5.1 Organization of Timer/Counter-1 of ATmega328

(1) In Chapter-1, we began learning the Arduino Kit by blinking a LED (L). The 1-sec time-gap between ON and OFF conditions of the LED was given by calling a 'Time Delay Subroutine, *delay (1000)*' of the Arduino IDE Interface. Assume that the *delay (1000)* subroutine is not available, then how are we going to insert this time-gap? In this chapter, we will see that this time-gap and many other timing functions can be generated using the TC1 (Timer/Counter-1) resource of the ATmega328 MCU.

(2) The Conceptual View of the TC1 Resource is depicted below in Fig-5.1.

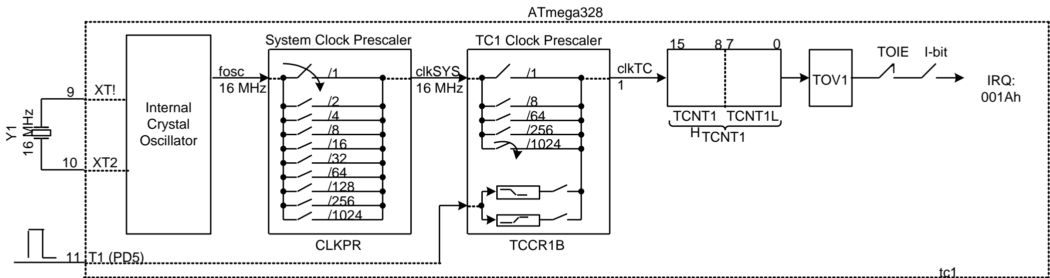


Figure-1: *Conceptual view for the TC1 resources of the ATmega328 Microcontroller*

(3) The TC1 (TCNT1) is a 16-bit register, and it is composed of two 8-bit registers. The lower 8-bit register is known as TCNT1L, and the upper 8-bit register is known as TCNT1H.

(4) The clocking pulse for the TCNT1 is named as clkTC1. It has two sources: the internal oscillator and the external pulses.

(5) When using the internal oscillator, the clkTC1 can be as low as 152.587890625 Hz, and it can be as high as 16 MHz. In Arduino UNO, the 'System Clock Prescaler' is programmed to pass 16 MHz; however, the division factor can be changed through program codes. The 'TC1 Clock Prescaler' division factor can also be changed using program codes.

(6) When the TC1 is configured to receive clocking (driving) pulses form the internal oscillator, we say that the TC1 is operating as T1 (Timer-1).

(7) When the TC1 is configured to receive clocking (driving) pulses form the external source via Pin-11 (T1), we say that the TC1 is operating as C1 (Counter-1). In the C1 mode of operation, there is no division factor; but, an option is available as to which edge (rising/falling) of the incoming pulse is to sense.

(8) The configuration as shown in Fig-5.1 depicts the normal mode of operation of TCNT1. In this mode, the TC1 works as an up-counter; it begins counting from all 0s (0000h) and reaches at the full count of all 1s (FFFFh). At the arrival of the next pulse, the TC1 rolls-over from all 1s to all 0s. This event is known as roll-over event or overflow event.

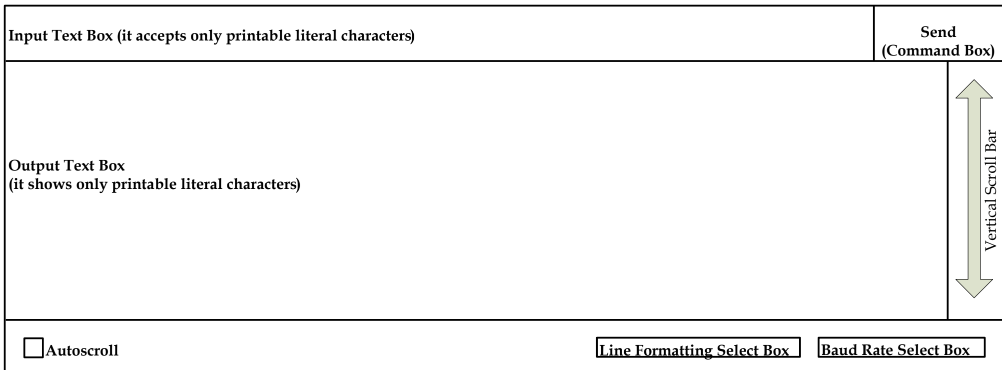
(9) Whenever a roll-over (overflow) occurs, the TOV1 (T1 Overflow Flag) of the MCU assumes LH-state. The user program may continuously poll this bit to know the exact time of the occurrence of a roll-over event.

(10) The TC1 could be loaded with a known pre-set value. When the TC1 is started, it begins counting from the pres-set value. Changing the value of the pre-set parameter, we can change the timing of occurrence of the roll-over event. This feature can be very well utilized to generate varying time delay functions.

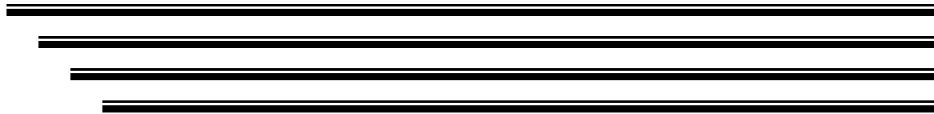
(11) There is also another way of knowing the arrival of the roll-over event, and it is the interrupt method. When the switches I (Global Interrupt Enable Bit via SREG-Register) and TOIE1 (T1 Overflow Interrupt Enable) of Fig-5.1 are kept in closed conditions, the MCU is automatically interrupted by the roll-over event. The MCU goes to the interrupt subroutine (ISR) at location 001Ah, performs the assigned tasks and then goes back to the mainline program (the interrupted program).

6
Interrupt Structure and Programming

7 b
USART Port Organization and Programming



8
SPI Port Organization and Programming



9

Two-wire Port Organization and Programming

10
Fuse Bits Organization and Programming

11
Lock Bits Organization and Programming

12

Interfacing of Peripheral Modules and Sensors

12.1 Organization Interfacing and Programming of ASCII-type LCD

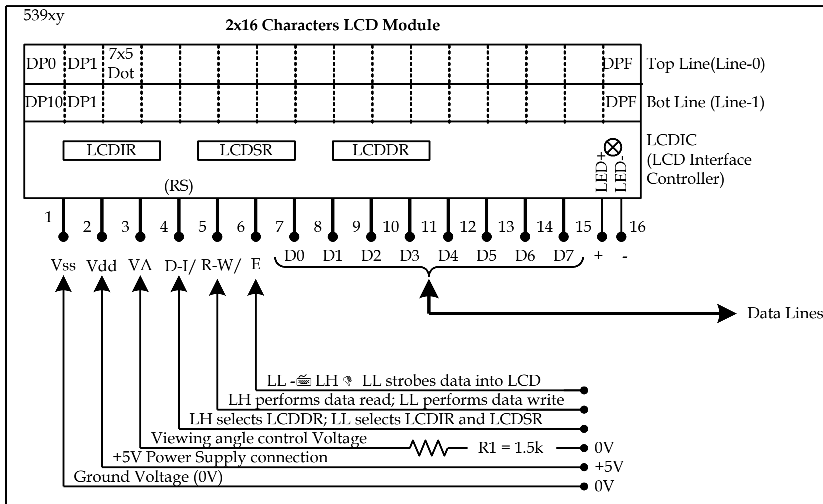


Figure-12.1: *Physical pin diagram of 16-charater 2-line LCD*

12.1.1 Organization of LCD

In Fig-12.1, we have presented a Liquid Crystal Display (LCD) unit. It has two lines: Top Line and Bot (Bottom) Line. Each line can show 16-charcater whose positions are shown as DP0 (display position-0), DP1, …, DPF (DP15). The LCD offers a 7x5 dot matrix block for the presentation of a character or any other customized sysmbol.

The LCD is an ASCII (American Standard Code for Information Interchange) –type display. The name has come from the fact that an English character will appear on the LCD whenever a user sends an 8-bit ASCII of that character into the LCD. For example: to see the character A on the LCD, we must send its ASCII code which is 01000001B (41h). The values of the English characters are given in Section-12.16.

There is a tiny microprocessor inside the LCD, which communicates with the external host computer (the ATmega328). The communication is maintained with the help of the following three 8-bit registers.

- LCDIR : LCD Instruction Register
  This is a writeable register. It takes various command (control) bytes from the MCU as per 'LCD Instruction Set (Section-12.1.7)' for the purpose of initialization. The command bytes are then distributed among different operational modules of the LCD.

- LCDDR : LCD Data Register
  This is a writeable register. It takes the data (8-bit or 2-nibble of 4-bit) from the MCU and shows the corresponding printable ASCII character on the display.

- LCDSR : LCD Status Register
  This is a readable register. The MCU reads this register to know various operational status of the LCD. For example: the LL-value of the busy flag (BF) indicates that the LCD (LCDIR/LCDDR) is ready to accept the next data (control/data) byte. If the register is not available for reading (Fig-12.2), the user cannot access the BF to check the readiness of the LCD. In this case, the user must insert 'Time Delay' after writing a data byte into the LCD. The time delay allows the LCD to digest the current data byte before getting ready to receive the next data byte.

12.1.2 Interfacing of LCD with ATmega328 of Arduino UNO

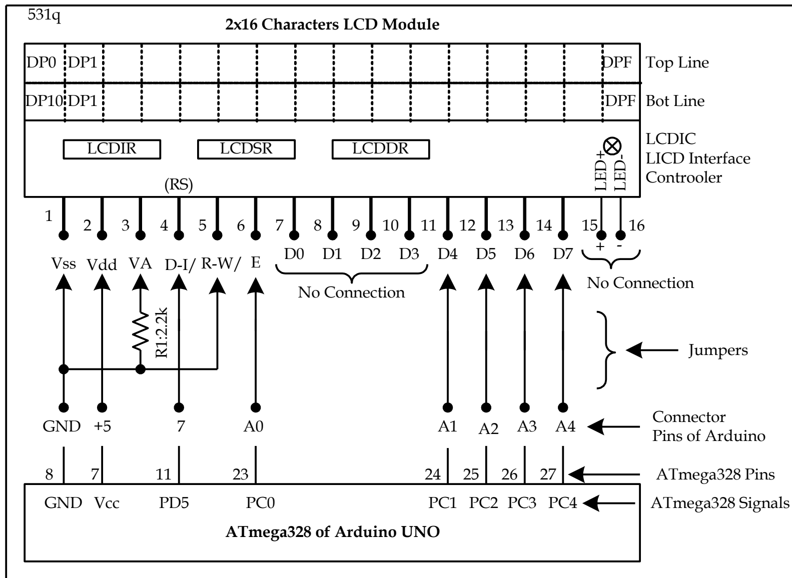


Figure-12.2: *Interfacing connection diagram between LCD and ATmega328 of Arduino UNO Kit*

(1) For acceptable visibility of the printed characters on the LCD, the recommended voltage at VA (viewing angle) point: 3.3V – 3.7V, which could be accurately set using a potentiometer. For the sake of convenience, we have inserted a 2.2k (R1) register between VA-point and GND.

(2) The R-W/ (Read or Write) pin (Pin-5) is tied to GND voltage. This arrangement has kept the LCD always in the 'write mode'.

(3) Before putting the LCD into operation, it is initialized by writing few control bytes (Section-12.1.6) into the LCDIR. The LCDIR is selected by putting LL at Pin4 (D-I/ pin: D for Data Register and I for Instruction Register). The LCDR is selected by putting LH at the D-I/ pin of the LCD.

(4) The LCD exchanges 8-bit data with the host MCU. To minimize pin counts, the LCD can be operated using only the upper four (D4 – D7) data lines. This is known as 4-bit interfacing of the LCD. When operated in 4-bit mode, the original 8-bit data is sent as two 4-bit nibbles with MS-nibble first.

(5) After selection of the register and R-W/ function, we place data on the data lines and then we pulse the E-pin (enable) from LL → LH → LL to accomplish data exchange process.

12.1.3 Programming of LCD

(1) The following Control Bytes are sent, in sequence, into the LCDIR to initialize the LCD.

Control Byte-3 : Display ON/OFF; Cursor ON/OFF; Cursor BLINK/NOTBLINK.
Control Byte-4 : Cursor Shift/Display Shift; Cursor Shift RIGHT/LEFT.
Control Byte-5 : 8/4 Data Lines; 2 Lines LCD; 7x5 Font-size.

(2) The cursor position is set before writing a character (s) into the LCD. It is done by sending Control Byte-7 into the LCDIR.

Control Byte-7 : Cursor Position (Line TOP/BOT; DP0 to DP15 Position)

(3) Data byte is sent into LCDDR

12.1.4 Arduino Programming Language Header File, Function, and Method for LCD

(1) #include <LiquidCrystal.h>
This file contains the definitions, meanings, codes for the symbols, symbolic names, functions, and methods related with the programming of LCD.

(2) LiquidCrystal lcd(RS, E, D4, D5, D6, D7)
This function passes the connection information (Fig-12.2) to other executive functions.

(3) lcd.begin()

This function/method carries out all tasks of Step-1 of Section-12.1.3 to initialize the LCD.

(4) lcd.setCursor (DPX, LX)
This functions sets the cursor position at the designated ‘Display Position’ of the ‘Designated Line’.

(5) lcd.clear()
This function clears the display by sending Cbyte-0 of the Instruction Set.

(6) lcd.print(‘A’)
Lcd.print(“AB”)
Lcd.print(0x41)
Lcd.print

12.1.5 ASCII Code Set

12.1.6 LCD Instruction Set

12.1.7 Exercises

LCD stands for Liquid Crystal Display. It is an ASCII/Text type display which bears the following meaning: *In order to see a visible character (A – Z, a – z, 0 – 9, and Punctuation Marks) on the LCD, the user program must send into LCDDR the 8-bit ASCII code of that character.* The ASCII (American Code for Information Interchange) codes of the printable characters of the English Language are given below:

***(i)* Control Characters (not printable):**

NULL (Null)	: 00h	SOH (Start of Heading)	: 01h
STX (Start Text)	: 02h	ETX (End Text)	: 03h
EOT (End of Transmission)	: 04h	ENQ (Enquiry)	: 05h
ACK (Acknowledge)	: 06h	BEL (Bell Sound)	: 07h
BS (Backspace)	: 08h	HT (Horizontal Tab)	: 09h
LF (Line Feed)	: 0Ah	VT (Vertical Tab)	: 0Bh
FF (Form Feed)	: 0Ch	CR (Carriage Return)	: 0Dh
SO (Shift Out)	: 0Eh	SI (Shift In)	: 0Fh
DLE (Data Link Escape)	: 10h	DC1 (Direct Control 1)	: 11h
DC2 (Direct Control 2)	: 12h	DC3 (Direct Control 3)	: 13h
DC4 (Direct Control 4)	: 14h	NAK (Negate Acknowledge)	: 15h
SYN (Synchronous Idle)	: 16h	ETB (End Transmission Block)	: 17h
CAN (Cancel)	: 18h	EM (End of Medium)	: 19h
SUB (Substitute)	: 1Ah	ESC (Escape)	: 1Bh
FS (From Separator)	: 1Ch	GS (Group Separator)	: 1Dh
RS (Record Separator)	: 1Eh	US (Unit Separator)	: 1Fh
SP (Space)	: 20h	DEL (Delete)	: 7Fh

***(ii)* Punctuation Marks Characters:**

! (Note of Exclamation)	: 21h	" (Double Quote)	: 22h
# (Numeric Sign)	: 23h	$ (Dollar Sign)	: 24h
% (Percent Sign)	: 25h	& (Ampersand Sign)	: 26h

‘ (Single Quote) : 27h
((Opening Parenthesis) : 28h
) (Closing Parenthesis) : 29h
* (Asterisk) : 2Ah
+ (Plus Sign) : 2Bh
, (Comma) : 2Ch
- (Minus Sign) : 2Dh
. (Period) : 2Eh
/ (Forward Slash) : 2Fh
: (Colon) : 3Ah
; (Semicolon) : 3Bh
< (Less then) : 3Ch
= (Equal) : 3Dh
> (Greater then) : 3Eh
? (Question Mark) : 3Fh
@ (At the rate of) : 40h
[(Opening Bracket) : 5Bh
\ (Backward Slash) : 5Ch
] (Closing Bracket) : 5Dh
^ (Exponentiation Sign) : 5Eh
_ (Dash Sign) : 5Fh
` (Single Quote) : 60h
{ (Opening Brace) : 7Bh
| (Pipe) : 7Ch
} (Closing Brace) : 7Dh
~ (Tide) : 7Eh

***(iii)* Number Characters:**
0 (Numeral Zero) : 30h
1 : 31h
2 : 32h
3 : 33h
4 : 34h
5 : 35h
6 : 36h
7 : 37h
8 : 38h
9 : 39h

***(iv)* Upper-case and Lower-case Alphabets:**
A , a : 41h, 61h
B, b : 42h, 62h
C, c : 43h, 63h
D, d : 44h, 64h
E, e : 45h, 65h
F, f : 46h, 66h
G, g : 47h, 67h
H, h : 48h, 68h
I, i : 49h, 69h
J, j : 4Ah, 6Ah
K, k : 4Bh, 6Bh
L, l : 4Ch, 6ch
M, m : 4Dh, 6Dh
N, n : 4Eh, 6Eh
O, o : 4Fh, 6Fh
P, p : 50h, 70h
Q, q : 51h, 71h
R, r : 52h, 72h
S, s : 53h, 73h
T, t : 54h, 74h
U, u : 55h, 75h
V, v : 56h, 76h
W, w : 57h, 77h
X, x : 58h, 78h
Y, y : 59h, 79h
Z, z : 5Ah, 7Ah

Before we put the LCD into operation, we need to carry out the following initialization tasks, on the LCD, as has been laid down in the Instruction Set of the LCD.

(1) Sending Control Byte-0 (01h) into LCDIR to clear the display,

(2) Sending Control Byte-2 (06h) into LCDIR to configure that the cursor will shift and not the display.

(3) Sending Control Byte-3 (0Fh) into LCDIR to configure:
  (a) Display will be ON,
  (b) Cursor will remain ON, and
  (c) Cursor will blink.

(4) Sending Control Byte-4 (14h) into LCDIR to configure that:
  (a) Cursor will shift and
  (b) Cursor will shift right.

(5) Sending Control Byte-5 (3Ch) into LCDIR to configure:
- (a) 8 data lines,
- (b) 2 Line LCD, and
- (c) 10x5 (Row and Column) Font-size.

Before we send the ASCII code on the LCD to print a character, we need to send into LCDIR, the printing/cursor position (Line and display position). This is done by writing Control Byte-7 (80h – 8Fh for Top Line and C0h – CFh for Bottom Line) into LCDIR.

After writing a data byte into LCDIR or LCDDR, we must insert some time delay by executing the following Time Delay Subroutine. This Time Delay allows the LCD to absorb current data byte written into it.

```
TDEL:           mov     cx, 00FFh
AGND:     loop          AGND            ; again decrement
          ret
```

Arduino Functions/Methods to perform data read/write operations with LCD.

```
                    RS = Register Select (LCD Instruction or Data Register)
                         E = Enable pin for data strobe inside the LCDIR/LCDDR
                              D4 = Data bit-4
                                   D5 = Data bit-5
                                        D6 = Data bit-6
                                             D7 = Data bit-7
LiquidCrystal lcd(RS,   E,   D4,  D5,  D6,  D7);   ← Arguments of the function
                 (A4,   A5,  A0,  A1,  A2,  A3);  ← Arguments must be defined in terms of known connector pins
                   (PC4  PC5  PC0  PC1  PC2  PC3)  ← The signals of ATmega328, which drive the values of the arguments
              (27   28   23   24   25  26)   ← The pins of the ATmega328 Microcontroller
```

* R-W/ pin of the LCD must be connected to GND point. The LCD will always remain in write mode.

13
ATmega328 Based System Design

13.1 Schematic Diagram of Digital Taxi Meter using Arduino UNO Learning Kit

- Full Member
- Posts: 207
-

1.
2.

- Karma: 6 [add]
- 58.97.132.119

Relationship among PORTX, PINX, DDRX, pinMode(), ... of Arduino
May 08, 2017, 06:23 pm Last Edit: May 09, 2017, 01:04 pm by GolamMostafa

4.2 Relationship among PORTX, PINX, DDRX; pinMode(); digitalWrite(), bitSet(), bitClear(), bitWrite(); digitalRead(), bitRead(); PORTX = 0xNN, DDRX = 0xNN

1. PORTB: PORTB stands for Port-B Register (PBR). It is a latch type (Flip-flop) register; it receives data from the Processor Unit (PU) for the output port-lines (PB5-PB0). We may recognize PORTB as a 'Data Transmission Register' for those port-lines which have been configured to work as output. In the present case, we have only one output port-line (PB5).

There is no internal pull-up resistor associated with an output port-line.

2. PINB: PINB stands for Pin-B Register. It receives data from the pins of the input port-lines (PB5-PB0). We may recognize PINB as a 'Data Reception Register' for those port-lines which have been configured to work as input. In the present case, we have only one input port-line (PB0).

Every input port-line is associated with an internal pull-up resistor, which (by default) remains disconnected. It can be connected with input line using program instruction.

3. DDRB: The full-name of DDRB is Data Direction Register for port-lines (PB5-PB0). This register helps to set the directions of the IO lines (port-lines) either as input or output depending on the requirement.

To set the direction of a port-line (say, PB5) as output, we write LH (1) at the corresponding bit (DDRB5) of the DDRB register. To set direction as input, we store LL (0) at the corresponding bit of DDRB.

4. Setting Direction of Port-line:
(a) To set the the direction of a single port-line (PB5) as output, we use the following function:
pinMode(DPin, value);
===> *pinMode(13, HIGH);*

(b) To set the the direction of a single port-line (PB0) as input with internal pull-up resistor connected, we use the following function:
pinMode(DPin, value);
===> *pinMode(8, INPUT_PULLUP);*

(c) To set the the direction of all the six (6) port-lines (PB5-PB0) as output at the same time, we take the help of DDRB register and use the following assignment statement:
DDRB = B00111111; //Leading two zeros are padding bits; B is notation for binary
===> *DDRB = 0xFF;* //0x is notation for hexadecimal number.

(d) To set the the direction of all the six (6) port-lines (PB5-PB0) as input (without internal pull-

up
register) at the same time, we take the help of DDRB register and use the following assignment statement:
DDRB = B00000000; ===> *DDRB = 0x00;*

(e) To connect internal pull-up resistors with all the input port-lines (PB5-PB0) of Step-4d, we use the following assignment statements:
DDRB = B00000000; // port-pins must be set as inputs
PORTB = B00111111; // 1s should be written to PORTB see Figure-2
LL ---> PUD-bit (Bit-4) of MCUCR; bitClear (MCUCR, 4); //see Figure-2

(f) In summary: (due to CrossRoads of Arduino Forum)
Or just use one of these in setup(), easier to remember and not mess it up:
pinMode (pinX, INPUT); // pinX = 0 to 13, A0 to A5
pinMode (pinX, INPUT_PULLUP);
pinMode (pinX, OUTPUT);

5. Writing Data into Port-line:

(a) To write 1-bit (0 or 1) into a port-line (PB5), we use the following function:
(*i*) *digitalWrite(DPin, Value);*
===> *digitalWrite(13, HIGH);*

(*ii*) *bitSet(13);* //writing LH at PB5
bitClear(13); //writing LL at PB5

(*iii*) *bitWrite(PORTX, n, x);* // X = B/C/D, n=Bit_position, x = Bit_value
===> *bitWrite(PORTB, 5, HIGH);*

(b) To write to all port-lines (PB5-PB0) at the same time, we use the following statements:
PORTB = 0xFF; // 1s are written to all pins
PORTB = 0x00; // 0s are written to all pins

(c) In summary
digitalWrite(Dpin, Value); //DPin = 0 to 13, A0 to A5
bitSet(VAR, n); // VAR (variable) PORTX, DDRX; n = 0 to 7
bitClear(VAR, n);
bitWrite(VAR, n, x); // x = HIGH or LOW

6. Reading Data from Port-line:

(a) To read 1-bit data from a port-line, we use the following functions:
(*i*) *boolean x = digitalRead(8);* //reading 1-bit data

(ii) *boolean x = bitRead(PORTB, 0);*

(b) To read data from all the port-lines at the same time, we use the following assignment statement:
byte x = PINB;

(c) In summary
boolean y = digitalRead(Dpin); //DPin = 0 to 13, A0 to A5
boolean z = bitRead(VAR, n); // VAR (variable) PINX; n = 0 to 7
byte = PINX; // PINX = PINB, PINC, PIND

7. Examples:

(a) Write C/C++ codes to check that K1-switch of Figure-1 is closed and only then keep blinking L (built-in LED) at 1-sec interval. Test the program (sketch) using Arduino UNO Kit. Use *goto* statement and *if-else* structure.

Code: [Select]

```
void setup()  //keep under this functions those tasks that are executed only for once/limited
number
                                                      //of times.
{
  //set direction of IO lines as needed (known as initialisation)
  pinMode(13, OUTPUT);                  //PB5 as outut port-line
  pinMode(8, INPUT_PULLUP);   // PB0 as input port-line with internal pull-up resistor enabled

  //check if K1 is closed or not
  L1: boolean x = digitalRead(8);
  L2: if (x != LOW)
    goto L1;
}

void loop()  //keep under this function those tasks that keeps executing
{
  digitalWrite(13, HIGH);
  delay(1000);
```

```
bitWrite(PORTB, 5, LOW);
delay(1000);
}
```

8. Exercises:

(a) Repeat Example-(a) using *while-do* structure instead of *goto, if-else.*

(b) Discuss merits and demerits of the programs of Example-(a) and Exercise-(a).

(c) Repeat Exercise-(a) by including *cli()* instruction under *setup()* function (before *while-do*). Observe that the L is not blinking even though you have closed K1. Find the exact reason for the L not to blink and then solve the problem by keeping the *cli()* instruction.

13. Clock Distribution in ATmega328 Microcontroller

7.2 Organization and Programming of SRAM Data Memory

(1) Apart from EEPROM Data Memory, there is another block of data memory inside the ATmega328 MCU; it is known as SRAM (Static RAM) Data Memory. There is a particular section of space in this data memory which is reserved for the storage of variables, temporary results, and etc.; it is known as User RAM or simply RAM.

(2) The conceptual view of the SRAM Data Memory is given below:

(3)

Address of Byte Location SRAM Address accessed by lds/sts	IO Address Accessed by in/out instruction	Register Name Can be Used with in/out	Content 7 0	Section	Size
$0000		r0		Register File	32 Bytes
$001F		r31			
$0020	$0000			IO Space	64 Bytes
	$0001				
	$0002				
	$0003	PINB			
$005F	$003F	SREG			
$0060	$0040	WDTCSR		Extended IO Space	116 Bytes
$00E6	$00C6	UD0			
$07FF					
$0800			Stack grow ↑	Stack Space	256 Bytes
$08FF(Top of Stack)					

sram

5.1 Organization and Programming of Code Memory (Flash), Data Memory (EEPROM), RAM, Stack, Register File, Fuse Bits, and Security Bits

5.1.1 Organization and Programming of Code Memory (Program Memory = Flash)

Figure-5.1: Conceptual organization of Code (Program/Flash) memory

5.1 Organization of Flash (Code Memory/Program Memory) of ATmega328 MCU

5.1 Organization of Flash (Code Memory/Program Memory) of ATmega328 MCU

(1) In the Lab using Arduino IDE, we have compiled our application programs (the sketches) to obtain binary codes (machine codes). However, we have not yet seen these binary codes which the MCU executes to produce the desired result like blink L (built-in LED of Arduino UNO Kit).

(2) We transferred (uploaded) the binary codes of Step-1 to the ATmega328 Microcontroller (MCU) of the Ard ... memory space of the MCU. This block c ... y = Program Memory). ... called Flash? "The term ... uld be erased at one time. The name ... where each byte is erased individually."

Address of Word organized Memory Location	Content (Word = 16-Bit)		Section
	High Byte	Low Byte	
	15 8	7 0	
(16383) $3FFF			Boot Section
$3C00			
$3BFF			Application Section
$0043	10111111	00001111	
$0042	11100000	00001000	
$0041	10111111	00001101	
$0040	11100101	00001111	
$003F			Interrupt Vector Section
(0) $0000			

codeMemOrg : GM 5/17

Figure-5.1

(3) Conce... ash) of ATmega328

(4) Inside t... lled Memory Locations.

(5) Every me... code called Address (adr).

(6) There are in total 16x1024 (16384) locations inside the code memory of ATmega328. The address of the 1st location is 0 ($0000) and the address of the last location is 16383 ($3FFF). $ is the notation for hexadecimal number; "0x notation for hexdecimal number is preferred while coding in assembly."

(7) (a) Every memory location can hold 16-bit (word = 2-byte) data/code. The 16-bit data is split as Low Byte and High Byte.

(8) (a) The Code Memory of ATmega328 of Arduino Kit has three sections:
(i) Interrupt Vector Section,
(ii) Application Section, and
(iii) Boot Section.

(b) This is the application section into which the machine codes of our sketches are stored during uploading.

(c) The machine codes of an application program have two parts: (*i*) Executable Codes and (*ii*) Data on which the Executable Codes work. The executable codes are stored in the 'Application section' of the Code Memory and the data are stored in the 'SRAM = Static RAM Memory' area of the MCU." The SRAM would be discussed in a
separate post. It might happen that data (variable) are stored into Code Memory to save SRAM space.

(d) The boundary addresses and operational modes of the memory sections are determined by Fuse Bits.
(e) The access rights to Boot and Application sections are controlled by:
(i) Boot Lock Bits,
(ii) Application Lock Bits,

(9) The Code Memory is a non-volatile EEPROM type memory. Volatile memory loses its information (data) when it is disconnected from power supply. EEPROM stands for Electrically Erasable Electrically Programmable Random Access Ream Only Memory.

"Flash is one type of EEPROM, with a limited number of re-write cycles (10,000 times). The EEPROM is a different type, with more re-write cycles (100,000). RAM is Random Access Memory that can be read and written. (Vs what you spelled out, RARWM). '328P and other AVR devices use Static RAM (SRAM) (vs Dynamic RAM, or DRAM, such as is found in a PC).
SRAM can be written and retains its data as long as power is available. DRAM must be continuously refreshed to retain the data."

(10) Calculation of Code Memory Capacity
(i) 16x1024 word locations
==> 16 K word locations (in Computer Science, 1024 stands for 1 K)
Word locations are individually accessible by the processor during instruction fetching and execution.

(ii) 16x1024 2-byte locations

==> 16x102x2 byte locations
==> 32 K byte locations
Byte locations can also be individually accessed during Paralle/Serial Programming.

(iii) 32x1024x8 bit locations (8-byte = 8-bit)
==> 262 144 bits
The bit locations cannot be accessed individually. In 8051, there are a good number of bit addressable locations in SRAM.

The Code Memory contains limited amount of memory space. So, we must exercise the art of writing program with minimum number of instructions in order consume less memory space.

(11) Programming the Code Memory refers to the processes of Erasing, Writing, and Verifying its contents. Verification is done to ensue that the programmer has correctly written source data into the designated locations; this is done by reading back the fused data and comparing it with the original source data. Some programmers avoid verification to speed up the programming process.

(12) Erasing of the Code Memory refers to turning all the bits of the memory into LH states. The erasing is done by one of the following methods:
(i) Parallel Programming and
(ii) Serial Programming (In-system Programming = ISP).

(13) New information (code/data) can be written into code memory using one of the following methods:
(i) Parallel Programming,
(ii) Serial Programming (In-system Programming = ISP), and
(iii) Self-programming.

(14) The numerical values of the addresses of the code memory and data memory (EEPROM) locations come into picture only when we write programs using Assembly Language. The High Level Language like C/C++ hides to the user all kinds of low level activities. For example:

Code: [Select]

```
byte myArray[2] = {0x3F, 0x66};   // HLL Code

//---ASM Code------
ldi r16, 0x3F
sts $0600, r16   //$0600 is the address of a Data Space (not shown here)
```

(15) Binary Code (of an application program) storage organization into Code Memory
Assume that the following machine codes (written as hex bytes as a matter of

convenience) represent some actions to be carried out by the processor (the microcontroller). These codes must be loaded first into code memory. The arrangement of storage is shown in Fig-5.1.

```
000040  e50f   ldi  r16, 0x5F  ; stack initialize
000041  bf0d   out  spl, r16
000042  e008   ldi  r16, 0x08
000043  bf0e   out  sph, r16
```

5.2 Home Works

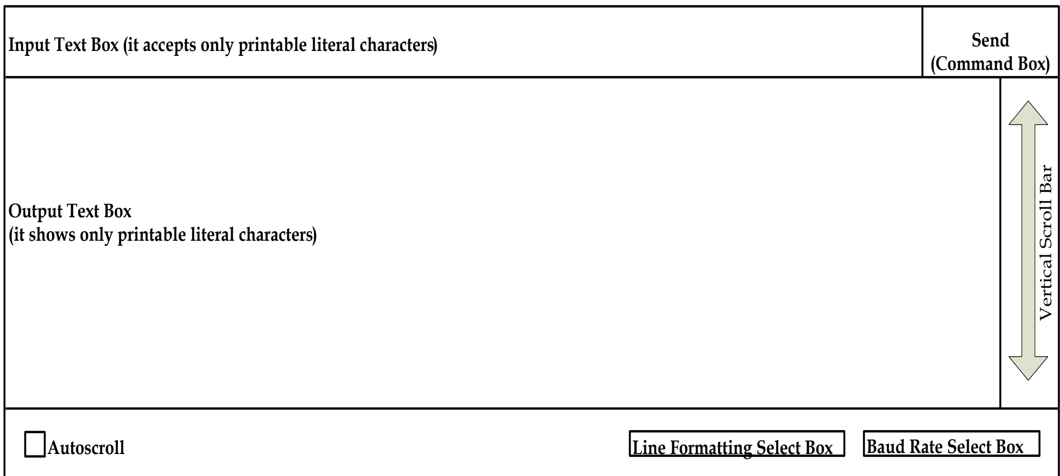
(1) Download data sheets for ATmega328 and ATTiny85 microcontrollers.

(2) Draw conceptual diagram for the Code Memory of ATTiny85 MCU.

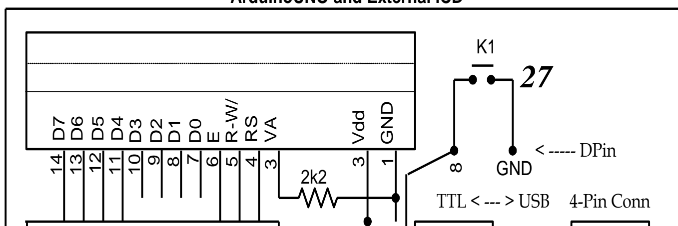
Address of Byte Location			Content	Section	Size
SRAM Address accessed by lds/sts	IO Address Accessed by In/out	Register Name	7 0		
$0000		r0	8-bit	Register File	32 Bytes
$001F		r31			
$0020			8-bit	IO Space	64 Bytes
$005F					
$0060			8-bit	Extended IO Space	116 Bytes
$00FF					
$0010			8-bit	User	1792

USART Interface for Asynchronous Serial Communication between Serial Monitor of Arduino IDE (at IBMPC Side) and the ATmega328 MCU (at Arduino Side)

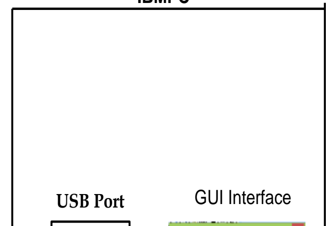
Serial Monitor (Expanded)



ArduinoUNO and External ICD



IBMPC



(1) Assume that all the needed initializations are done; the Receiver and Transmitter of the USART of the ATmega32 of ArduinoUNO are ready to operate in normal way. It is also assumed that the Serial Monitor of the Arduino IDE at the IBMPC side is also ready to work.

(2) Transmitter Section of ATmega328

(a) The Tx (Transmitter) always sends 8-bit binary formatted data to the Host Computer (the IBMPC in the present example) in asynchronous frame format (Fig-1).

(b) The lowest level of instructions (known as Machine Codes) to send an 8-bit data (say 0x37 = B00110111): (r16 register holds the data byte, 0x37.) Let us remember that these are the machine codes and not the ASM or C which the microcontroller executes.

```
000040    -    e307
000041    -    b90c
```

(c) The next higher level of instructions (Assembly Codes) to send the 8-bit data of Step-1: (r16 register holds the data byte 0x37.)

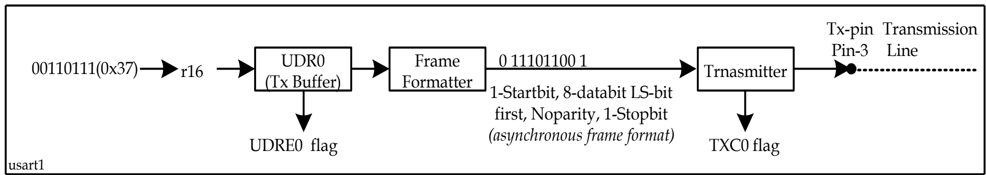
```
.org    0x0040
ldi     r16, 0x37
out     UDR, r16
```

(d) The next higher level of instructions (C Codes) to send the 8-bit data of Step-1:

```
Serial.write(0x37);
```

Is this C code is equivalent to 2-line assembly code of Step-2c? We will see it later.

(3) Path followed by the data (0x37) to arrive at physical Tx-pin (Pin-3) of the ATmega328.



(a) 8-bit data enters into Tx Buffer (UDR0 register) in parallel format. From Tx Buffer, the data shifts into Frame Formatter in serial format. When the data shift is complete, the Tx Buffer becomes empty and the UDRE0 flag assumes LH-state. The default value of UDRE0 flag is LH, meaning that the Tx Buffer is ready to accept data. This flag can be used by the programmer in the following ways to write new data byte into the UDR0 register (Tx Buffer).

(i) Continuous polling of the UDRE0 flag to see that it has assumed LH-state. This process is known as Polling Method.

(ii) Let the UDRE0 flag interrupt the user when it assumes LH-state. This process is known as Interrupt Method.

(b) Actual data transmission is carried out by the Transmitter. It shifts out (transmits) data bit-by-bit. When the data transmission is complete (all data bits of the frame have shifted out from the Transmitter), the TXC0 flag assumes LH-state. The default value of TXC0 flag is LL. This flag can be used by the programmer in the following ways to write new data byte into the UDR0 register (Tx Buffer).

(i) Continuous polling of the TXC0 flag to see that it has assumed LH-state.

(ii) Let the TXC0 flag interrupt the user when it assumes LH-state.

(4) Program Codes to write data into Tx Buffer by Polling the UDRE0 flag

(a) Pseudo Codes

```
L1:  If (UDRE0 ==LH)
         {
             Write data byte into UDR0
             Clear UDRE0 flag by writing LH into this bit position.
         }
     Else
         Goto     L1
```

(b) ASM Codes

```
L1:  in    r16, UCSR0A        ; Bit-5 is the UDRE0 flag
     Rlc   r16
     Rlc   r16
     Rlc   r16
     Brcc L1               ; UDRE0 flag = LL; Tx Buffer is not empty
L1A:ldi   r16, 0x37
L1B:Out UDR0, r16
```

(c) Literate C Codes in Arduino IDE

```
L1:  while (bitRead(UCSR0A, 5) != HIGH)    //checking UDRE0 flag for Tx Buffer's emptiness
```

```
        ;
L1A:bitWrite(UCSR0A, 5, 1);          //clear the UDRE0 flag
    L1B:UDR0 = 0x37;                 //write data into Tx Buffer
```

(d) Nested C Codes in Arduino IDE

```
L1:  Serial.write(0x37);
```

(5) Program Codes to write data into Tx Buffer by Polling the TXC0 flag

The codes for Pseudo, ASM, Literate C, and Nested C would be exactly similar to the codes of Step-4, but these are now to be modified and mapped for TXC0 flag.

(6) Program Codes to write data into Tx Buffer being interrupted by the UDRE0 flag

(a) Pseudo Codes

```
Enable UDRIE0-bit (USART Data Register Empty Interrupt Enable 0)
Enable I-Bit of SREG
Wait for Interrupt

ISRTX:
Clear UDRE0 flag
Write data into Tx Buffer via UDR register
Return from interrupt
```

(b) ASM Codes

```
L1:  in    r16, UCSR0A       ; Bit-5 is the UDRE0 flag
     Rlc   r16
     Rlc   r16
     Rlc   r16
     Brcc L1                 ; UDRE0 flag = LL; Tx Buffer is not empty
L1A:ldi   r16, 0x37
L1B:Out UDR0, r16
```

(c) Literate C Codes in Arduino IDE

```
L1:  while (bitRead(UCSR0A, 5) != HIGH)    //checking UDRE0 flag for Tx Buffer's emptiness
        ;
L1A:bitWrite(UCSR0A, 5, 1);          //clear the UDRE0 flag
    L1B:UDR0 = 0x37;                 //write data into Tx Buffer
```

(d) Nested C Codes in Arduino IDE

```
L1:  Serial.write(0x37);
```

(5) Program Codes to write data into Tx Buffer by Polling the TXC0 flag

The codes for Pseudo, ASM, Literate C, and Nested C would be exactly similar to the codes of Step-4, but these are now to be modified and mapped for TXC0 flag.

```
Serial.write('\n');  //CR(0Dh) and LF(0Ah)
 Serial.write(0x34); //sends 8-bit binary for 34h (=ASCII); shows 4
 Serial.write('\n');  //CR(0Dh) and LF(0Ah)
```

```
Serial.print(36, 16); //sends ASCII codes for digit 2 and 4
Serial.write('\n');  //CR(0Dh) and LF(0Ah)
Serial.print(36, 10); //sends ASCII for 3 and 6
Serial.write('\n');  //CR(0Dh) and LF(0Ah)
Serial.print(0x41);  //sends ASCII codes for 6 and 5
                 //no ASCII code for CR(0Dh and LF(0Ah)
Serial.println(0x44);//sends ASCII for 6, 8, CR(0Dh) and LF(0Ah)
Serial.print(36); //sends ASCII for 3 and 6
Serial.write('\n');  //CR(0Dh) and LF(0Ah)
Serial.print(1.23456, 4); //Sends ASCII for 1, point, 2, 3, , 4, 5
```

```
Serial.println(0x44);//sends ASCII for 6, 8, CR(0Dh) and LF(0Ah)
 Serial.print(36); //sends ASCII for 3 and 6
 Serial.write('\n'
```

14.Fuse Bits of ATmega328 Microcontroller

A Fuse Bit, in the context of ATmega328 Microcontroller, can be conceptually visualized as a 'Jumper Wire' which has two conditions:

(a) Unbroken (Unprogrammed; Logic value 1)

(b) Broken (Programmed; Logic value 0)

A Fuse Bit establishes a definite 'Functional Mode' of the microcontroller (MCU) at the time 1st power up initialization when no instruction has yet been executed. For example: the fuse bit named CKDIV8, when it is pr ogrammed, defines that the CPU Clock (clkCPU = clkSYS/8) will be equal to 'Oscillator Frequency/8', and it is shown in Fig-12. In Fig-12.1, the LL (Logic Low = programmed) value of CKDIV8 closes K2 and opens K1.

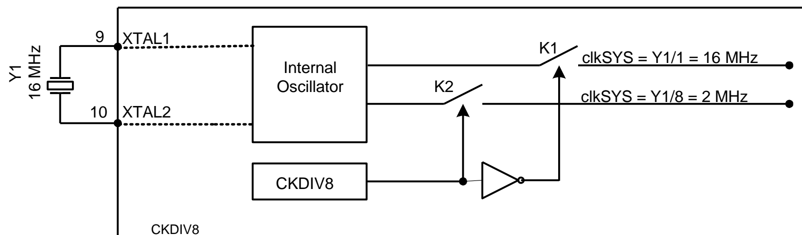


Figure-12.1 *Role of fuse bit CKDIV8 in the determination of system clock frequency*

There are 24 fuse bits inside ATmega328 of which only 19 are active. The value of a fuse bit remains unchanged even in 'Chip Erase' process during Programming Session. These 24 fuse bits are organized as:

(a) Extended Fuse Byte, **(b)** High Fuse Byte, **(c)** Low Fuse Byte

(4) The value of a fuse bit can be altered using Parallel Programmer like TOP2013 and the like. The value can also be changed using Serial Programmer; but, it is not recommended.

(5) The value of a fuse bit must be altered before 'Programming the Lock Bits' of the MCU. Once, the lock bits are programmed, the value of the fuse bit cannot be changed. The option that is left is to erase the chip (chip erase brings the lock bits back to their default values) and then set the fuse bits.

2. Extended Fuse Byte, EF (1 = Unprogrammed, 0 = Programmed)

Bit Position →	7	6	5	4	3	2	1	0
Symbolic Name →	-	-	-	-	-	BODLEVEL2	BODLEVEL1	BODLEVEL0
Default Value →	1	1	1	1	1	1	1	1
Arduino Value →	0	0	0	0	0	1	0	1

(EF7 – EF3)No Use in ATmega328 (Reserved for future)

;--

(EF2) BODLEVEL2 Brown-out Detector Trigger Level-2
(EF1) BODLEVEL1 Brown-out Detector Trigger Level-1
(EF0) BODLEVEL0 Brown-out Detector Trigger Level-0

(a) The ATmega328 works on 5V supply (V_{cc}). There is EEPROM memory inside the MCU, which also works on 5V. It is used to save critical data like password,

balance, and etc. The EEPROM is a type of memory that can be programmed (read, write, erase) by ROM Programmer and Program Instructions.

(b) During power up, the V_{cc} slowly rises to 5V. Similarly, during power failure, the 5V (V_{cc}) slowly drops to 0V. It has been observed that the EEPROM erratically gets written when the V_{cc} is below 3V (typical value). As a result, the data of the EEPROM becomes corrupted.

(c) The ATmega328 contains a circuit called 'Brown-out Detector which prevents the EEPROM from being written when the V_{cc} remains below 3V.

(d) The Brown-out detection circuit (Fig-12.2) continuously monitors V_{cc}, and it keeps the MCU at reset state until the V_{cc} crosses the voltage level determined by the fuse bits [BODLEVEL3:BODLEVEL0]. Brown-Out Reset (internal reset) will occur before V_{cc} drops to a voltage where correct operation of the microcontroller is no longer guaranteed.

[BODLEVEL3:BODLEVEL0]	V_{BOT} (Brown-out Trigger Level)
111	BOD circuit is disabled
110	1.7V – 1.8V – 2.0V
101	2.5V – 2.7V – 2.9V
100	4.1V – 4.3V – 4.5V

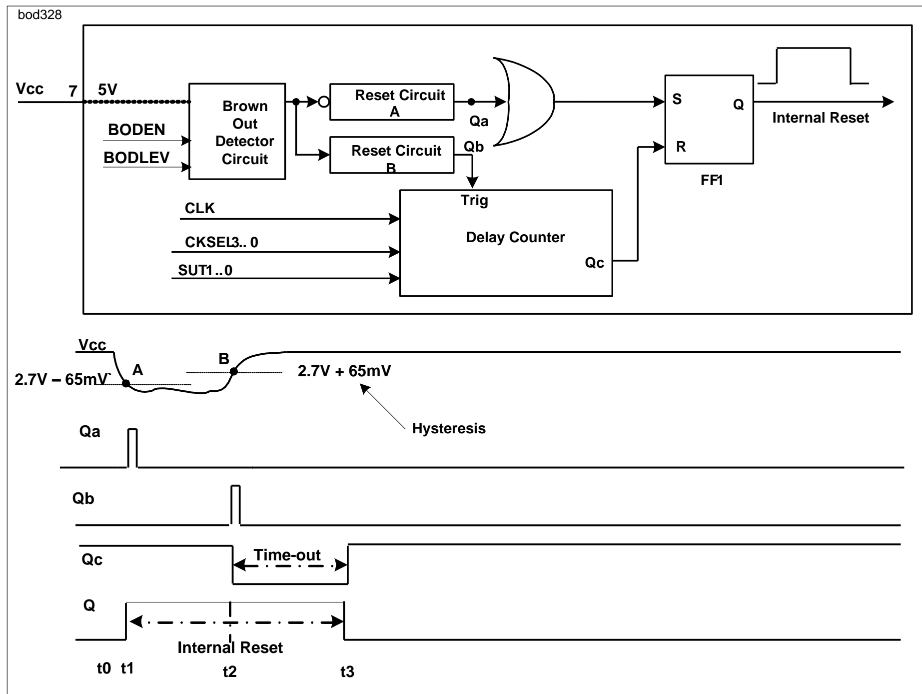


Figure-12.2 *Brown-out detection circuit for ATmega328 Microcontroller*

3. High Fuse Byte, HF (1 = Unprogrammed, 0 = Programmed)

Bit Position →	7	6	5	4	3	2	1	0
Symbolic Name →	RSTDISBL	DWEN	SPIEN	WDTON	EESAVE	BOOTSZ1	BOOTSZ0	BOOTRST
Default Value →	1	1	0	1	1	0	0	1
Arduino Value →	1	1	0	1	1	0	1	0

(HF7) RSTDISABL External Reset Disable (when programmed)

(a) When this fuse bit is not programmed (unprogrammed), the Pin-1 of ATmega328 works as a Reset Pin, and this pin can now receive signal from external hardware device to reset the MCU. Fig-12.2 explains it.

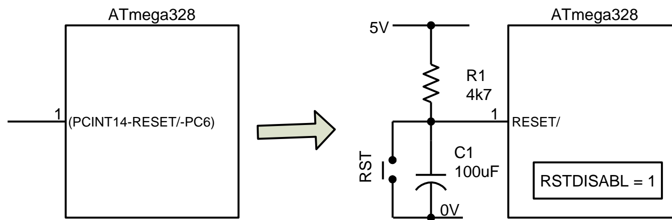


Figure-12.2: *Role of RSTDISABL bit to configure Pin-1 of ATmega328 as Reset/-pin*

(b) If we want to use Pin-1 as a 1-bit digital IO port (PC6) or an interrupt pin (PCINT14), the RSTDISABL bit must be programmed (logic value 0). The choice of option between PC6 and PCINT14 is made by program instruction.

;--

(HF6) DWEN debugWire Enable (when programmed)

;--

(HF5) SPIEN Enable Serial Program and Data Downloading (when programmed)

(1) The program code/data are written into Flash and EEPROM memory using commercial expensive Parallel ROPM Programmer.

(2) The ATmega328 has built-in circuitry called (In System Programming Interface, ISP or SPI?) by which we can write program code/data into the Flash/EEPROM of the MCU. The setup and procedures are elaborately discussed in the data sheets.

(3) The fuse bit named SPIEN, when programmed (logic value 0) allows a user to engage the ISP Interface for the programming of ATmega328.

;--

(HF4) WDTON Watchdog Timer Always On (when programmed)
;--

(HF3) EESAVE EEPROM Memory is preserved through the Chip Erase (when programmed)
During programming of the ATmega328, the Chip is erased to bring all bits of the flash memory at LH (1) states. During this process, the EEPROM and Lock Bits (not the fuse bits) are also erased. However, with the help of the EESAVE fuse bit, the user can preserve the contents of the EEPROM Memory from being erased during 'Chip Erase' cycle.
;--

(HF2) BOOTSZ1 Select Boot Size-1
(HF1) BOOTSZ0 Select Boot Size-0
(HF0) BOOTRST Select Reset Vector

(a) After power up reset (after the reception of reset signal at Pin-1), the ATmega328 looks for a particular memory location in the flash memory from which it will begin program execution. This location is known as Boot Location. The size of Flash memory of ATmega328 MCU: 0000h – 3FFFh (16-bit word organized).
(b) The Boot Location is always 0000h if the BOOTRST fuse bit is not programmed (unprogrammed; logic value 1).
(c) If the BOOTRST fuse bit is programmed (logic value 0), then the combinations of BOOTSZ1 and BOOTSZ0 determine four different Boot Locations. These locations are shown below: (Note: IVSEL bit of MCUCR should be consulted to find the memory locations for the placement of the interrupt vectors.)
(i) (BOOTSZ1, BOOTSZ0, BOOTRST) = (0, 0, 0); Boot Location: 3800h
(ii) (BOOTSZ1, BOOTSZ0, BOOTRST) = (0, 1, 0); Boot Location: 3C00h
(iii) (BOOTSZ1, BOOTSZ0, BOOTRST) = (1, 0, 0); Boot Location: 3E00h
(iv) (BOOTSZ1, BOOTSZ0, BOOTRST) = (1, 1, 0); Boot Location: 3F00h

;--
----

4. Low Fuse Byte, LF (1 = Unprogrammed, 0 = Programmed)

Bit Position →	7	6	5	4	3	2	1	0
Symbolic Name →	CKDIV8	CKOUT	SUT1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0
Default Value →	0	1	1	0	0	0	1	0
Arduino Value →	1	1	1	1	1	1	1	1

(LF7) CKDIV8 Divide Clock by 8 (when programmed)
If the fuse bit named CKDIV8, when it is pr ogrammed, defines that the system clock (clkSYS = clkCPU) will be equal to 'Oscillator Frequency/8', and it is shown in Fig-12.3.

In Fig-12.3, the LL (Logic Low = programmed) value of CKDIV8 closes K2 and opens K1.

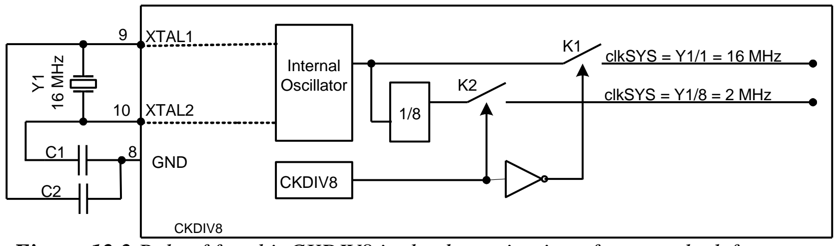


***Figure-12.3** Role of fuse bit CKDIV8 in the determination of system clock frequency*

;--

(LF6) CKOUT Clock Output (when programmed)

;---

(LF5) SUT1 Select start-up time-1
(LF4) SUT0 Select start-up time-0
;---

(LF3) CKSEL3 Select Clock Source-3
(LF2) CKSEL2 Select Clock Source-2
(LF1) CKSEL1 Select Clock Source-1
(LF0) CKSEL0 Select Clock Source-0
The combinations of the fuse bits [CKSEL3:CKSEL0] determine the source of the system clock of the ATmega328 MCU followed by 'optional divisor 8'. The combinations are:

(a) [CKSEL3:CKSEL0] = [1 1 1 1] : External low power crystal (8 MHz – 16 MHz) as is shown in Fig-12.3. Recommended values for C1 and C2 : 12pF – 22pF.

(b) [CKSEL3:CKSEL0] = [0 0 1 0] : Calibrated internal 8 MHz oscillator (Fig-12.4).

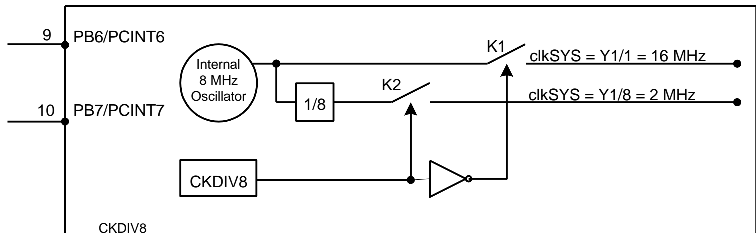


Figure-12.4 *Role of fuse bits (CKSEL3:CKSEL0) in the determination of internal oscillator for clkSYS*