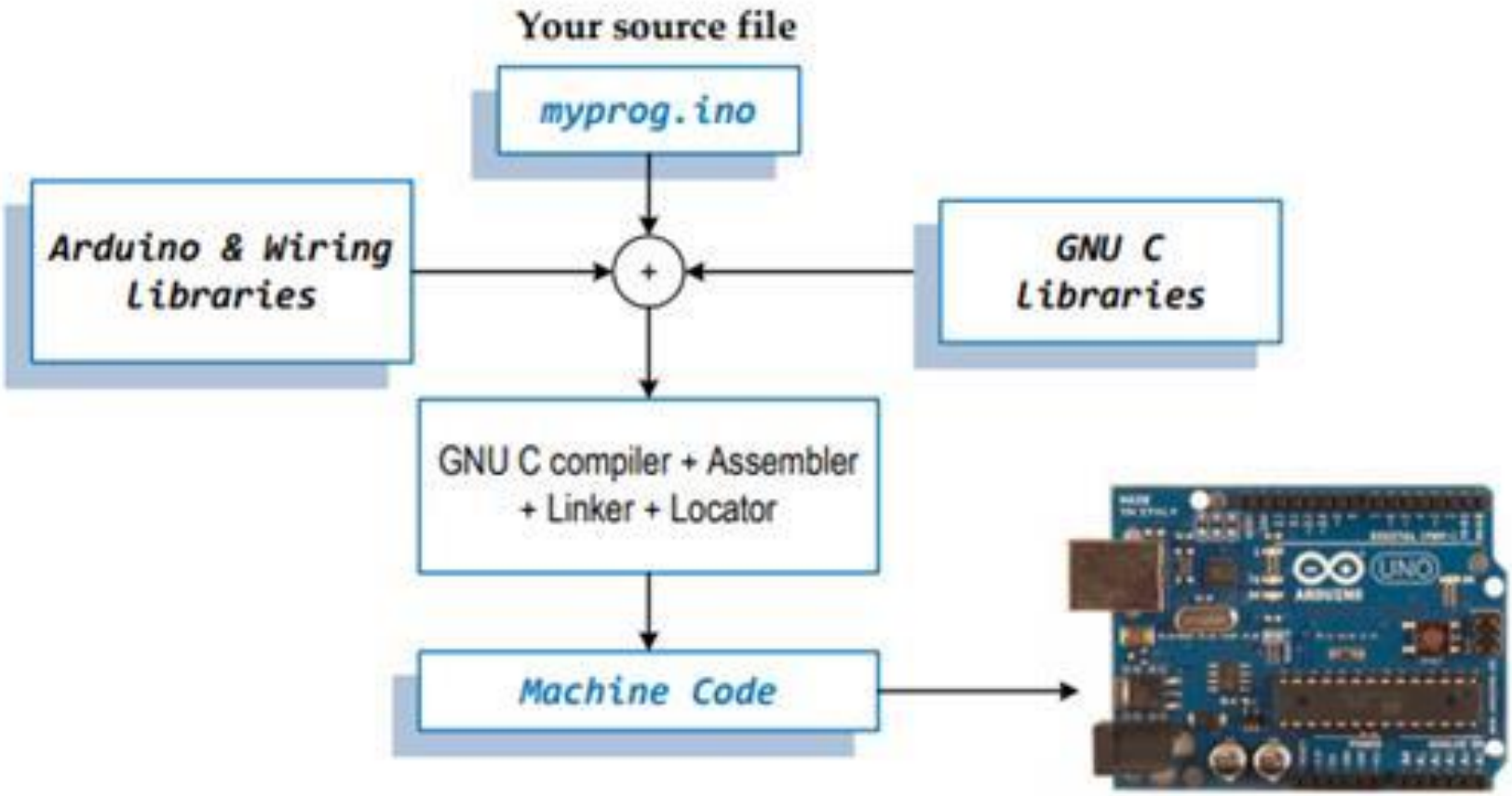
The program is uploaded on the AVR micro-controller attached on Arduino.



- Software development tools for Arduino – Integrated Development Environment (IDE)



- Source codes to machine code translation using Arduino IDE



- Arduino IDE


Compile
Open
Serial Monitor
Upload
sketch_jun15a | Arduino 1.0.5
File Edit Sketch Tools Help
Save
sketch_jun15a
Area for Writing Program
(source code editor)
Area for Error Messages
1
Arduino Uno on COM6

- Arduino language is based on C/C++ language – we will review the C language.
- The basic stucture of an Arduino program:

```
/*
  Turns on LED for 1 sec, then off for 1 sec, repeatedly.
 */

void setup() {
  pinMode(13, OUTPUT);      // init digital pin 13 as an output.
}

void loop() {
  digitalWrite(13, HIGH);   // turn the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off
  delay(1000);              // wait for a second
}
```

comments

function name

function start

function body

function end

- Each Arduino program must have at least 2 functions:
  - `setup()` – is the 1st thing run in the Arduino prog. This is where things that need to be *initialized* should be placed. `setup()` run **once** & only **once**.
  - `loop()` – contains anything that needs to run *repeatedly* in the prog. Any instructions in `loop()` will be run repeatedly until the prog is stopped.
- A C program consists of :
  - **functions** to be executed
  - **variables** declaration, definition or allocation
- C **functions**
  - similar to ASM *subroutines* or Pascal *procedures*
  - a function contains *statements* that specify the computing operations to be done & *variables* declaration for storing data. Function body are enclosed in braces { }
  - `main` is a special function – program execution begins at `main`.

- C functions
  - Where is the `main()` function in Arduino progs?
  - In Arduino, `main` is defined internally ⇒ you do not need to write the `main`

```
int main(void)
{
    :
    setup();

    for (;;) {
        loop();
        :
    }
    return 0;
}
```

- Comments
  - any characters between /* & */ are ignored by compiler
  - anything after // until end of line
  - blank lines are ignored by compiler
- Preprocessor instructions
  - use to include the contents of other files with `#include<..>` or `#include "..."`
  - use to define constants & macros with `#define ...`

C variables & data types

- variables are used to store data
- all variables must be **declared** before they are used. Declaration consists of a data *type* & the symbolic *name* for variable:

```
int  ADCreading; /*var ADCreading store integer value */
float ctrsignal; /* var ctrsignal store real value */
```

- **signed int** data types for variables

Data Types	Size (bytes)	Ranges
char	1	-128 – +127
int	2	-32768 – +32767
short	2	-32768 – +32767
long	4	-2147483648 – +2147483647

C variables & data types

- **unsigned int** data types for variables

Data Types	Size (bytes)	Ranges
unsigned char	1	0 – 255
unsigned int	2	0 – 65535
unsigned long	4	0 – 4294967295

- **real** data types for variables

Data Types	Size (bytes)	Ranges
float	4	$\pm 1.18 \times 10^{-38} - \pm 3.4 \times 10^{38}$
double	8	$\pm 9.46 \times 10^{-308} - \pm 1.79 \times 10^{308}$

- Constant numbers in C

Constant Types	Examples
Decimal number	`65`
Long Decimal number	`65L`
Hexadecimal number	`0x41` or `0X41`
Octal number	`0101`
Character	`'A'`
Real number	`65.0, 65E0, 650E-1`

- Character constant in C (must be enclosed in ' '):

`\a`	alert (bell) character	`\\`	backslash
`\b`	backspace	`\?`	question mark
`\f`	formfeed	`\'`	single quote
`\n`	newline	`\"`	double quote
`\r`	carriage return	`\`*ooo*	octal number
`\t`	horizontal tab	`\x`*hh*	hexadecimal number
`\v`	vertical tab		

- **Identifier** – symbolic names given to variables, functions, constants or labels.
- Valid **identifier** must:
  - start with a letter or underscore (_)
  - consists of letters, numbers or _ only
  - not one of these C keywords:
- C identifier is **case** sensitive ! `ADCreading` ≠ `adcreading`

```
auto        extern      sizeof
break       float       static
case        for         struct
char        goto        switch
const       if          typedef
continue    int         union
default     long        unsigned
do          register    void
double      return      volatile
else        short       while
enum        signed
```

- C **statements**
  - specify actions to be performed, such as assignment operation or function calls.
  - are free-form i.e. can start & finish at any column
  - must be terminated with semicolon (;)
  - are executed in sequence
  - block of statements are enclosed in { & } & not terminated by a semicolon
  - a statement with only a semicolon is an *empty statement* & does not peform any operation

```
for (i = 0; i < 100; i++)
        ;                    /* empty statement */
```

4.6 A Brief Review of C language

Arithmetic operators

Operator	Meaning	Example	Result
*	Multiplication	x * y	The product of x and y.
/	Division	x / y	The quotient of x by y.
%	Modulo division	x % y	The remainder of the division x / y.
+	Addition	x + y	The sum of x and y.
-	Subtraction	x - y	The difference of x and y.
+ (unary)	Positive sign	+x	The value of x.
- (unary)	Negative sign	-x	The arithmetic negation of x.
++	Increment	++x x++	x is incremented (x=x+1). The prefixed operator (++x) increments the operand *before* it is evaluated; the postfixed operator (x++) increments the operand *after* it is evaluated.
--	Decrement	--x x--	x is decremented (x=x-1). The prefixed operator (--x) decrements the operand *before* it is evaluated; the postfixed operator (x--) decrements the operand *after* it is evaluated.

4.6 A Brief Review of C language

Relational operators

Operator	Meaning	Example	Result: 1 (true) or 0 (false)
`<`	less than	`x < y`	`1` if `x` is less than y
`<=`	less than or equal to	`x <= y`	`1` if `x` is less than or equal to `y`
`>`	greater than	`x > y`	`1` if `x` is greater than `y`
`>=`	greater than or equal to	`x >= y`	`1` if `x` is greater than or equal to `y`
`==`	equal to	`x == y`	`1` if `x` is equal to `y`
`!=`	not equal to	`x != y`	`1` if `x` is not equal to `y`. In all other cases, the expression yields `0`.

Assignment operators

Operator	Meaning	Example	Result
`=`	Simple assignment	`x = y`	Assign the value of `y` to `x`
`op=`	Compound assignment	`x += y`	`x op= y` is equivalent to `x = x op (y)` (where `op` is a binary arithmetic or binary bitwise operator)

4.6 A Brief Review of C language

Bitwise operators

Operator	Meaning	Example	Result (for each bit position)
&	bitwise AND	x & y	1, if 1 in both x and y
|	bitwise OR	x | y	1, if 1 in either x or y, or both
^	bitwise exclusive OR	x ^ y	1, if 1 in either x or y, but not both
~	bitwise NOT	~x	1, if 0 in x
<<	shift left	x << y	Each bit in x is shifted y positions to the left
>>	shift right	x >> y	Each bit in x is shifted y positions to the right

Logical operators

Operator	Meaning	Example	Result: 1 (true) or 0 (false)
&&	logical AND	x && y	1 if both x and y are not equal to 0
||	logical OR	x || y	1 if either or both of x and y is not equal to 0
!	logical NOT	!x	1 if x equals 0. In all other cases, the expression yields 0.

4.6 A Brief Review of C language

Memory accessing operators

Operator	Meaning	Example	Result
`&`	Address of	`&x`	A constant pointer to `x`
`*`	Indirection	`*p`	The object (or function) pointed to by `p`
`[ ]`	Array element	`x[i]`	`*(x+i)`, the element with index `i` in the array `x`
`.`	Member of a structure or union	`s.x`	The member named `x` in the structure or union `s`
`->`	Member of a structure or union	`p->x`	The member named `x` in the structure or union pointed to by `p`

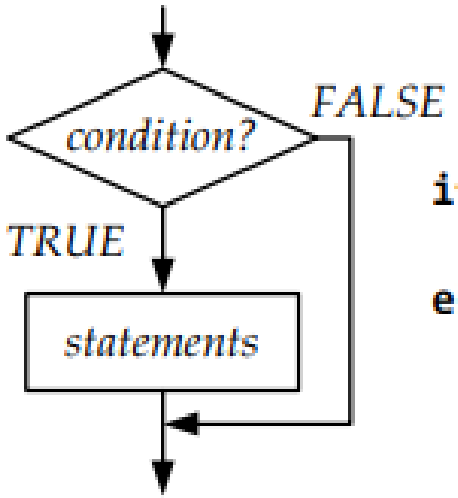
Other operators

Operator	Meaning	Example	Result
`()`	Function call	`pow(x,y)`	Execute the function with the arguments `x` and `y`
`(type)`	Cast	`(long)x`	The value of `x` with the specified type
`sizeof`	Size in bytes	`sizeof(x)`	The number of bytes occupied by `x`
`?:`	Conditional evaluation	`x?y:z`	If `x` is not equal to `0`, then `y`, otherwise `z`
`,`	Sequence operator	`x,y`	Evaluate `x` first, then `y`

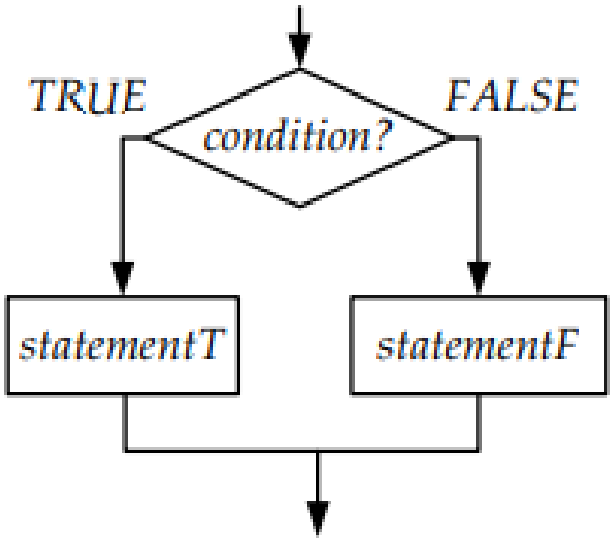
4.6 A Brief Review of C language

- **Program control structures** – for decision making & controlling program flow such as doing selections, iterations & loops.
- **if . . . else** creates conditional jump

```
if (condition)
   statements;
```




```
if (condition)
   statementT;
else
   statementF;
```



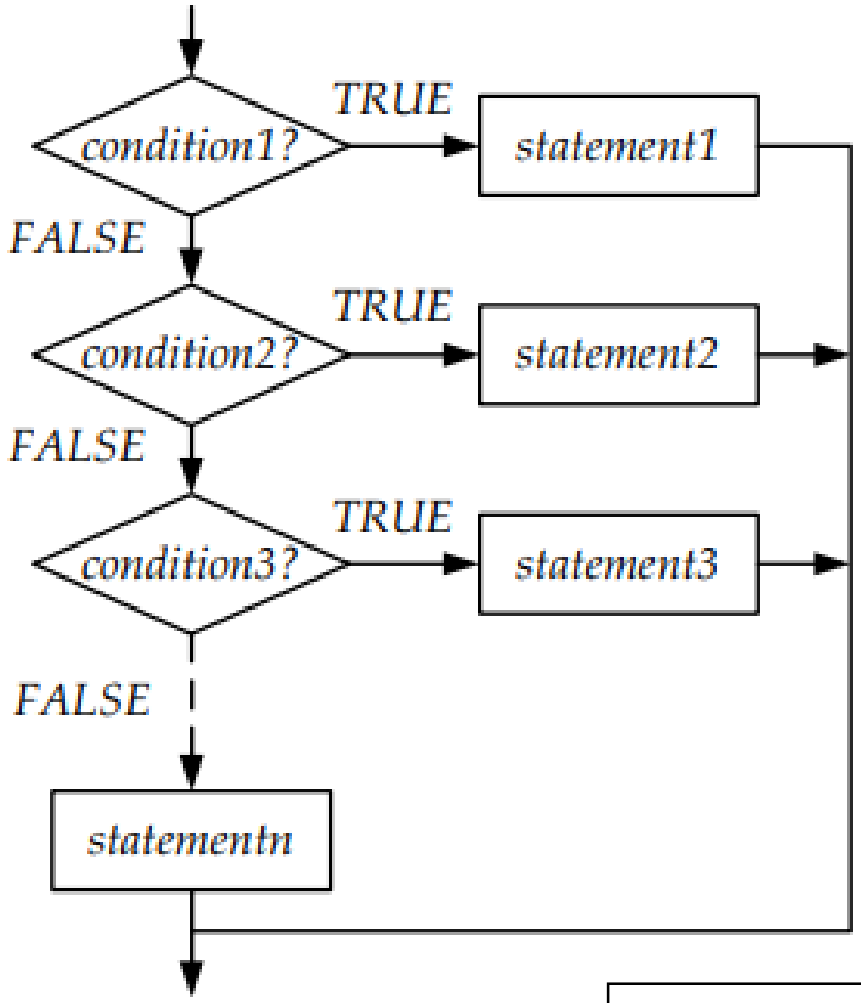
- `condition` = 0 ⇒ FALSE, `condition` ≠ 0 ⇒ TRUE

4.6 A Brief Review of C language

- **if** . . . **else if** . . . **else** for multiple alternatives conditional jumps

```
if (condition1)
    statement1;
else if (condition2)
    statement2;
else if (condition3)
    statement3;
   :
else
    statementn;
```

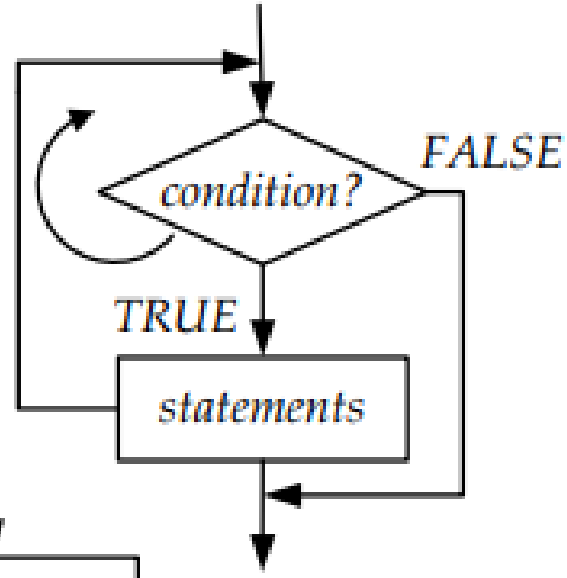
```
switch (condition)
{
   case condition1:
      statement1; break;
   case condition2:
      statement2; break;
   case condition3:
      statement3; break;
   default:
      statementn;
}
```



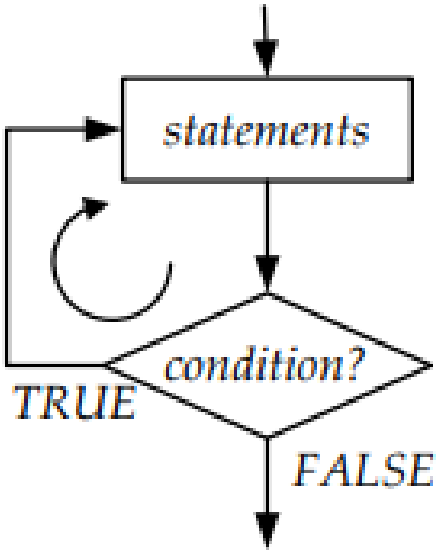
4.6 A Brief Review of C language

- Iteration & loops with **while** & **do** . . . **while**:

```
while (condition)
{
      statement1;
      statement2;
            :
      statementn;
}
```

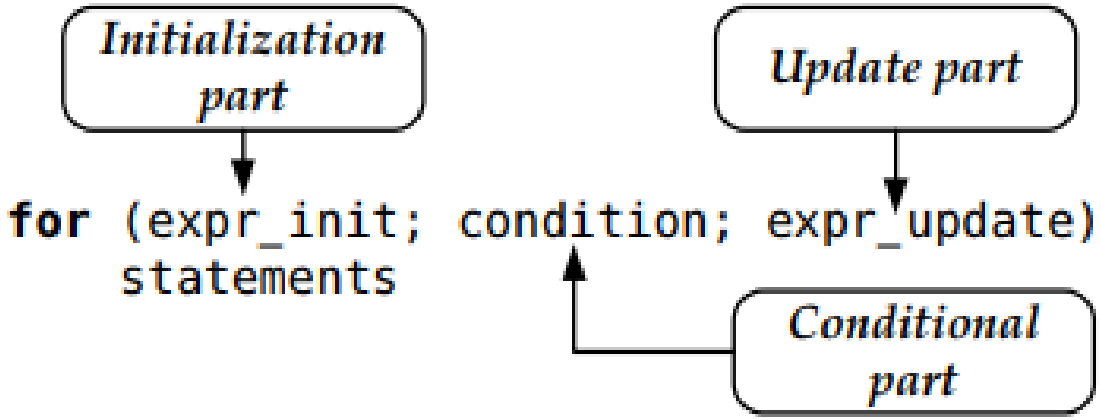



```
do
{
      statement1;
      statement2;
            :
      statementn;
} while (condition)
```



4.6 A Brief Review of C language

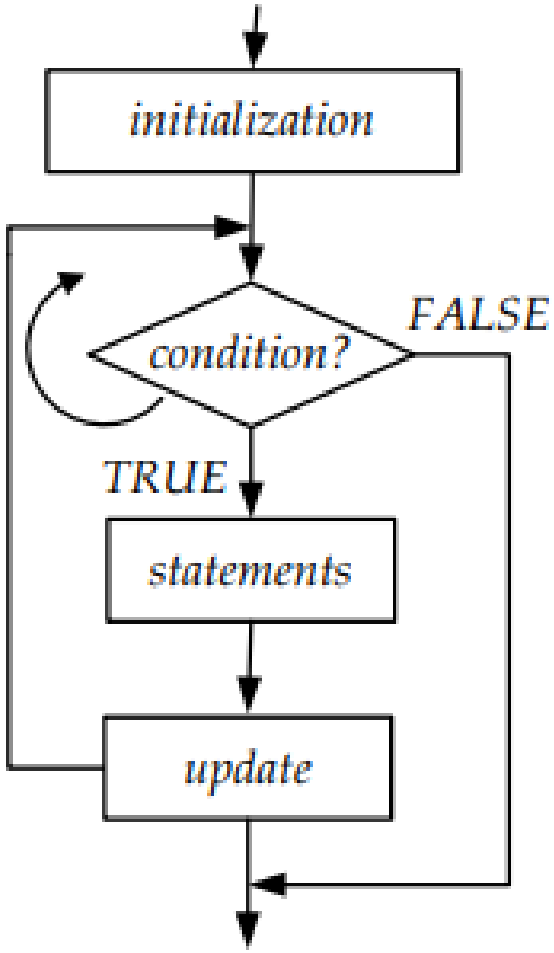
- Syntax of **for** loop:



- Actions of **for** keyword:
  - *Initialization part* is executed **once** at the start of the loop
  - *Conditional part* is tested, if TRUE the statements are executed
  - *Update part* is executed at the end of each loop iteration

4.6 A Brief Review of C language

- **for** loop:




```
for (expr_init; condition; expr_update)
    statements
```

||

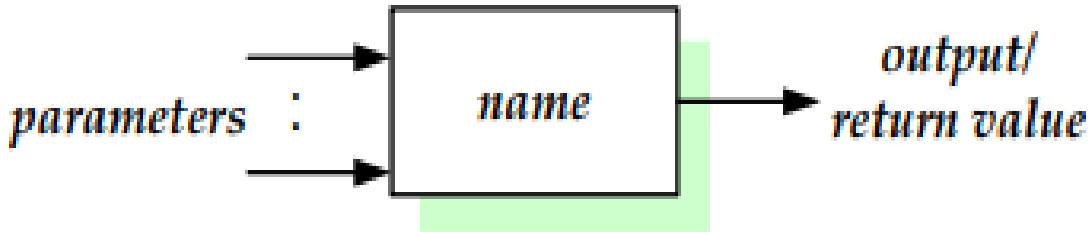
```
expr_init;
while (condition)
{
    statements
    expr_update;
}
```

C functions

- General form of a function

```
output_type  name (par_type parameters)
{
        statements

        return(out_value);
}
```



- output_type – the type of the function's return value/ouput (int, char, float etc.). If no output keyword void is used.
- name is the name of the function. Follow the rules for *identifier* for writing name.

- C functions
  - `par_type parameters` – the declaration of **variables** that are passed to the function to do the work. If more than 1 parameters, separate them with `,`.
  - If the `output_type` is not `void`, the last statement in the function is `return(out_value)`.
  - Example:.

```
/* calc the area of a rectangle */
float calc_area_rect (float width, float length)
{
    float area;

    area = width * length;
    return(area);
}
void main(void)
{
    float area;
    area = calc_area_rect(2.5, 6.0);  /* call function to do the work */
}
```

- **Array** in C.
  - **Arrays** are used to store large numbers of data of the same type & only 1 variable is used to reference them.
  - The format for declaring an array:

    ```
    data_type name [array_size]
    ```

  - `data_type` – `char`, `int`, `float`, etc.
  - `name` – name of array that will be used to access the array. Follow the rules of *identifier*.
  - `array_size` – the number of item the array can store.
  - array can be initialized when declaring. In this case `array_size` can be ommited.
  - each item/element of an array is referenced using the name & index: `name[0]` is the 1st item & `name[array_size-1]` is the last item.

- **Array** in C.
  - **Exercise:** *what are the values of* `sum`, `average` *& the arrays* `length` *&* `len` *elements after this program is executed ? How many bytes of memory are used to store the array* `len`*?*

```
void main(void)
{
    int i, sum, average;
    int length[] = {7, 6, 4, 7, 2, 10};
    int len[6];

    sum = 0;
    for (i = 0; i < 6; i++)
    {
        sum += length[i];
        len[5 - i] = length[i];
    }
    average = (length[5] + length[3])/2;
    length[3] += 3;
    length[2] = length[2] + average;
}
```

- **Pointers** in C.
  - A pointer is a variable which can store an *address* of a variable of function.
  - A pointer refers to a location in memory, & its type indicates how the data at this location is to be interpreted.
  - The unary operator **&** gives the *address* of a variable.
  - The unary operator * is the *indirection / dereferencing* operator; it accesses the variable the pointer points to.
  - 2 *arithmetic operations* can be performed on pointers:
    - An integer can be added to or subtracted from a pointer.
    - One pointer can be subtracted from another pointer of the same type.

- **Pointers** in C.
  - **Example:**

```
char c;
char *cptr;      /* pointer to a char    */
int x;
int *iptr;       /* pointer to an integer */

cptr = &c;       /* cptr = 0x1000        */
*cptr = 0x45;    /* c = 0x45             */
++cptr;          /* cptr = 0x1001        */

iptr = &x;       /* iptr = 0x1002        */
*iptr = c;       /* x = 0x0045           */
++iptr;          /* iptr = 0x1004        */
```

RAM

Address	*Contents*	*Variables*
0x1000	0x90	c
0x1001	0x78	
0x1002	0x56	x
0x1003	0x34	

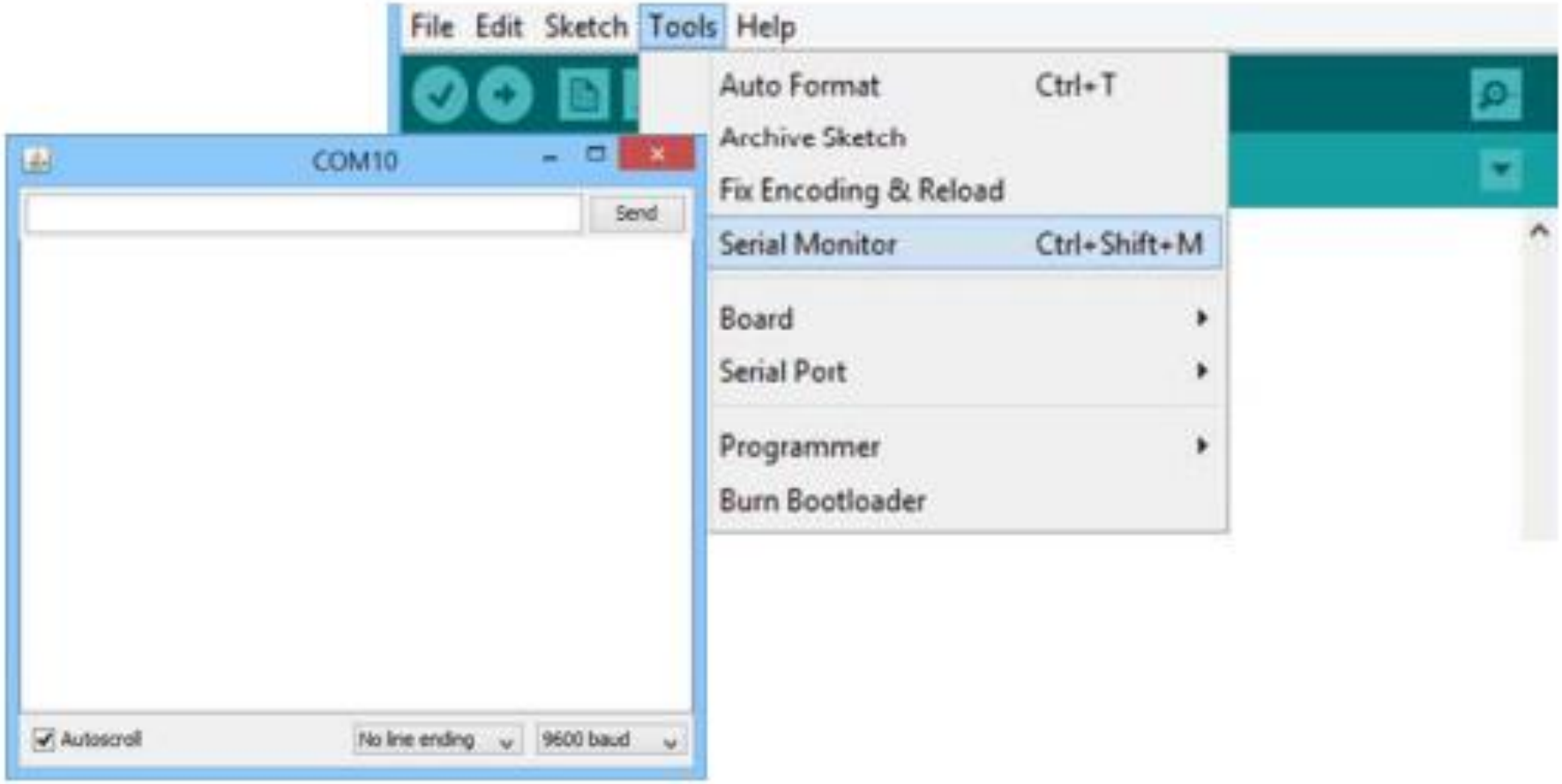
- **Pointers** in C.
  - **Example:** Code fragment to copy 5 bytes from RAM starting at address 0xC000 & store in an array.

```
#define RAM_start (char *) 0xC000

char * ptr2ram;                  /* declare a ptr to RAM       */
char ramcopy[5];                 /* array to store copy of RAM */
int i;

ptr2ram = RAM_start;             /* ptr2ram points to 0xC000   */
for (i=0; i< 5; i++)             /* do it 5 times              */
{
    ramcopy[i] = *ptr2ram;       /* copy a byte to array       */
    ++ptr2ram;                   /* update ptr to next location*/
}
```

- **Serial Monitor** can be used to display values & messages for debugging Arduino programs.



- **Serial Monitor** – commands for using it:
  - `Serial.begin(9600)` – to enable input/output to serial monitor. Must be written in `setup()`
  - `Serial.println(val)` – to display `val` value to serial monitor with *newline* added.
  - `Serial.print(val)` – as above but without *newline*.
  - `Serial.print("Error")` – display message "Error" without *newline*.
- Make sure USB cable is connected to PC & the same baud rate value (9600) is selected in serial monitor!

- *Example:* displaying values & messages for debugging Arduino programs.

```
int debugMode = true;          // set to false to turn off
int result;                    // variable to be monitored

void setup() {
  Serial.begin(9600);
}

void loop() {
  :
  if (debugMode) {
    Serial.println("In Debug"); // these messg will be
    Serial.print("Result = ");  // printed only
    Serial.println(result);     // in debug mode
  }
  :
}
```

References:

- http://www.circuitstoday.com/arduino-uno-pinout-schematics