ATmega328P

Narendiran S

July 28, 2020

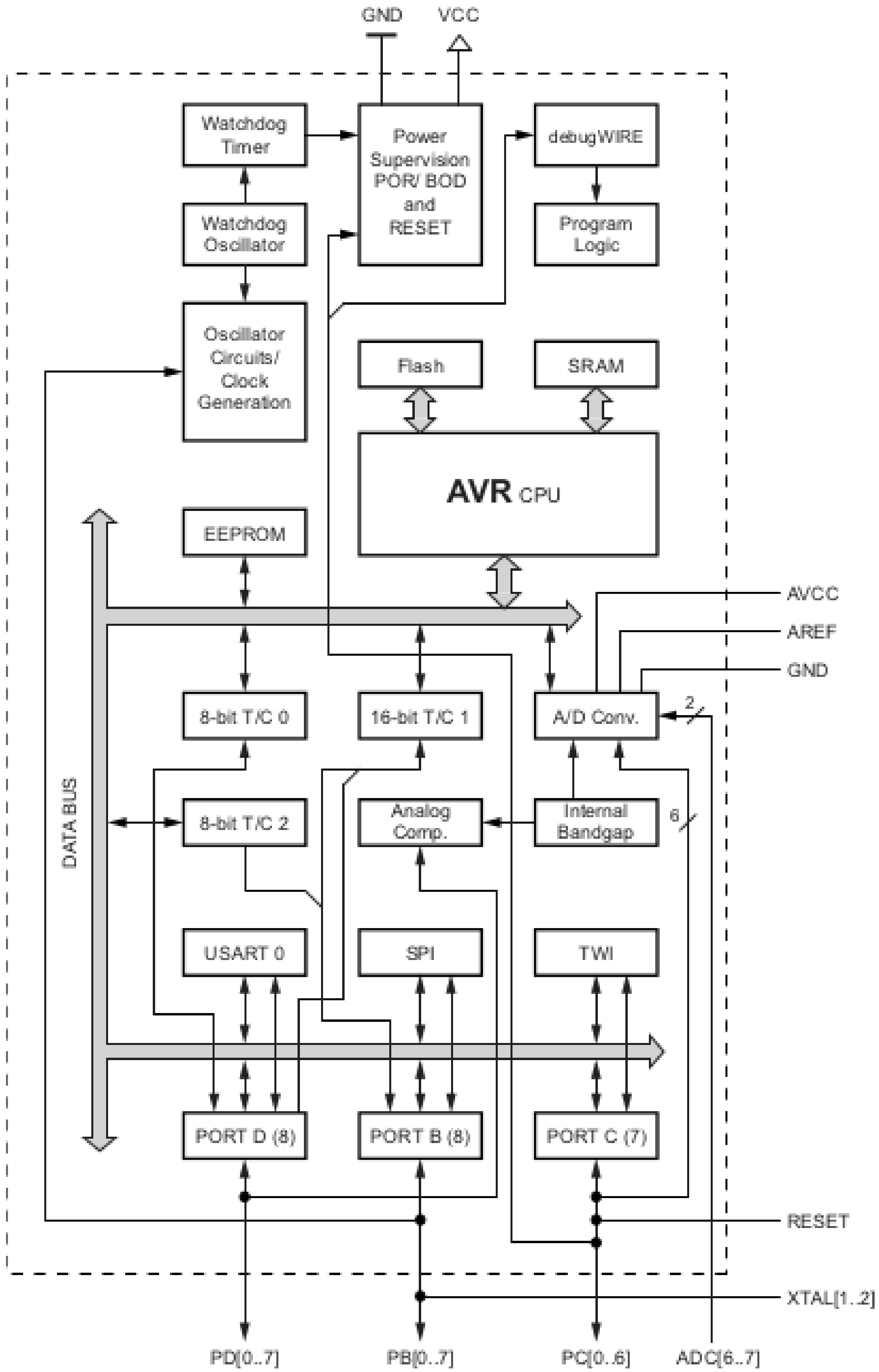
Contents

Atmega328P Basics

1.1 Features

- 8 bit CMOS μC with RISC Architecture
- 32 x 8-bit General purpose registers
- 32 KByte of flash program memory
- 1 KByte EEPROM
- 2 KByte of internal SRAM
- On-chip 2-cycle multiplier
- Optional boot code section with independent lock bits
  - In-system programming by on-chip boot program
  - True read-while-write operation
- Two 8-bit Timer/Counter with separate prescaler and compare mode
- One 16-bit Timer/Counter with separate prescaler, compare mode and capture mode
- Real time counter with separate oscillator
- Six PWM channels
- 6/8(DEPENDING ON PACKAGE) chaneel 10 bit ADC ◦ Also with Temperature measurement
- Programmable serial USART
- 2-wire serial interface (Phillips I2C compatible)
- Programmable watchdog timer with separate on-chip oscillator
- On-chip analog comparator
- Interrupt and wake-up on pin change
- Power-on reset and programmable brown-out detection
- External and internal interrupt sources
- Six sleep modes: Idle, ADC noise reduction, power-save, power-down, standby and external standby
- 2.7V to 5.5V for ATmega328P

1.2 Block Diagram



1.3 Pins

1.3.1 Power Pins

VCC, Gnd - 2.7V to 5.5V

1.3.2 PORTB - PB7:PB0

- Bidirection I/O with internal pull-up resistor(selectable for each bit)
- Tristate when reset
- Depending on the clock selection fuse settings,
  - *PB6* – input of inverting oscillator amplifier and input to internal clock operating circuit
  - *PB7* – output of inverting oscillator amplifier
- If internal calibrated RC oscillator is used as clock source, *PB7* and *PB6* is used as *TOSC2* and *TOSC1* input for Timer/Counter2

1.3.3 PORTC - PC5:PB0

- Bidirection I/O with internal pull-up resistor(selectable for each bit)
- Tristate when reset

1.3.4 PC6/$\overline{RESET}$

- Low level on this pin will gnerate reset, even if no clock running.
- ***RSTDIBL*** fuse == programmed(0) – *PC6* is input pin.
- ***RSTDIBL*** fuse == unprogrammed(1) – *PC6* is reset pin.

1.3.5 PORTD - PD7:PD0

- Bidirection I/O with internal pull-up resistor(selectable for each bit)
- Tristate when reset

1.3.6 AV_{CC}

- Supply voltage pin for A/D converter
- Connected to External Vcc when not used
- Connected to Vcc through LPF when used

1.3.7 AREF

Analog reference pin of A/D Converter

1.3.8 ADC7:ADC6

Analog input to ADC(10bit ADC)

1.4 Modes

1.4.1 Idle Mode

Stops the CPU while allowing SRAM, TImer/Counters, USART, 2-wire serial interface, SPI port and interrupt system to continue functioning.

1.4.2 Power-Down Mode

Saves the register contents but freezes the oscillator, disabling all other chip functions untill next interrupt or hardware reset.

1.4.3 Power-Save Mode

The asynchonous timer continues to run, allowing user to maintain timer base while reset of devices is sleeping.

1.4.4 ADC Noise reduction mode

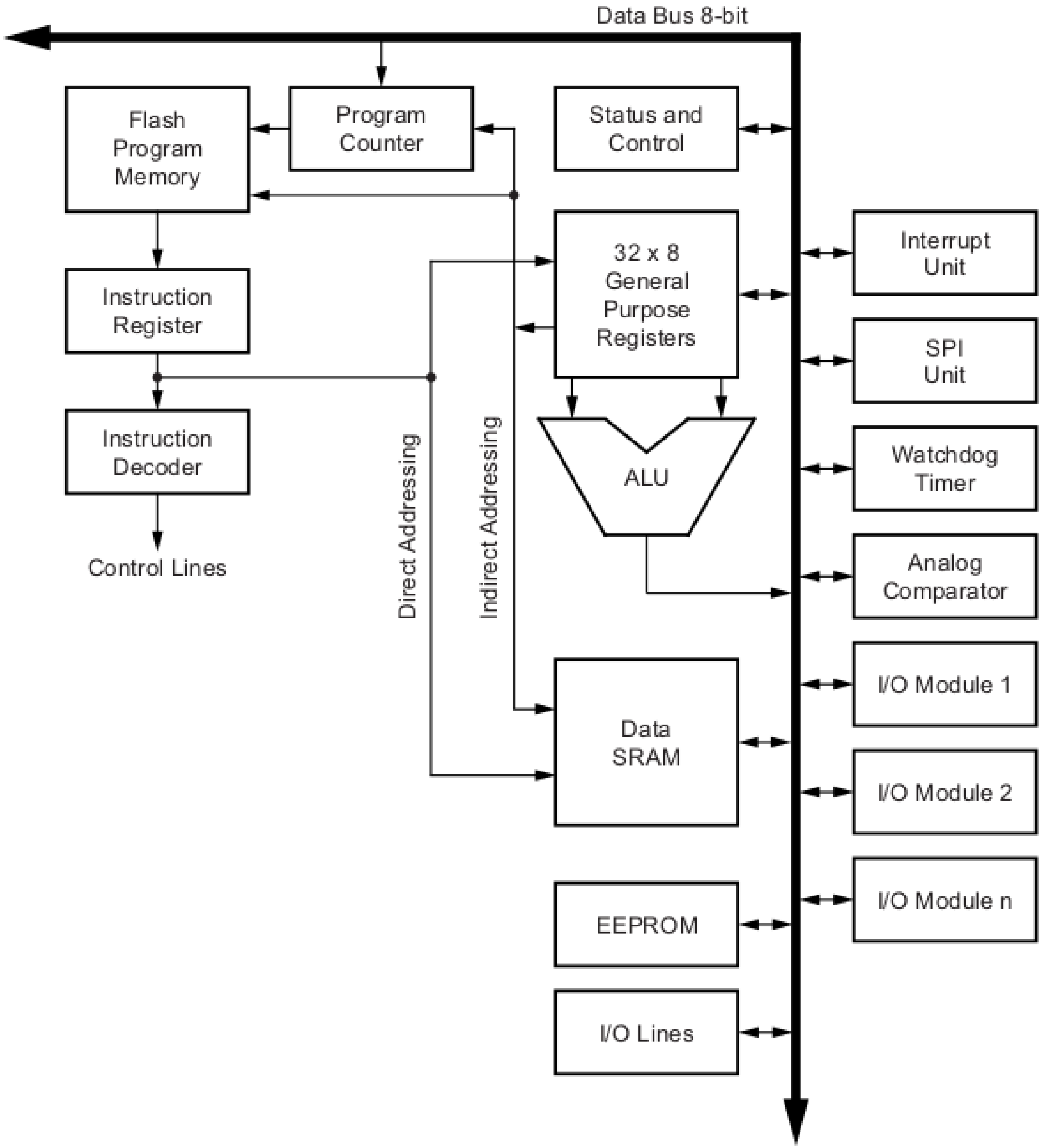
Stops CPU and all I/O modules except asynchonous timer and ADC to minimize switching noise during ADC conversions.

1.4.5 Standby Mode

The crystall/oscillator is running while reset of devices is sleeping. Allows very fast start-up combined with low power consumption.

1.5 AVR CPU Core

The main function of CPU core is access memory, perform caluclations, control peripherals and handle interrupts.



- For performance and parallelism, the AVR uses Harvard Architecture - with seperate memories and buses for program and data.
- Instructions in Program memory are exectued with a single level pipelining.
- The program memory is **In-system Reprogrammable Flash memory**.
- The register file consist of 32 x 8-bit General Purpose Registers with a single clock cycle assess time.
- One ALU operation uses two operatands from register file and store back the result to register file in one clock cycle.
- Six 32-bit register combine to form the X-, Y- and Z- registers which help in 16-bit indirect address register pointer for data space.
- One of these pointers acts as address pointer for look-up tables in Flash Program Memory.
- Program memory adress cotains 16-bit or 32-bit Instructions.
- Program Flash memory space is divided into two sections - each section have dedicated lock bits for read/write protection.
  - Boot Program section
  - Application Program section
- I/O memory space contains 64 addresses for CPU peripheral functions as control register, SPI and Other I/O functionscan be accessed directly or through register file from 0x20 - 0x5F.
- Has extended I/O space from 0x60 - 0xFF in SRAM.

1.5.1 Reset and Interrupt vectors

- Interrupts and reset vectors have seperate program vecotr in program memory space.
- Interrupts maye be disbaled when boot lock bits ***BLB02*** or ***BLB12*** are programmed.
- Lowest ddresses in program memory space are reset and interrupt vectors.
- The lower the addess the higher the priority.
- RESET has the highest followed by INT0(the external interrupt request 0).
- The interrupt vectors can be moved to start of boot flash section by setting ***IVSEL*** bit of **MCUCR** (MCU control register).
- THe reset can be moved to start of boot flash sectio by programming the ***BOOTRST*** fuse.

1.5.2 Interrupt Handling

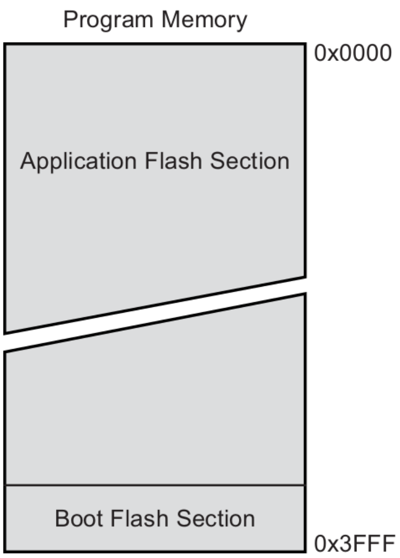
- The ***I-bit*** (global interrupt enable bit) of **Status register** must be enabled.
- When a interrupt occurs, ***I-bit*** (global interrupt enable bit) is cleared and all interrupts are disabled.
- The user can write logic one to ***I-bit*** to enable nested interrupts.
- The ***I-bit*** is automatically set when returning from interrupt Instructions.

1.6 AVR Memories

Two main memory spaces - Data memory and Program memory space and a EEPROM memory for data storage.

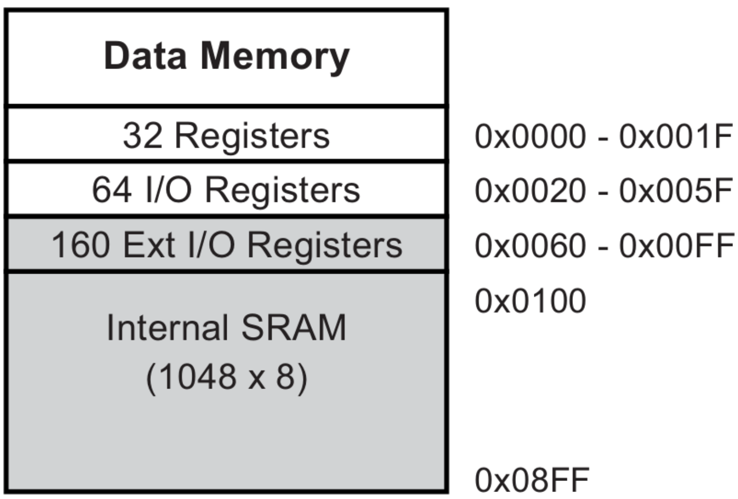
1.6.1 In-System Reprogrammable Flash Program Memory

- 32 KBytes on-chip in-system reprogrammable flash memory for program space.
- Since, the Instructions are all 16-bit or 32-bit wide, the flash(program space) is organized as 16K x 16.
- Endurance of atleast 10,000 write/erase cycle.
- For software security, Flash program memory space is divided into
  - Boot Loaded section
  - Application Program section
- The Program Counter is bits wide and thus can address 16K program memory location.



1.6.2 SRAM Data Memory

- The ATmega328P is a complex microcontroller with more peripheral units than can be supported within the 64 locations reserved in the opcode for the IN and OUT instructions.
- For the extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.



- The lower 2303(0x08FF) data memory locations addresses both the register files, the I/O memory, extended I/O memory and the internal data SRAM.
  - The first 32 location addresses the register file.
  - The next 64 location addresses the standard I/O memory.
  - The following 160 location address the extended I"O memory.
  - The last 2048 location address the internal data SRAM.

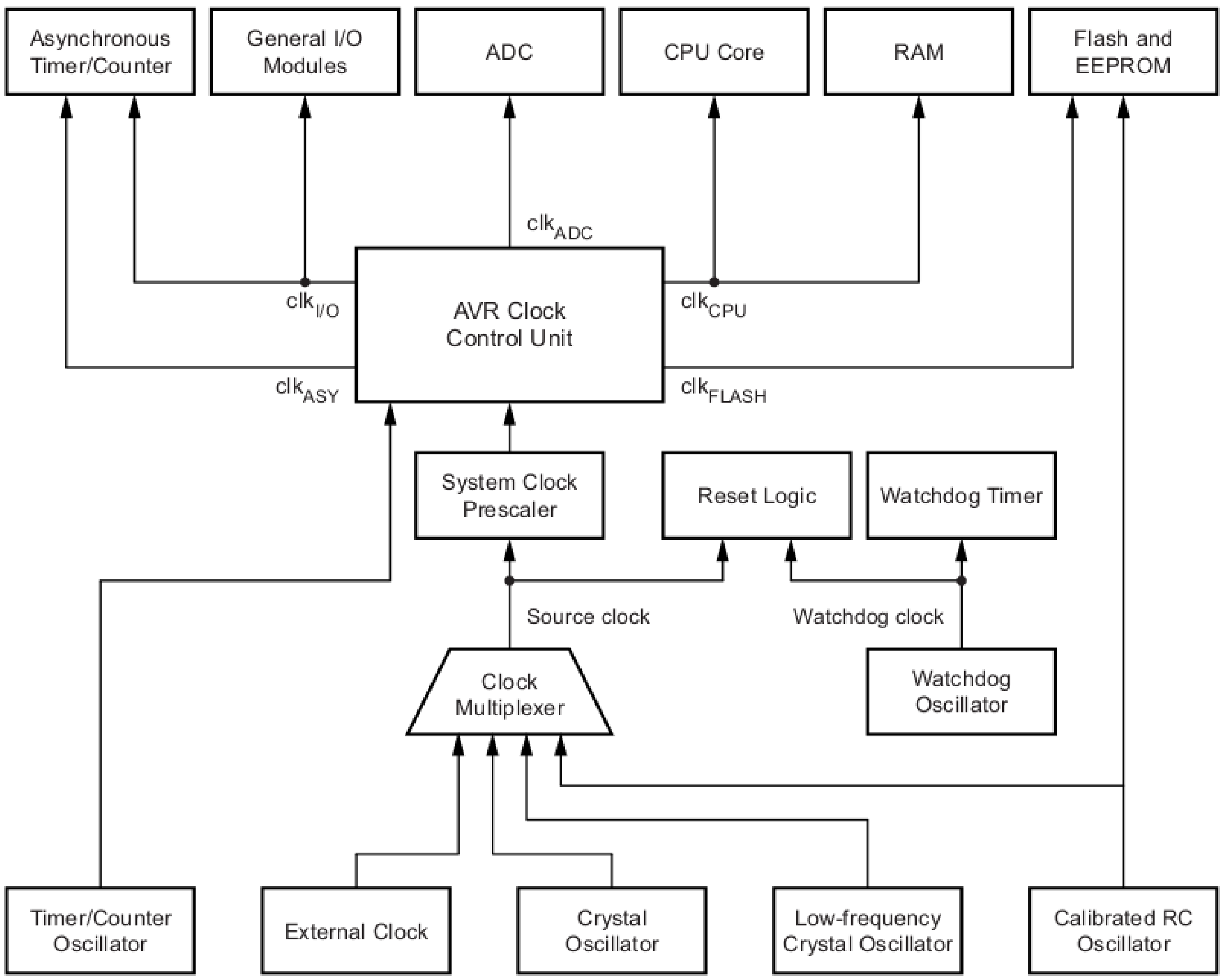
1.6.3 EEPROM Data Memory

- 1 K Byte of data EEPROM memory.
- Organized as seperate data space.
- Endurance of atleast 100,000 write/erase cycle.
- EEPROM are accessible in I/O space.
- Specific Write procedure is followed.

1.6.4 I/O Memory

- I/O and peripherals are placed in the I/O spaces.
- All I/O locations are accesed by LD/LDS/LDD and ST/STS/STD instructions.
- The I/O registers withing 0x00 - 0x1F are directly bit-accessible using SBI and CBI Instructions.

1.7 System Clock and Clock Options



1.7.1 Clock Systems

CPU Clock

- clk_{CPU} is routed to all parts of AVR core.
- General purpose register file, Status register and data memory holding stack pointer.
- Halting CPU clock will inhibts the core from perfrorming general operations and caluclations.

I/O Clock

- $clk_{I/O}$ is used in I/O modules like Timers/Counter, SPI, USART, etc.
- For external interrupt module also but some external interrupts are detected by asynchonous logic and can be used even when I/O clock is halted.

Flash Clock

- clk_{FLASH} controls operation of flash interface.

Asynchronous Timer Clock

- clk_{ASY} allows asynchonous Timer/Counter to be clocked directly from external clock or an external 32 kHz clock crystall.
- This clock allows using Timer/COunter as real-time counter even when device is in sleep mode.

ADC Clock

- $clk_{ADC} haddedicatedclockdomain$
- Gives more accurate ADC conversion result

1.7.2 Clock Sources

Selectable clock sources using flash fuse bits.

CKSEL[3:0]	**Device Clocking Option**
1111 - 1000	Low power crystall oscillator
0111 - 0110	Full swing crystal oscillator
0101 - 0100	Low frequency crystal oscillator
0011	Internal 128kHz RC oscillator
0010	Calibrated internal RC oscillator
0000	External clock

For fuses, "1" denotes unprogrammed and "0" denotes programmed.

Default Clock Source

- Devices is shipped with interface RC oscillator at 8.0MHz with fuse $\boldsymbol{CKDIV8}$ programmed meaning $----->$ the internal oscillator produces a 8.0 Mhz clock but due to $\boldsymbol{CKDIV8}$ being programmed the system clock gets $\frac{8.0MHz}{8} = 1MHz$.
- The startup time is set to maximum and time-out period enabled.
- Default configuration $--->$ $\boldsymbol{CKSEL} = 0010$; $\boldsymbol{SUT} = 10$; $\boldsymbol{CKDIV8} = 0$.

Clock Start Sequence

- Clock Source needs a sufficient V_{CC} and minimum number of oscillating cycles before stablizing.
- To ensure sufficient V_{CC}, the device issues an internal reset with time-out delay (t_{TOUT}).
- The number of cyles in the dealy is set by $\boldsymbol{SUTx}$ bits and $\boldsymbol{CKSELx}$ fuse bits.
- The main purpose of dealy is to keep AVR in reset until it is supplied with minimal V_{CC}.
- The start-up sequence for the clock includes both the time-out delay and the start-up time when the device starts up from reset.

Clock Output Buffer

- device can output system clock on the *CLKO* pin.
- enabled by *CKOUT* fuse.
- any clock source can be used to output from this pin.

TIMER/COUNTER OSCILLATOR

- uses the same crystal oscillator for low-frequency oscillator and Timer/Counter oscillator.
- Since, It shares the Timer/Counter oscillator pins *TOSC1* and *TOSC2* pins with *XTAL1* and *XTAL2*, the system clock must be four times the oscillator and so the Timer/Counter oscillator can only be used when the calibrated internal RC oscillator is selected as system clock source.

Low Power Crystall OSscillator

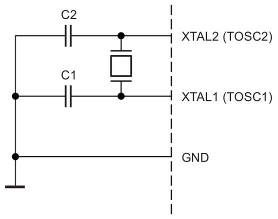


Figure 1.1: VGA Connector

CKSEL[3:1]	Frequency Range (MHz)
100	0.4 to 0.9(only Ceramic resonators)
101	0.9 to 3.0
110	3.0 to 8.0
111	8.0 to 16.0

- *XTAL1* and *XTAL2* are inputs and outpus of an inverting amplifier which can be configured as on-chip oscillator.
- Either Quartz Crystall or Ceramic resonator can be used.
- Crystal Oscillator is a low power oscillator with reduced voltage swing on the XTAL2 output.
- Not capable of driving other clock inputs.
- C1 and C2 should be of the same values – 12pF to 22pF.
- The ***CKSEL[0]*** fuse together with the ***SUT[1:0]*** fuses select the start-up times as shown in Table below.

Oscillator Source / Power Conditions	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	CKSEL0	SUT1..0
Ceramic resonator, fast rising power	258CK	14CK + 4.1ms(1)	0	00
Ceramic resonator, slowly rising power	258CK	14CK + 65ms(1)	0	01
Ceramic resonator, BOD enabled	1KCK	14CK(2)	0	10
Ceramic resonator, fast rising power	1KCK	14CK + 4.1ms(2)	0	11
Ceramic resonator, slowly rising power	1KCK	14CK + 65ms(2)	1	00
Crystal oscillator, BOD enabled	16KCK	14CK	1	01
Crystal oscillator, fast rising power	16KCK	14CK + 4.1ms	1	10
Crystal oscillator, slowly rising power	16KCK	14CK + 65ms	1	11

Full Swing Crystal Oscillator

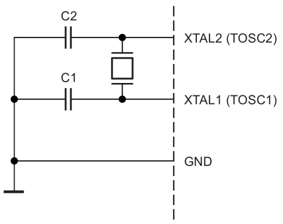


Figure 1.2: VGA Connector

CKSEL[3:1]	Frequency Range (MHz)
011	0.4 to 16.0

- *XTAL1* and *XTAL2* are inputs and outpus of an inverting amplifier which can be configured as on-chip oscillator.

- Either Quartz Crystall or Ceramic resonator can be used.
- Full-Swing with rail-to-rail swing on the XTAL2 output.
- Can drive other clock input
- Power consumption is more than Low power crystal oscillator
- Needs V_{CC} = 2.7 to 5.5V
- C1 and C2 should be of the same values – 12pF to 22pF.
- The ***CKSEL[0]*** fuse together with the ***SUT[1:0]*** fuses select the start-up times as shown in Table below.

Oscillator Source / Power Conditions	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	CKSEL0	SUT1..0
Ceramic resonator, fast rising power	258CK	14CK + 4.1ms(1)	0	00
Ceramic resonator, slowly rising power	258CK	14CK + 65ms(1)	0	01
Ceramic resonator, BOD enabled	1KCK	14CK(2)	0	10
Ceramic resonator, fast rising power	1KCK	14CK + 4.1ms(2)	0	11
Ceramic resonator, slowly rising power	1KCK	14CK + 65ms(2)	1	00
Crystal oscillator, BOD enabled	16KCK	14CK	1	01
Crystal oscillator, fast rising power	16KCK	14CK + 4.1ms	1	10
Crystal oscillator, slowly rising power	16KCK	14CK + 65ms	1	11

Low Frequency Crystal Oscillator

- To use with 32.765kHz watch crystal
- Crystal Cap(CL) – 6.5,9.0 and 12.5pF
- ***CKSEL[3:0]*** == 0101.
- The Start-up Times for the Low-frequency Crystal O scillator Clock Selection

SUT1..0	Additional Delay from Reset (V_{CC} = 5.0V)	Recommended Usage
00	4CK	Fast rising power or BOD enabled
01	4CK + 4.1ms	Slowly rising power
10	4CK + 65ms	Stable frequency at start-up
11	Reserved	

Calibrated Internal RC Oscillator

- 8.0MHz clock
- Voltage and temperature dependent
- Calibration is done in **OSCCAL**.
- Default mode shipeed with CKDIV8 prescalar programmed to prescale causing the system clock to be 1.0MHz.
- ***CKSEL[3:0]*** == 0010.

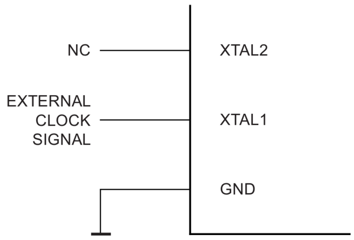
Power Conditions	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	SUT1..0
BOD enabled	6CK	14CK[1]	00
Fast rising power	6CK	14CK + 4.1ms	01
Slowly rising power	6CK	14CK + 65ms[2]	10
Reserved			11

128kHz Internal Oscillator

- low power oscillator with 128kHz frequency
- ***CKSEL[3:0]*** == 0010.

Power Conditions	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	SUT1..0
BOD enabled	6CK	14CK[1]	00
Fast rising power	6CK	14CK + 4.1ms	01
Slowly rising power	6CK	14CK + 65ms[2]	10
Reserved			11

External Clock



- XTAL1 must be connected to external source.
- 0 - 16 MHz frequency.
- ***CKSEL*** == 0000.
- Start-up times are determined by the SUT fuses as

Power Conditions	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	SUT1..0
BOD enabled	6CK	14CK	00
Fast rising power	6CK	14CK + 4.1ms	01
Slowly rising power	6CK	14CK + 65ms	10
Reserved			11

1.7.3 System Clock Prescalar

- The system clock can be divided by setting the **CLKPR** (Clock Prescale Registers) value.
- Used to decrease the system clock frequency and the power consumption when the requirement for processing power is low.
- Affects the clk_{SYS}, clk_{IO}, clk_{ADC}, clk_{CPU} and clk_{FLASH}.
- A special write procedure is followed to change ***CLKPS*** bits:
  (i) Write the clock prescaler change enable (***CLKPCE***) bit to one and all other bits in **CLKPR** register to zero.
  (ii) Within four cycles, write the desired value to ***CLKPS*** bit while writing a zero to ***CLKPCE***.
  (iii) Interrupt must be disabled.

Register Description

OSCCAL – Oscillator Calibration Register

7	6	5	4	3	2	1	0
CAL7	CAL6	CAL5	CAL4	CAL3	CAL2	CAL1	CAL0

- The oscillator calibration register is used to trim the calibrated internal RC oscillator to remove process variations from the oscillator frequency.
- A pre-programmed calibration value is automatically written to this register during chip reset.
- The application software can write this register to change the oscillator frequency.
- If EEPROM is to be used, shouldn't do calibration for more than 8.8 MHz.
- ***CAL7*** bit detected range of operation of oscillator. Setting zeros gives the Lowest requency range, setting this bit to 1 gives the highest frequency range.
- The ***CAL[6:0]*** bits are used to tune the frequency within the selected range. A setting of 0x00 gives the lowest frequency in that range, and a setting of 0x7F gives the highest frequency in the range.

CLKPR – Clock Prescale Register

7	6	5	4	3	2	1	0
CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0

- ***CLKPCE*** - Cloc k Prescaler Change Enable must be written to logic one to enable change of the ***CLKPS*** bits.
- The ***CLKPCE*** bit is only updated when the other bits in CLKPR are simultaneously written to zero.

CLKPS[3:0]	**Clock Division Facter**
0000	1
0001	2
0010	4
0011	8
0100	16
0101	32
0110	64
0111	128
1000	256

- ***CLKPS[3:0]*** - Clock Prescaler Select Bits define the division factor between the selected clock source and the internal system clock.
- The ***CKDIV8*** fuse determines the initial value of the ***CLKPS*** bits.
  - If ***CKDIV8*** is unprogrammed, the ***CLKPS*** bits will be reset to 0000.
  - If ***CKDIV8*** is programmed, ***CLKPS*** bits are reset to "0011", giving a division factor of 8 at start up.
- Note that any value can be written to the ***CCLKPS*** bits regardless of the ***CCKDIV8*** fuse setting.

1.8 Power Management and Sleep modes

- Sleep modes enable the application to shut down unused modules in the MCU, thereby saving power.
- When enabled, the brown-out detector (BOD) actively monitors the power supply voltage during the sleep periods.
- To further save power, it is possible to disable the BOD in some sleep modes.

1.8.1 Sleep Modes

- To enter any of the six sleep modes, the ***SE*** bit in **SMCR** register must be written to logic one.
- The ***SM[2:0]*** bits in the **SMCR** register select which sleep mode.
- *SLEEP* instruction must be executed.
- If an enabled interrupt occurs while the MCU is in a sleep mode, the MCU wakes up.
- The MCU is then halted for four cycles in addition to the start-up time, executes the interrupt routine, and resumes execution from the instruction following SLEEP.
- The contents of the register file and SRAM are unaltered when the device wakes up from sleep.
- If a reset occurs during sleep mode, the MCU wakes up and executes from the reset vector.
- The Active Clock Domains and Wake-up Sources in the Different Sleep Modes,

	Active Clock Domains					Oscillators		Wake-up Sources							
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT1, INT0 and Pin Change	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other/O	Software BOD Disable
Idle			X	X	X	X	X(2)	X	X	X	X	X	X	X	
ADC noise Reduction				X	X	X	X(2)	X(3)	X	X(2)	X	X	X		
Power-down								X(3)	X				X		X
Power-save					X		X(2)	X(3)	X	X			X		X
Standby(1)						X		X(3)	X				X		X
Extended Standby					X(2)	X	X(2)	X(3)	X	X			X		X

Idle Mode

- Stops the CPU but allows the SPI, USART, analog comparator, ADC, 2-wire serial interface, Timer/Counters, watchdog, and the interrupt system.
- Halts clk_{CPU} and clk_{FLASH} and allows other clocks.
- Idle mode enables the MCU to wake up from external triggered interrupts as well as internal ones like the timer overflow and USART transmit complete interrupts.

ADC Noise Reduction Mode

- Stops the CPU but allows ADC, the external interrupts, the 2-wire serial interface address watch, Timer/Counter2 and the watchdog.
- Halts $clk_{I/O}$, clk_{CPU} and clk_{FLASH} and allows other clocks.
- Improves the noise environment for ADC, enabling higher resolution measurement.
- ADC Noise Reduction Mode enables the MCU to wake up from external reset, a watchdog system reset, a watchdog interrupt, a brown-out reset, a 2-wire serial interface address match, a Timer/Counter2 interrupt, an SPM/EEPROM ready interrupt, an external level interrupt on INT0 or INT1 or a pin change interrupt.

Power-down Mode

- Stops the external oscillator but allows the external interrupts, the 2-wire serial interface address watch, and the watchdog.
- Halts all clocks and asynchronous modules only.
- Power-down mode enables the MCU to wake up from an external reset, a watchdog system reset, a watchdog interrupt, a brown-out reset, a 2-wire serial interface address match, an external level interrupt on INT0 or INT1, or a pin change interrupt.

Power-save Mode

- Only diffence from Power-down mode is Timer/Counter2 is enabled and it will run.
- Timer overflow or output compare event from Timer/Counter2 can wake up.

Standby Mode

- Selects the external crystal clock option.
- Identical to power-down except oscillator is running.

External Standby Mode

- Selects the external crystal clock option.
- Identical to power-Save except oscillator is running.

Register Description

SMCR – Sleep Mode Control Register

7	6	5	4	3	2	1	0
-	-	-	-	SM2	SM1	SM0	SE

SM[2:0]	**Sleep Mode**
000	Idle
001	ADC Noise Reduction
010	Power-down
011	Power-save
110	Standby
111	External Standby

- ***SE*** bit must be written to logic one just before the SLEEP instruction is executed, to make the MCU enter the sleep mode.

1.8.2 Power Reduction Register

- To stop the clock to individual peripherals to reduce power consumption.
- The current state of the peripheral is frozen and the I/O registers can not be read or written.
- Peripheral should in most cases be disabled before stopping the clock.
- Wake up peripherals can be done by writing zero to bits in **PRR**.

Register Description

PRR – Power Reduction Register

7	6	5	4	3	2	1	0
PRTWI	PRTIM2	PRTIM0	-	PRTIM1	PRSPI	PRUSART0	PRADC

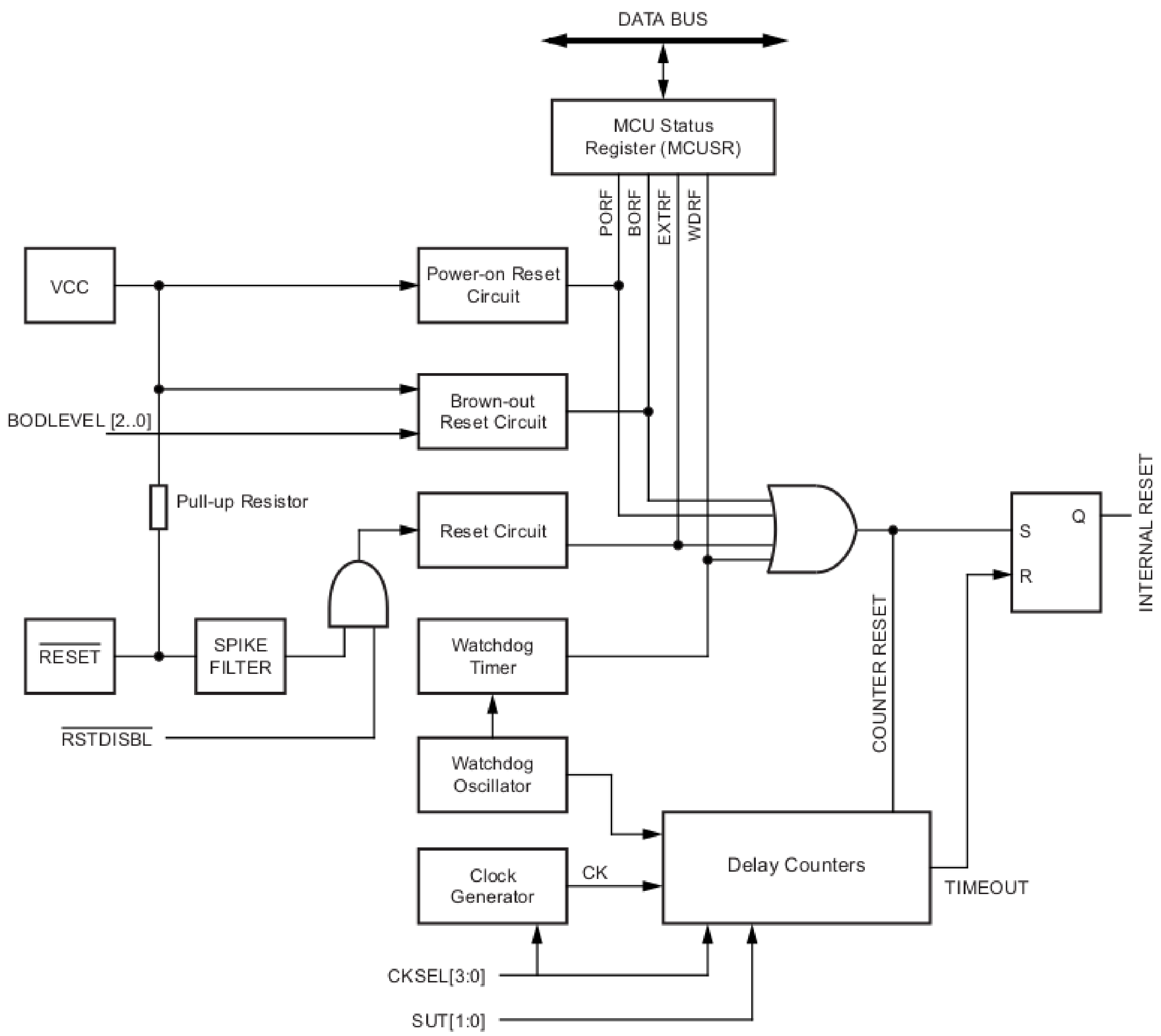
Bits	**Name**
PRTWI	Power Reduction TWI
PRTIM2	Power Reduction Timer/Counter2
PRTIM1	Power Reduction Timer/Counter1
PRTIM0	Power Reduction Timer/Counter0
PRSPI	Power Reduction Serial Peripheral Interface
PRUSART0	Power Reduction USART0
PRADC	Power Reduction ADC

1.8.3 Minimizing Power Consumption

- In general, sleep modes should be used as much as possible.
- ADC should be disabled before entering any sleep mode.
- Analog comparator should be disabled in all sleep modes.
- If the brown-out detector is not needed by the application, this module should be turned off by ***BODLEVEL*** fuses.
- If Internal Voltage Reference is not needded and ADC or analog comparator or BOD is not needed, the Internal Voltage Reference can be disabled.
- If the watchdog timer is not needed in the application, the module should be turned off.
- If On-chip Debug System is not needed, can be disabled by ***DWEN*** fuse.
- For Port pins,
  - No pins drive resistive loads.
  - Input buffers are disabled when I/O clock and ADC clocks are stopped.
  - If the input buffer is enabled and the input signal is left floating or have an analog signal level close to V CC /2, the input buffer will use excessive power.
  - Digital input buffers can be disabled by writing to the digital input disable registers (**DIDR1** and **DIDR0**).

1.9 RESETTING AVR

All I/O registers are set to their intial values and program starts execution form reset vector.



1.9.1 Reset Sources

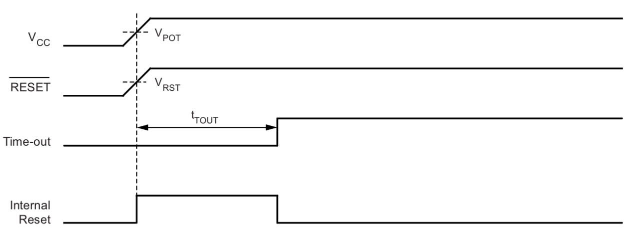
(I) Power-on Reset - MCU resets when supply voltage is below the power-on reset threshold(V_{POT}).

(II) External Reset - MCU resets when low level is present on $\overline{RESET}$ is helow for minimum pulse length.

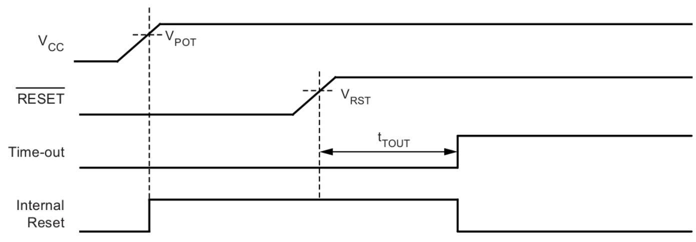
(III) Watchdog System reset - MCU resets when watchdog timer period expires and watchdog system reset mode is enabled.

(IV) Brown-out reset - MCU resets when supply voltage V_{CC} is below brown-out threshold (V_{BOT}) and brown-out detected is enabled.

Power-on Reset



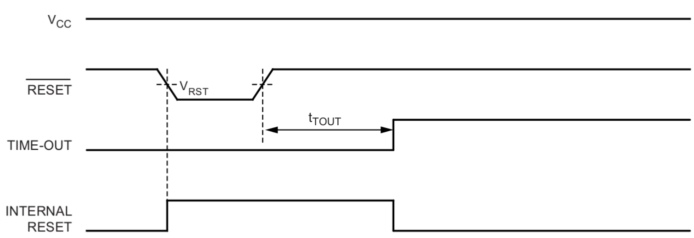
MCU Start-up, $\overline{RESET}$ Tied to V_{CC}



MCU Start-up, $\overline{RESET}$ Extended Externally

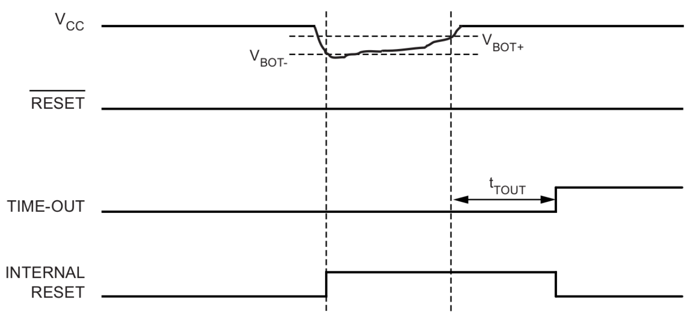
- A power-on reset (POR) pulse is generated by an on-chip detection circuit.
- The POR is activated whenever V_{CC} is below the detection level.
- The POR circuit can be used to trigger the start-up reset, as well as to detect a failure in supply voltage.
- Reaching the power-on reset threshold voltage invokes the delay counter, which determines how long the device is kept in RESET after V_{CC} rise.

External Reset



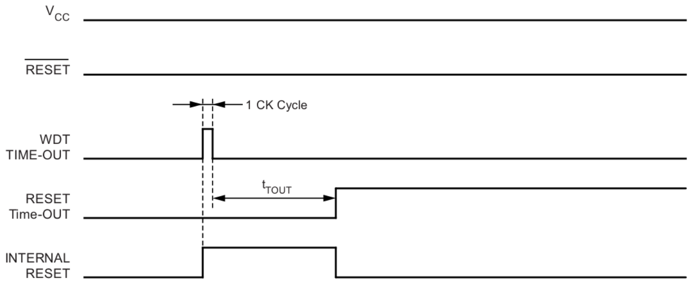
- An external reset is generated by a low level on the $\overline{RESET}$ pin.
- Shorter pulses are not guaranteed to generate a reset.
- When the applied signal reaches the reset threshold voltage – V_{RST} – on its positive edge, the delay counter starts the MCU after the time-out period – t_{OUT} – has expired.
- The external reset can be disabled by the ***RSTDISBL*** fuse.

Brown-out Detection



- On-chip brown-out detection (BOD) circuit for monitoring the V_{CC} level during operation by comparing it to a fixed trigger level.
- The trigger level for the BOD can be selected by the ***BODLEVEL*** fuses.

Watchdog System Reset



- When the watchdog times out, it will generate a short reset pulse of one CK cycle duration.
- On the falling edge of this pulse, the delay timer starts counting the time-out period t_{OUT}.

Compiling and Running

2.1 GCC[4]

- GNU Compiler Collectin - compiler system.
- Supports various language, processor and host operating system.
- AVR GCC - Reffering to GCC targeting AVR.
- AVR GCC translates high-level langues to assembly.
- AVR GCC three language - C, C++, Ada.

2.2 GNU Binutils[4]

Source : Tool Chain overview

- Binary Utilites - contains the GNU assembler(gas), GNU linker(ld), etc.
- avr-as - The assembler.
- avr-ld - The linker.
- avr-objcopy - Copy and translate object files to different format.
- avr-objdump - Display information from object file including disassembly.
- avr-size - List section sizes and total sizes.
- avr-nm - List symbols from objects files.
- avr-strings - List printable strings from files.
- avr-readelf - Display contents of ELF files.
- avr-addr2line - Convert addresses to file and line.

2.3 avr-lib[4]

Open source standard C Libary - standard C Libary and Libary function specifit to AVR.

2.4 Compiler Options[3]

-On – for optimization level; n indicates the optimization level; 0 being the default no optimization;

- *-O0* – reduces compilation time and this is default
- *-O* and *-O1* – the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- *-O2* – Optimize even more than *-O1*
- *-O3* – Optimize even more than *-O2*
- *-Os* – Optimize for size and enables all *-O2* optimization but not expected to increase code size
- *-Ofast* – enables all *-O3* optimzations and disregarads strict standard compliance
- *-Og* – Optimize for debugging experience

2.5 Compilation

- Will create the .obj object binary files.
- Use *avr-gcc* along with following options
  - Optimization option *-On* - use *-Os* generally.
  - Warning option *-Wall* - enables all the warning
  - Debug option *-g* - Produce debugging information.
  - MCU option *-mmcu* - the actual MCU - Supported MCU
  - C file option *-c* - the actual c file.
  - Output file name *-o* - Output file name
- To see the object binary file use *avr-objdump -S fileName.o*

```
avr-gcc -Os -Wall -g -mmcu=atmega8 -c hello.c -o hello.o
```

2.6 Linking

- Link the bianary object file to binary elf file.
- Use *avr-gcc* along with following options
  - Optimization option *-On* - use *-Os* generally.
  - Warning option *-Wall* - enables all the warning
  - Debug option *-g* - Produce debugging information.
  - MCU option *-mmcu* - the actual MCU - Supported MCU
  - .obj file option - the actual .obj file.
  - Output file name *-o* - Output file name
- To see the object binary file use *avr-objdump -S fileName.elf*

```
avr-gcc -Os -Wall -g -mmcu=atmega8 hello.o -o hello.elf
```

2.7 Generating the hex file

- The Intel hex file is what we program into procesosr.
- Use *avr-objcopy* along with following options
  - section Option *-j* – which sections to copy - generally .text and .data section
  - Output format option *-O* - what Output format should be used - eg) ihex
  - The input .elf file
  - The output .hex file

```
avr-objcopy -j .text -j .data -O ihex hello.elf hello.hex
```

2.8 AVRDUDE[2]

2.8.1 Introduction

- AVRDUDE - AVR Downloader UploDEr is a program for downloading and uploading the on-chip memories of Atmel's AVR microcontroller.
- Can program Flash, EEPROM, fuse ,lock bits and signature bytes.
- Can read or write all chip memory types mentioned above.
- Supports varieous programmers from STK500, AVRISP, mkII, JTAG ICE, PPI, serial bit-bang adapters, etc.
- The STK500, JTAG ICE, etc uses serial port to communicate.
- The JTAGICE, AVRISP, USBasp, USBtinyISP uses USB using *libusb*.

2.8.2 Command Line Options

- *-p partno* – the mandatory option which specifies the MCU.
- *-b baudrate* – Specify the Baudrate.
- *-c programmer-id* – Specify the pgorammer used. eg)arduino, avrisp, avrisp2, avrispmkII, avrispv2, jtag1, stk500, stk500v1, stk500v2, usbasp, usbtiny, etc.
- *-C config-file* – Configuration data file.
- *-e* – Causes a chip erase of FLash ROM, EEPROM to 0xff and clears all lock bits.
- *-F* – Override device signature check.
- *-P port* – Specifty the port to be used.
- *-u* – Used if you want to write fuse bits - this cuases disabling the safemode for fuse bits.
- *-t* – uses interactive terminal mode instead of up or downloading files.
- *-v* – Verbose
- *-U memtype:op:filename[:format]*
  - *memtype* – Memory types are
    (i) *calibration* – One or more bytes of RC oscillator calibration data.
    (ii) *eeprom* – The EEPROM.
    (iii) *efuse* – The extended fuse byte
    (iv) *flash* – The flash ROM of device
    (v) *fuse* – The fuse byte in devices with a single fuse byte.
    (vi) *hfuse* – The high fuse byte.
    (vii) *lfuse* – The low fuse byte.
    (viii) *lock* - The lock byte.
    (ix) *signature* - The three device signature byte (device ID).
  - *op* – Operations are
    (i) *r* – Read the specifed device memory and write to specified file.
    (ii) *w* – read the specifed file and write to specifed device memory.
    (iii) *v* – read the specified device memory and the specified file and perform a verify operation .
  - The *filename* can be either a fileName, immediate byte value (in decimal, binary,hexadecimal, etc)
  - *format* is optional and can
    1. *i* - Intel hex
    2. *r* - raw binary
    3. *e* - the elf files
    4. *m* - immediate mode
    5. *d* - decimal
    6. *b* - binary(0b)
    7. *d* - hexadecimal(0x)

2.8.3 Example

Downloading hex file into device Flash

```
avrdude -p atmega8 -b 19200 -c stk500 -p /dev/ttyUSB0 -v -U flash:w:hello.hex:i
```

Uploading Flash from device into file

```
avrdude -p atmega8 -b 19200 -c stk500 -p /dev/ttyUSB0 -v -U flash:r:"./readFashMemory.bin":r
```

Reading Device signature

```
avrdude -p atmega8 -b 19200 -c stk500 -p /dev/ttyUSB0 -v -U signature:r:"deviceSinature.text":h
```

2.8.4 Writing the High fuse

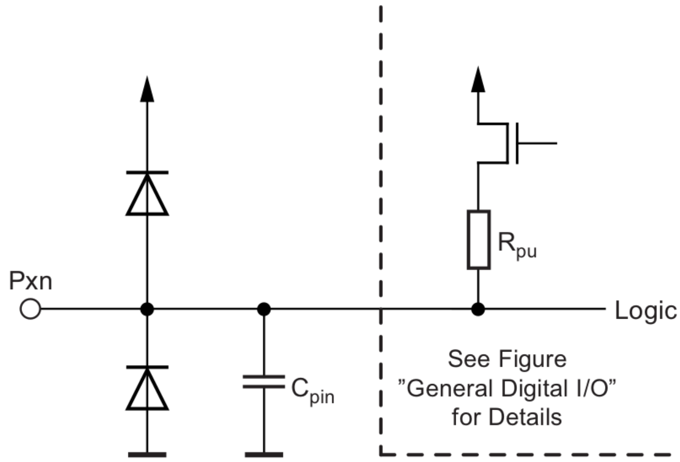
```
avrdude -p atmega8 -b 19200 -c stk500 -p /dev/ttyUSB0 -v -U hfuse:w:0x65:m
```

Also see this[1].

Input/Output Ports

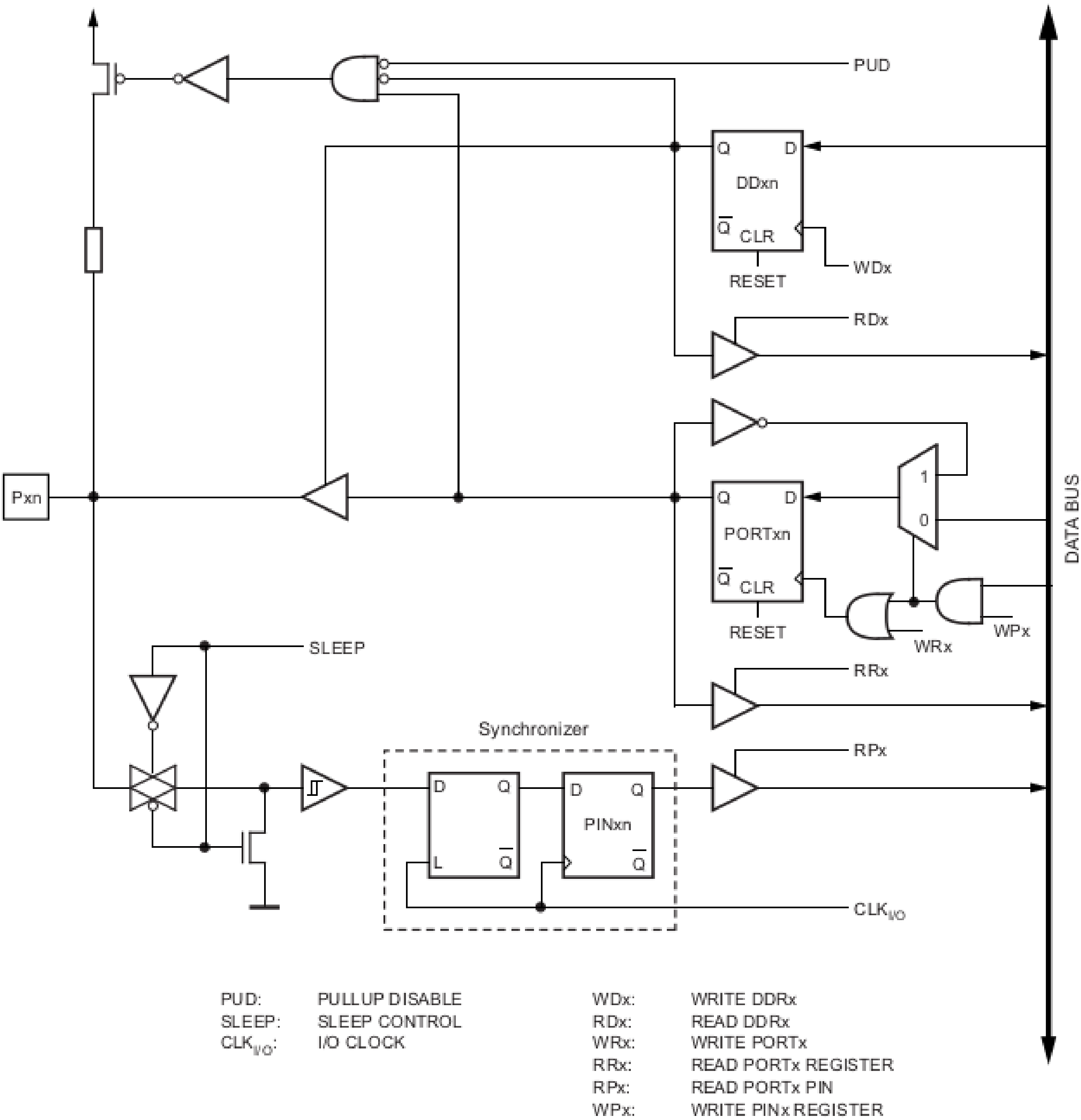
3.1 Introduction

- The directin/drive value/pull-up register of one port pin can be changed without changing the directin/drive value/pull-up register of any other pin - true *read-modify-write*.
- Each output buffer has symmetrical drive characteristics with both high sink and source capability.
- All I/Opins have protection diode to both V_{CC} and Ground.



- Three I/O memory address locations are allocated for each port, one each for the data register – **PORTx**, data direction register – **DDRx**, and the port input pins – **PINx**.
- Most pins are multiplexed with alternative functions.
- Generally, after reset, the port pins are tri-stated.
- Disbaling the ***PUD*** bit in **MCUCR** register disables the pull-up function of all pins.
- Unconnected pins should not float and must be connected to internal pull-up or external pull-up/pull-down registor.

3.2 General Digital I/O



3.2.1 DDR Registers

- It is used to select the direction of a pin.
- DDxn == 1 −− > Pin n of Port x is configured as output.
- DDxn == 0 −− > Pin n of Port x is configured as input.

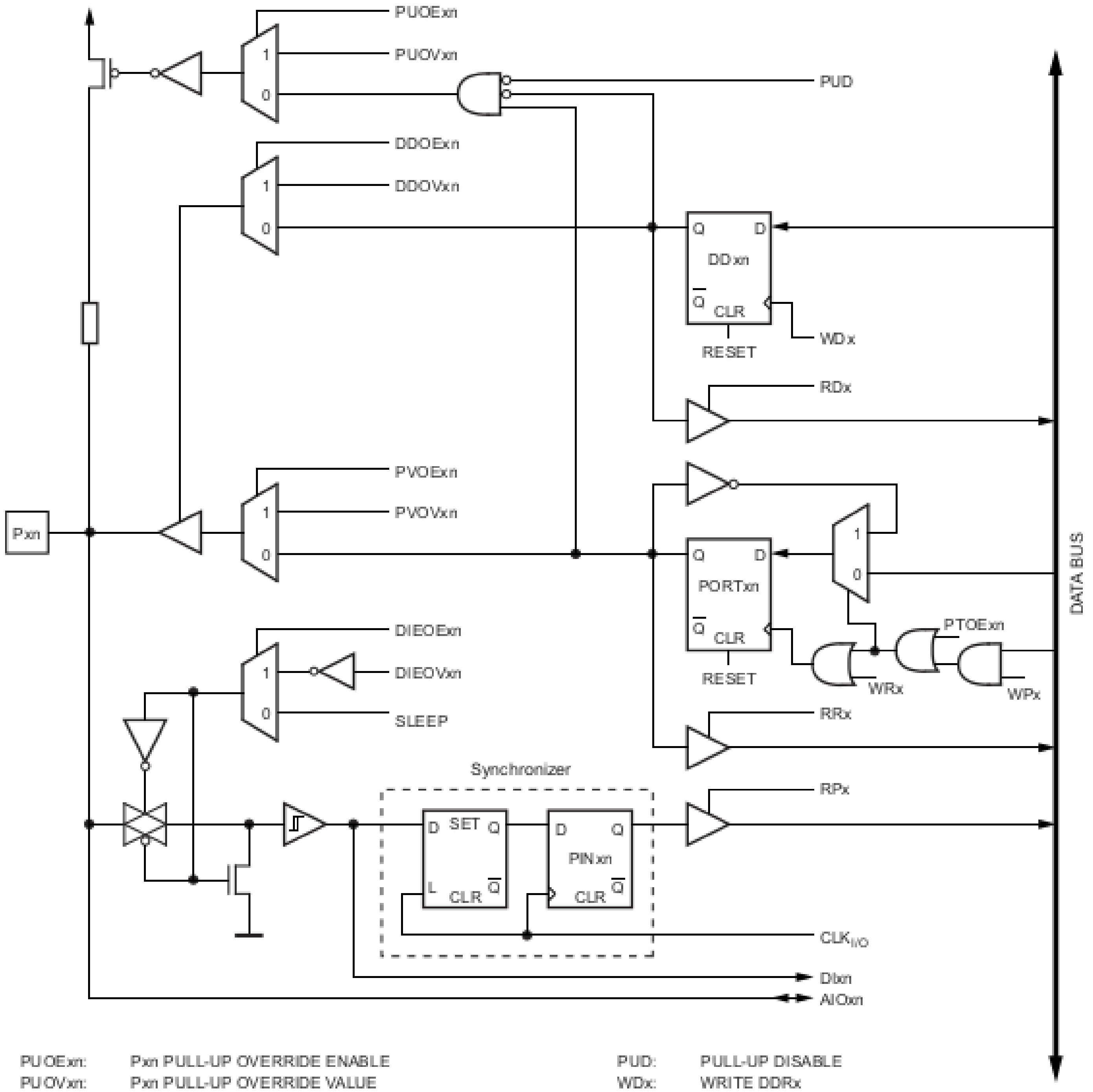
3.2.2 PORT registers

- If the pin is configured as Output - Drive the pin.
  - PORTxn == 1 −− > Pin n of Port x is driven to logic HIGH.
  - PORTxn == 0 −− > Pin n of Port x is driven to logic LOW.
- If the pin is configured as Input - configure pull-up resistor.
  - PORTxn == 1 −− > Pin n of Port x has pull-up resistor activaed.
  - PORTxn == 0 −− > Pin n of Port x has pull-up resistor deactivaed.

3.2.3 PIN Registers

- It is used to read the status of a pin.
- Writing 1 to a PINxn makes the Pin n of Port x toggle.

3.3 Alternate Port Functions



PUOExn:	Pxn PULL-UP OVERRIDE ENABLE	PUD:	PULL-UP DISABLE
PUOVxn:	Pxn PULL-UP OVERRIDE VALUE	WDx:	WRITE DDRx
DDOExn:	Pxn DATA DIRECTION OVERRIDE ENABLE	RDx:	READ DDRx
DDOVxn:	Pxn DATA DIRECTION OVERRIDE VALUE	RRx:	READ PORTx REGISTER
PVOExn:	Pxn PORT VALUE OVERRIDE ENABLE	WRx:	WRITE PORTx
PVOVxn:	Pxn PORT VALUE OVERRIDE VALUE	RPx:	READ PORTx PIN
DIEOExn:	Pxn DIGITAL INPUT ENABLE OVERRIDE ENABLE	WPx:	WRITE PINx
DIEOVxn:	Pxn DIGITAL INPUT ENABLE OVERRIDE VALUE	$CLK_{I/O}$:	I/O CLOCK
SLEEP:	SLEEP CONTROL	DIxn:	DIGITAL INPUT PIN n ON PORTx
PTOExn:	Pxn, PORT TOGGLE OVERRIDE ENABLE	AIOxn:	ANALOG INPUT/OUTPUT PIN n ON PORTx

Signal Name	Full Name	Description
PUOE	Pull-up override enable	If this signal is set, the pull-up enable is controlled by the PUOV signal. If this signal is cleared, the pull-up is enabled when {DDxn, PORTxn, PUD} = 0b010.
PUOV	Pull-up override value	If PUOE is set, the pull-up is enabled/disabled when PUOV is set/cleared, regardless of the setting of the DDxn, PORTxn, and PUD register bits.
DDOE	Data direction override enable	If this signal is set, the output driver enable is controlled by the DDOV signal. If this signal is cleared, the output driver is enabled by the DDxn register bit.
DDOV	Data direction override value	If DDOE is set, the output driver is enabled/disabled when DDOV is set/cleared, regardless of the setting of the DDxn register bit.
PVOE	Port value override enable	If this signal is set and the output driver is enabled, the port value is controlled by the PVOV signal. If PVOE is cleared, and the output driver is enabled, the port value is controlled by the PORTxn register bit.
PVOV	Port value override value	If PVOE is set, the port value is set to PVOV, regardless of the setting of the PORTxn register bit.
PTOE	Port toggle override enable	If PTOE is set, the PORTxn register bit is inverted.
DIEOE	Digital input enable override enable	If this bit is set, the digital input enable is controlled by the DIEOV signal. If this signal is cleared, the digital input enable is determined by MCU state (normal mode, sleep mode).
DIEOV	Digital input enable override value	If DIEOE is set, the digital input is enabled/disabled when DIEOV is set/cleared, regardless of the MCU state (normal mode, sleep mode).
DI	Digital input	This is the digital input to alternate functions. In the figure, the signal is connected to the output of the schmitt trigger but before the synchronizer. Unless the digital input is used as a clock source, the module with the alternate function will use its own synchronizer.
AIO	Analog input/output	This is the analog input/output to/from alternate functions. The signal is connected directly to the pad, and can be used bi-directionally.

Interrupts

4.1 Introduction

- Each interrupt vector occupies two instruction Word(2x16bit) in Atmega328p.
- The complete placement of Reset and Interrupt Vectors in ATmega328P

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset, brown-out reset and watchdog system reset
2	0x002	INT0	External interrupt request 0
3	0x0004	INT1	External interrupt request 1
4	0x0006	PCINT0	Pin change interrupt request 0
5	0x0008	PCINT1	Pin change interrupt request 1
6	0x000A	PCINT2	Pin change interrupt request 2
7	0x000C	WDT	Watchdog time-out interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 compare match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 compare match B
10	0x0012	TIMER2 OVF	Timer/Counter2 overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 capture event
12	0x0016	TIMER1 COMPA	Timer/Counter1 compare match A
13	0x0018	TIMER1 COMPB	Timer/Counter1 compare match B
14	0x001A	TIMER1 OVF	Timer/Counter1 overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 compare match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 compare match B
17	0x0020	TIMER0 OVF	Timer/Counter0 overflow
18	0x0022	SPI, STC	SPI serial transfer complete
19	0x0024	USART, RX	USART Rx complete
20	0x0026	USART, UDRE	USART, data register empty
21	0x0028	USART, TX	USART, Tx complete
22	0x002A	ADC	ADC conversion complete
23	0x002C	EE READY	EEPROM ready
24	0x002E	ANALOG COMP	Analog comparator
25	0x0030	TWI	2-wire serial interface

- The location of reset vector is affected by ***BOOTRST*** fuse.
- The Interrupt vector start address is affected by ***IVSEL*** bit in **MCUCR** register.
- The reset and interrupt Vector placement is shown below as

BOOTRST	**Reset Address**
0	Boot reset address
1	0x0000

IVSEL	**Interrupt Vectors Start Address**
0	0x0002
1	Boot reset address + 0x0002

4.1.1 Register Description

MCUCR – MCU Control Register

7	6	5	4	3	2	1	0
-	BODS	BODSE	PUD	-	-	IVSEL	IVCE

- When ***IVSEL*** bit is cleared, the interrupt vectors are placed at the start of the flash memory - the application section.
  - When the ***BLB12*** is programmed, interrupts are disabled while executing from boot loader section.
- When ***IVSEL*** bit is set, the interrupt vectors are moved to the beginning of the boot loader section of the flash.
  - When the ***BLB02*** is programmed, interrupts are disabled while executing from application section.
- The actual address of the start of the boot flash section is determined by the ***BOOTSZ*** fuses.
- Writing ***IVSEL*** bit is done by
  (a) Write interrupt vector change enable ***IVCE*** bit to one.
  (b) Write desired value to ***IVSEL*** while Writing zeros to ***IVCE***.
- ***IVCE*** is cleared by hardware.

4.2 External Interrupts

- Triggered by *INT0*, *INT1* and *PCING[23:0]* pins.
- Pin changed interrupt *PCI2* will be triggered if any *PCINT[23:16]* toggles based on the **PCMSK2** register.
- Pin changed interrupt *PCI1* will be triggered if any *PCINT[14:8]* toggles based on the **PCMSK1** register.
- Pin changed interrupt *PCI0* will be triggered if any *PCINT[7:0]* toggles based on the **PCMSK0** register.
- Due to asynchronous nature of Pin change interrupts, *PCINT[23:0]* can be used to wake up.
- The *INT0* and *INT1* can be triggered by falling, rising or low level choosen by **EICRA** - External Interrupt Control Register.
- Due to asynchronous nature of External interrupts in low level interrupt, *INT0* and *INT1* can be used to wake up.

4.2.1 Register Description

EICRA - External Interrupt Control Register A

7	6	5	4	3	2	1	0
-	-	-	-	ISC11	ISC10	ISC01	ISC00

- ***ICS11:ICS10*** - Interrupt Sense Control 1 Bit 1 and Bit 0
- ***ICS01:ICS00*** - Interrupt Sense Control 0 Bit 1 and Bit 0

ICS11:ICS10	**Description**
00	Low level of ***INT1*** generates interrupt
01	Any logic change on ***INT1*** generates interrupt
10	The falling edge of ***INT1*** generates an interrupt request.
11	The rising edge of ***INT1*** generates an interrupt request.

ICS01:ICS00	**Description**
00	Low level of ***INT0*** generates interrupt
01	Any logic change on ***INT0*** generates interrupt
10	The falling edge of ***INT0*** generates an interrupt request.
11	The rising edge of ***INT0*** generates an interrupt request.

EIMSK – External Interrupt Mask Register

7	6	5	4	3	2	1	0
-	-	-	-	-	-	INT1	INT0

- Enable the corresponding External Interrupt Request Enable bits (***INT1*** or ***INT0***) and ***I-biy*** of status Register **SREG** to enable the External interrupt.

EIFR – External Interrupt Flag Register

7	6	5	4	3	2	1	0
-	-	-	-	-	-	INTF1	INTF0

- When interrupt occurs on the External interrupt pins *INT0* and *INT1*, the corresponding External Interrupt Flag bits (***INTF1*** or ***INTF0***) are set.
- The Flag is cleared by by writing 1 to it in interrupt routine.

PCICR – Pin Change Interrupt Control Register

7	6	5	4	3	2	1	0
-	-	-	-	-	PCIE2	PCIE1	PCIE0

- Enable the corresponding Pin Change Interrupt Enable bits (***PCIE2*** or ***PCIE1*** or ***PCIE0***) and ***I-bit*** of status Register **SREG** to enable the pin change interrupt.
- Setting 1 to ***PCIE2*** bit enabled interupt to occur in *PCINT[23:16]* pins based on **PCMSK2** register.
- Setting 1 to ***PCIE1*** bit enabled interupt to occur in *PCINT[14:8]* pins based on **PCMSK1** register.
- Setting 1 to ***PCIE0*** bit enabled interupt to occur in *PCINT[7:0]* pins based on **PCMSK0** register.

PCIFR – Pin Change Interrupt Flag Register

7	6	5	4	3	2	1	0
-	-	-	-	-	PCIF2	PCIF1	PCIF0

- When interrupt occurs on the Pin Change interrupt pins *PCINT[23:16]*, the ***PCIF2*** Pin Change Interrupt Flag 2 bits is set.
- When interrupt occurs on the Pin Change interrupt pins *PCINT[14:8]*, the ***PCIF1*** Pin Change Interrupt Flag 1 bits is set.
- When interrupt occurs on the Pin Change interrupt pins *PCINT[7:0]*, the ***PCIF0*** Pin Change Interrupt Flag 0 bits is set.
- The Flag is cleared by by writing 1 to it in interrupt routine.

PCMSK2 – Pin Change Mask Register 2

7	6	5	4	3	2	1	0
PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16

PCMSK1 – Pin Chage Mask Register 1

7	6	5	4	3	2	1	0
-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8

PCMSK0 – Pin Change Mask Register 0

7	6	5	4	3	2	1	0
PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0

- **PCMSK2**, **PCMSK1** and **PCMSK0** are used to select which pin to be enabled for Pin Change Interrupt.

4.3 Configuring External Interupt

(I) First, the *INT0* or *INT1* pins are configured as Input. (Optional)

(II) Next, the pull-up register may be enabled if needed.

(III) Next, the Interrupt Sense Control Bits are configured for level or edge triggered.

(IV) Finally, the Interrupt are enabled.

(V) Also, Global interrupt is enabled.

(VI) We define the ISR and check the Interrupt Flags if the interrupt occured.

(VII) An example configuration can be seen below.

```
// making PD2 as input for INT0, though not
↪ neeced
DDRD &= ~(1<<2);
// enabling the internal pull-up register for
↪ PD2 for INT0
PORTD |= (1<<2);

// making EICRA's ISC01 and ISC00 as 10 for
↪ falling edge detection at INT0
EICRA |= (1<<ISC01);
EICRA &= ~(1<<ISC00);
// making EIMSK's INT0 as 1 to enable External
↪ Interrput Request for INT0
EIMSK |= (1<<INT0);

// Enabling global Interrupts
sei();
```

```
ISR(INT0_vect)
{
    if((EIFR & (1<<INTF0)) != 0)        //
    ↪ INT0 interrupt as occured
    {
        //toggle Led at pinc 0
        PINC |= (1<<0);
    }
}
```

4.4 Configuring Pin Change Interupt

(I) First, the *PCINT[23:0]* pins are configured as Input. (Optional)

(II) Next, the pull-up register may be enabled if needed.

(III) Next, which PCINT is selected.

(IV) Finally, the Interrupt are enabled.

(V) Also, Global interrupt is enabled.

(VI) We define the ISR and check the Interrupt Flags if the interrupt occured.

(VII) An example configuration can be seen below.

```
// making PD4 as input for PCI20
DDRD &= ~(1<<4);
// enabling the internal pull-up register for
↪ PD4 for PCI20
PORTD |= (1<<4);

// Selecting the PCINT20 for PCI2 intterupt
PCMSK2 |= (1<<PCINT20);
// Enabling the PCI2 interrupt
PCICR |= (1<<PCIE2);

// Enabling global Interrupts
sei();
```

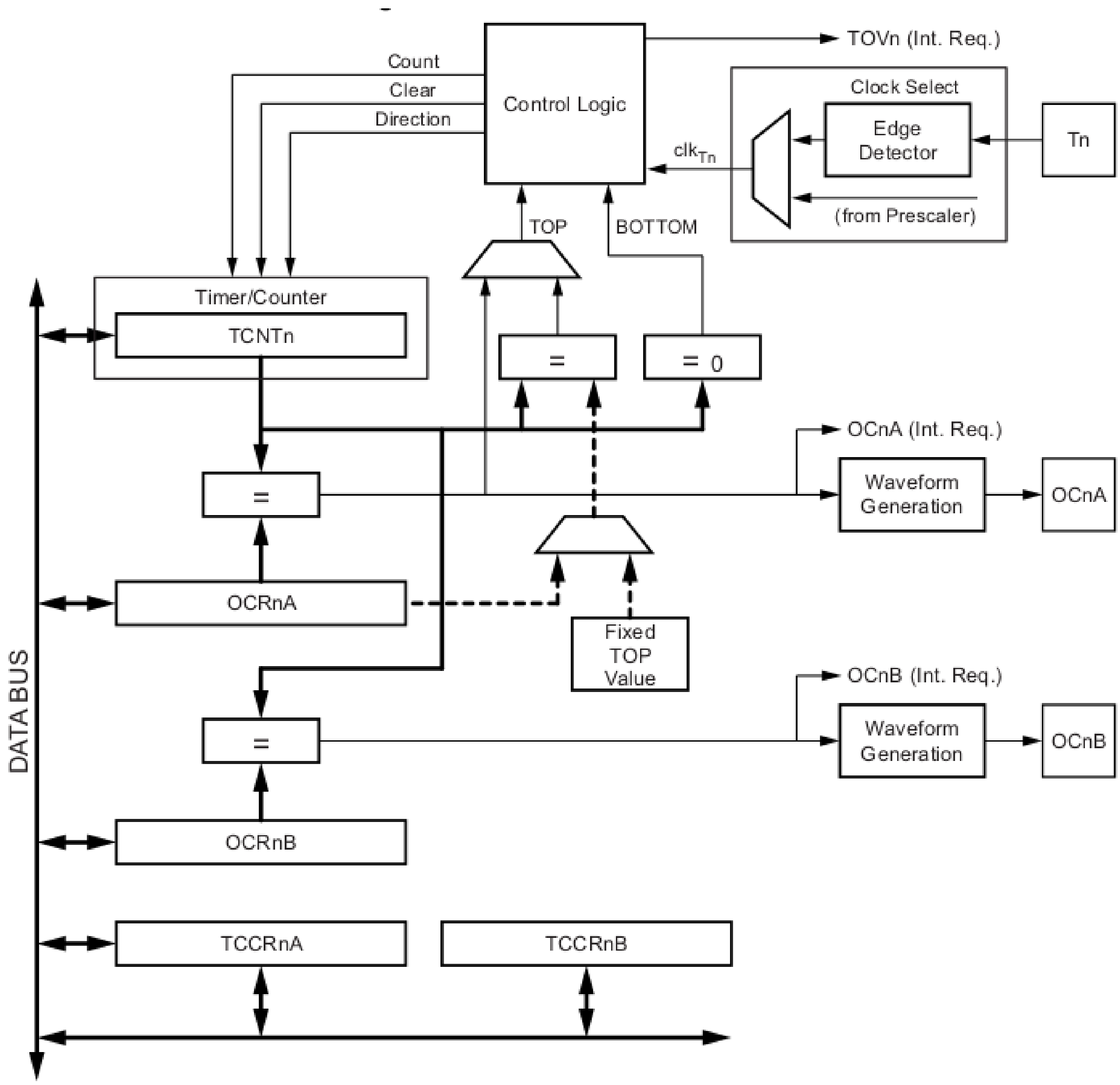
```
ISR(PCINT2_vect)
{
    if((PCIFR & (1<<PCIF2)) != 0)        //
    ↪ PCI2 interrupt as occured
    {
        //toggle Led at pinc 0
        PINC |= (1<<0);

    }
}
```

Timer/Counter 0

5.1 Features

- General purpose 8-bit Timer/Counter module.
- Two independent output compare units.
- Variable PWM.
- Three independent interrupt sources (TOV0, OCF0A, and OCF0B).
- Clear timer on compare match (auto reload)

5.2 Block Diagram



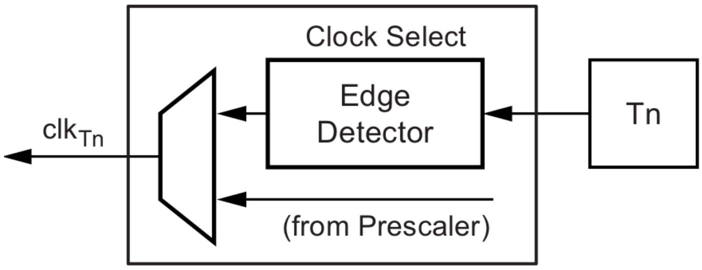
5.3 Terminologies and Registers

Parameter	Description
BOTTOM	counter reaches 0x00
MAX	ounter reaches 0xFF
TOP	counter reaches highest value (depends on mode of operation can be 0xFF, OCR0A).

Register - 8 bit	Name
TCNT0	Timer/Counter0 count value
TCCR0A	Timer/Coutner0 Control Register A
TCCR0B	Timer/Coutner0 Control Register B
OCBR0A	Output compare register A
OCBR0B	Output compare register B
TIFR0	Timer Interrupt Flag Register
TIMSK0	Timer interrupt Mask Register

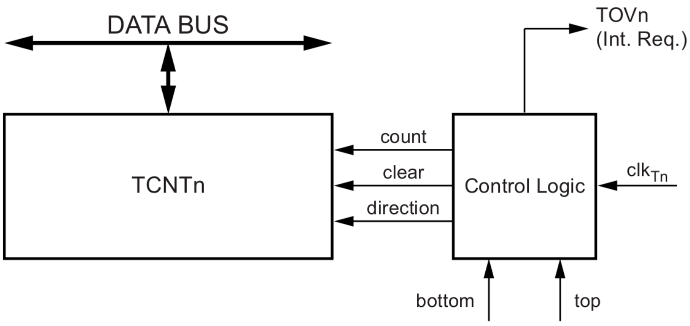
5.4 Timer/Counter0 Units

5.4.1 Clock Source/Select Unit



- The source for the Timer/Counter0 can be external or internal.
- External clock source is from *T0* pin.
- While Internal Clock source can be clocked via a prescalar.
- The output of this unit is the timer clock (clk_{T0}).
- It uses ***CS0[2:0]*** bits in **TCCR0B** register to select the source.

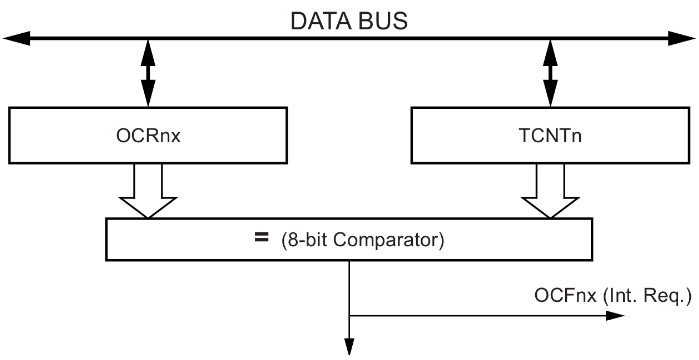
5.4.2 Counter Unit



Signal	Description
count	Increment or decrement **TCNT0** by 1
direction	Select between increment or decrement
clear	Clears **TCNT0** to 0x00
clk_{T0}	Timer/Coutner0 clock
top	Signalize that **TCNT0** has maximum value
bottom	Signalize that **TCNT0** has minimum value(0x00)

- The main part of the 8-bit Timer/Counter is the programmable bi-directional counter.
- Depending the mode of operation the counter is cleared, incremented, or decremented at each timer clock (clk_{T0}).
- Counting sequence is determined by ***WGM0[1:0]*** bits of **TCCR0A** -Timer/Counter0 Control register A and ***WGM02*** bit of **TCCR0B** - Timer/Counter0 Control register B.
- The Timer/Counter0 Overflow flag ***TOV0*** is set and can generate interrupt according to the mode.

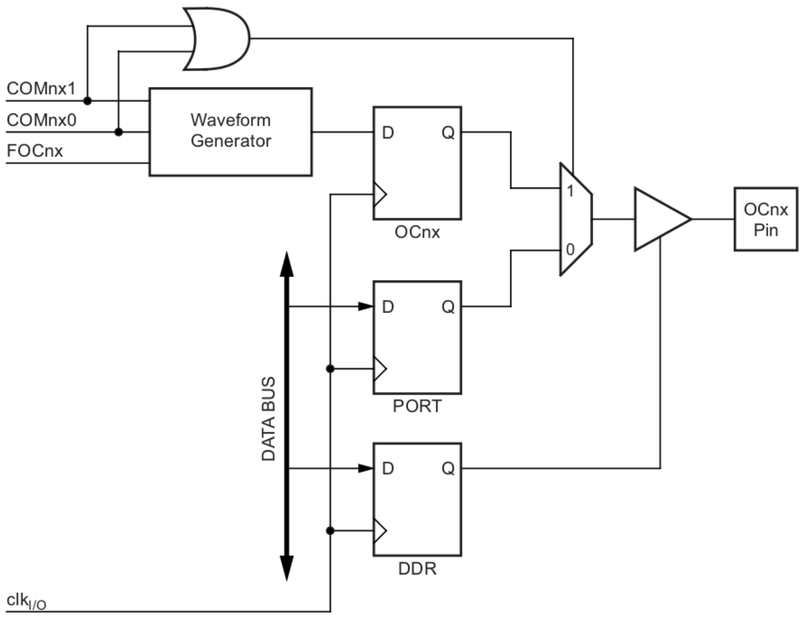
5.4.3 Output Compare Unit



- 8-bit comparator continuously compares **TCNT0** with both **OCR0A** and **OCR0B**.

- When **TCNT0** equals **OCR0A** or **OCR0B**, the comparator signals a match which will set the output compare flag at the next timer clock cycle.
- If interrupts are enabled, then output compare interrupt is generated.
- The waveform generator uses the match signal to generate an output according to operating mode set by the ***WGM0[2:0]*** bits and compare output mode ***COM0x[1:0]*** bits.

5.4.4 Compare Match Output Unit



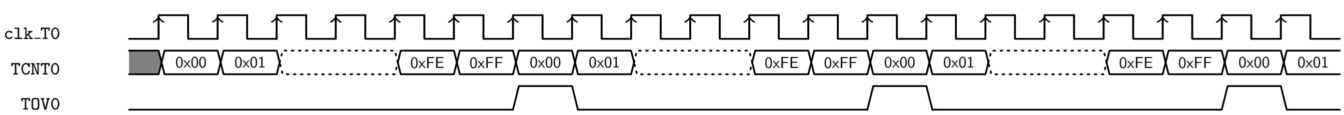
- This unit is used for changing the state of *OC0A* and *OC0B* pins by configuring the ***COM0x[1:0]*** bits.
- But, general I/O port function is overriiden by DDR reigster.

5.5 Modes of Operation

- The mode of operation can be defined by combination of waveform generation mode (***WGM0[2:0]***) and compare output mode(***COM0[1:0]***) bits.
- The waveform generation mode (***WGM0[2:0]***) bits affect the counting sequence.
- For non-PWM mode, ***COM0[1:0]*** bits control if the output should be set, cleared or toggled at a compare match.
- For PWM mode, ***COM0[1:0]*** bits control if the PWM generated should be inverted or non-inverted.

5.5.1 Normal Mode - Non-PWM Mode

- ***WGM0[2:0]*** $-->$ 000.
- Counter counts up and no counter clear.
- Overruns TOP(0XFF) and restarts from BOTTOM(0X00).
- ***TOV0*** Flag is only set when overrun.
- We have to clear ***TOV0*** flag inorder to have next running.
- But, if we use interrupt we don't need to clear it as interrupt automatically clear the ***TOV0*** flag.
- The timing can be seen below.

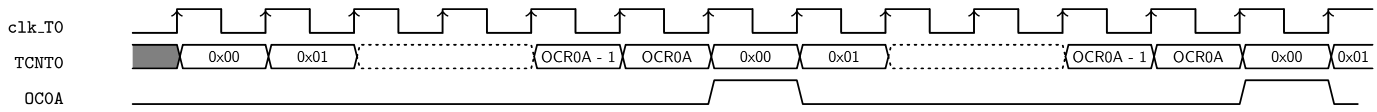


5.5.2 Clear Timer on Compare Match(CTC) Mode - Non-PWM Mode

- ***WGM0[2:0]*** $-->$ 010.
- Counter value clears when **TCNT0** reaches **OCR0A**.
- Interrupt can be generated each time **TCNT0** reaches **OCR0A** register value by ***OCF0A*** flag.
- When ***COM0A[1:0]*** == 01, the *OC0A* pin output can be set to toggle its match between **TCNT0** and **OCR0A** to generate waveform.
- The frequency of the waveform its

$$f_{OC0A} = \frac{f_{clkT0}}{2*N*(1+OCR0A)}$$

- Here N is prescalar factor and can be (1, 8, 64, 256, or 1024).

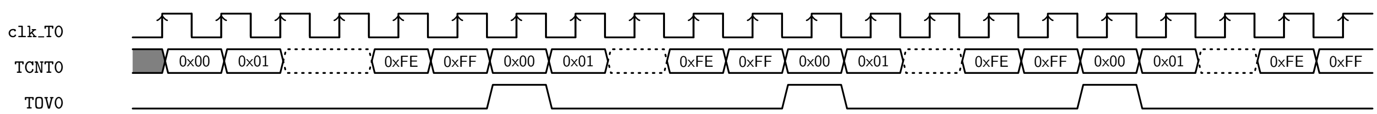


5.5.3 Fast PWM Mode

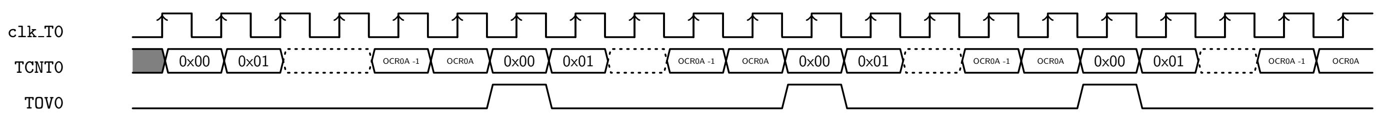
- ***WGM0[2:0]*** $-->$ 011 or 111.
- Power Regulation, Rectification, DAC applications.
- Single slope operations causing high frequency PWM waveform.
- Counter starts from BOTTOM to TOP and then restarts from BOTTOM.
- TOP is defined by
  - TOP == 0xFF if ***WGM0[2:0]*** $-->$ 011
  - TOP == **OCR0A** if ***WGM0[2:0]*** $-->$ 111
- When ***COM0A[1:0]*** == 01, the *OC0A* pin output can be set to toggle its match between **TCNT0** and TOP to generate waveform.
  - The above is possible only when ***WGM02*** bit is set.
  - And only on *OC0A* pin and not on *OC0B* pin.
- In Inverting Compare Mode ***COM0A[1:0]*** == 10 , the *OC0A* or *OC0B* pins is made 1 on compare match between **TCNT0** and TOP and made 0 on reaching BOTTOM.
- In Non-Inverting Compare Mode ***COM0A[1:0]*** == 11 , the *OC0A* or *OC0B* pins is made 0 on compare match between **TCNT0** and TOP and 1 made on reaching BOTTOM.
- The Timer/Counter overflow flag (***TOV0***) is set each time the counter reaches TOP.
- The PWM frequency is given by

$$f_{OC0xPWM} = \frac{f_{clkT0}}{N*256}$$

WGM[2:0] == 011



WGM[2:0] == 011

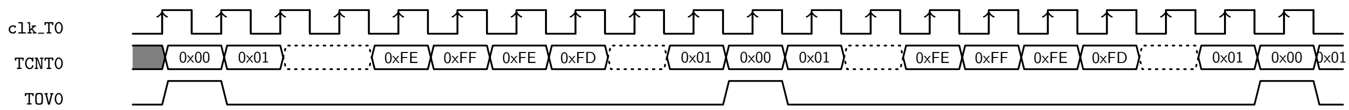


5.5.4 Phase Correct PWM Mode

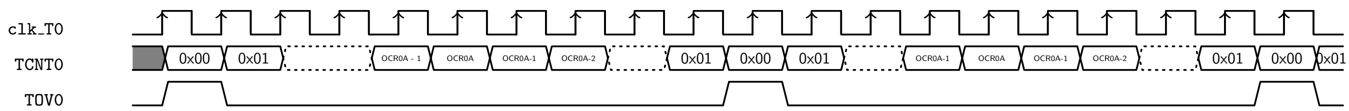
- ***WGM0[2:0]*** −− > 001 or 101.
- High resolution phase correct PWM.
- Motor control due to symmetric features
- Dual slope operations causing ower frequency PWM waveform.
- Counter starts from BOTTOM to TOP and then from TOP to BOTTOM.
- TOP is defined by
  - TOP == 0xFF if ***WGM0[2:0]*** −− > 001
  - TOP == **OCR0A** if ***WGM0[2:0]*** −− > 101
- When ***COM0A[1:0]*** == 01, the *OC0A* pin output can be set to toggle its match between **TCNT0** and TOP to generate waveform.
  - The above is possible only when ***WGM02*** bit is set.
  - And only on *OC0A* pin and not on *OC0B* pin.
- In Inverting Compare Mode ***COM0A[1:0]*** == 10 , the *OC0A* or *OC0B* pins is made 1 on compare match between **TCNT0** and TOP and made 0 on reaching BOTTOM.
- In Non-Inverting Compare Mode ***COM0A[1:0]*** == 11 , the *OC0A* or *OC0B* pins is made 0 on compare match between **TCNT0** and TOP and 1 made on reaching BOTTOM.
- The Timer/Counter overflow flag (***TOV0***) is set each time the counter reaches BOTTOM..
- The PWM frequency is given by

$$f_{OC0xPWM} = \frac{f_{clkT0}}{N*510}$$

WGM[2:0] == 001



WGM[2:0] == 101



5.6 Register Description

TCCR0A – Timer/Counter Control Register A

7	6	5	4	3	2	1	0
COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00

COM0B[1:0]	Non-PWM modes	Fast PWM	Phase Corrected PWM
00	No output @ *PD5 - OC0B* pin	No output @ *PD5 - OC0B*	No output @ *PD5 - OC0B*
01	Toggle *PD5 - OC0B* pin on compare Match.	Reserved	Reserved
10	Clear *PD5 - OC0B* pin on compare Match.	Clear *PD5 - OC0B* on compare match and set *PD5 - OC0B* at BOTTOM	Clear *PD5 - OC0B* on compare match when up-counting and set *PD5 - OC0B* on compare match when down-counting.
11	Set *PD5 - OC0B* pin on compare Match.	Set *PD5 - OC0B* on compare match and clear *PD5 - OC0B* at BOTTOM	Set *PD5 - OC0B* on compare match when up-counting and clear *PD5 - OC0B* on compare match when down-counting

COM0A[1:0]	Non-PWM modes	Fast PWM	Phase Corrected PWM
00	No output @ *PD6 - OC0A* pin	No output @ *PD6 - OC0A*	No output @ *PD6 - OC0A*
01	Toggle *PD6 - OC0A* pin on compare Match.	When WGM0[2] == 1, Toggle *PD6 - OC0A* pin on Compare match	Toggle *PD6 - OC0A* pin on Compare match
10	Clear *PD6 - OC0A* pin on compare Match.	Clear *PD6 - OC0A* on compare match and set *PD6 - OC0A* at BOTTOM	Clear *PD6 - OC0A* on compare match when up-counting and set *PD6 - OC0A* on compare match when down-counting.
11	Set *PD6 - OC0A* pin on compare Match.	Set *PD6 - OC0A* on compare match and clear *PD6 - OC0A* at BOTTOM	Set *PD6 - OC0A* on compare match when up-counting and clear *PD6 - OC0A* on compare match when down-counting

WGM0[2:0]	Mode of operation	TOP	TOV0 Flag set on
000	Normal	0xFF	MAX
001	PWM Phase Corrected	0xFF	BOTTOM
010	CTC	OCRA	MAX
011	Fast PWM	0xFF	MAX
101	PWM Phase Corrected	OCR0A	BOTTOM
111	Fast PWM	OCR0A	TOP

TCCR0B – Timer/Counter Control Register B

7	6	5	4	3	2	1	0
FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00

CS0[2:0]	**Description(Prescalar)**
000	No clock source(Timer/Counter Stopped)
001	$clk_{I/O}$ – no prescaling
010	$\frac{clk_{I/O}}{8}$
011	$\frac{clk_{I/O}}{64}$
100	$\frac{clk_{I/O}}{256}$
101	$\frac{clk_{I/O}}{1024}$
110	External clock source on *T0* pin. Clock on falling edge.
111	External clock source on *T0* pin. Clock on rising edge.

TIMSK0 – Timer/Counter Interrupt Mask Register

7	6	5	4	3	2	1	0
-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

Enable interrupts for compare match between **TCNT0** and **OCR0A** or **TCNT0** and **OCR0B** or overflow in **TCNT**0.

TIFR0 – Timer/Counter 0 Interrupt Flag Register

7	6	5	4	3	2	1	0
-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

FLag registers for interrupts on compare match between **TCNT0** and **OCR0A** or **TCNT0** and **OCR0B** or overflow in **TCNT0**.

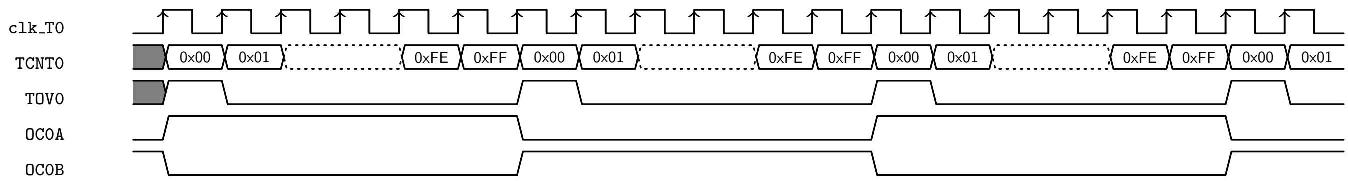
5.7 Configuring the Timer/Counter

5.7.1 Normal Mode

As Timer

$$ON\_TIME = \frac{max\_count}{\frac{F\_CPU}{PRESCALAR}}$$

- Depending on PRESCALR value, we get different ON_TIME.
- First, ***WGM0[2:0]*** bits are configured as 000 for Normal Mode in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0A[1:0]*** and/or ***COM0A[1:0]*** bits are configured to make outputs *OC0A* and/or *OC0B* pins to do nothing, set, clear or toggle in **TCCR0A** register.
- Next, Interrupt is Enabled by ***TOIE0*** (overflow enable) in **TIMSK0** reigster.
- Finally, Timer is started by setting prescalar in ***CS0[2:0]*** bits as needed prescalar of **TCR0B** reigster.
- Global Interrupt is enabled.
- A interrupt Service Routine for Timer0 overflow is Written.
- No need to clear the overflow flag as it is done by hardware.
- The timing when both pins *OC0A* and *OC0B* are made to toggle.



- The code can be seen below,

```
// MOde of operation to Normal Mode -- WGM0[2:0] === 000
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A & (~(1<<0) & ~(1<<1));
TCCR0B = TCCR0B & ~(1<<3);

/* What to do when timer reaches the MAX(0xFF) value */
// toggle OC0A and OC0B on each time when reaches the MAX(0xFF)
// which is reflected in PD6 and PD5

// Output OC0A to toglle when reaches MAX -- COM0A[1:0] === 01
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A & ~(1<<7);
TCCR0A = TCCR0A | (1<<6);

// Output OC0B to toglle when reaches MAX -- COM0B1:0] === 01
// COM0B[1](bit7) from TCCR0A, COM0B[0](bit6) from TCCR0A
TCCR0A = TCCR0A & ~(1<<5);
TCCR0A = TCCR0A | (1<<4);

//Enable Interrupt of OVERFLOW flag so that interrupt can be generated
TIMSK0 = TIMSK0 | (1<<0);

// start timer by setting the clock prescalar
// DIVIDE BY 8 from I/O clock
// DIVIDE BY 8-- CS0[2:0] === 010
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<1);
TCCR0B = TCCR0B & (~(1<<0) & ~(1<<2));

// enabling global interrupt
sei();
```

```
// SO ON TIME = max_count / (F_CPU / PRESCALAR)
// ON TIME = 0xFF / (16000000/8) = 128us
// since symmetric as toggling OFF TIME = 128us
// hence, we get a square wave of fequency 1 / 256us = 3.906kHz
```

```
ISR(TIMER0_OVF_vect)
{
    // do the thing when overflows.
}
```

As Counter

- Every rising/falling edge the count increases.
- So to reach 256 count, it would take a time of $\frac{0xFF}{frequency@T0pin}$.
- First, ***WGM0[2:0]*** bits are configured as 000 for Normal Mode in **TCCR0A** and **TCCR0B** registers.
- Finally, Counter is started by configuring ***CS0[2:0]*** bits to 110 or 111 for external falling or rising edge on *T0 - PD4*.
- The code when *T0* pin is used as counter @ falling edge.

```
// MOde of operation to Normal Mode -- WGM0[2:0] === 000
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A & (~(1<<0) & ~(1<<1));
TCCR0B = TCCR0B & ~(1<<3);

/* to count external event -we must connect source to T0 (PD4) */
// THE CLK IS CLOCKED FROM external source
// Falling edge of T0(PD4) -- CS0[2:0] === 110
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<2);
TCCR0B = TCCR0B | (1<<1);
TCCR0B = TCCR0B & ~(1<<0);
```

Application I - Delay

```
/* TCNT0 starts from 0X00 goes upto 0XFF and restarts */
/* No possible use case as it just goes upto 0xFF and restarts */
// MOde of operation to Normal Mode -- WGM0[2:0] === 000
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A & (~(1<<0) & ~(1<<1));
TCCR0B = TCCR0B & ~(1<<3);

/* What to do when timer reaches the MAX(0xFF) value */
// nothing should be done on OC0A for delay
// nothing  -- COM0A[1:0] === 00
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A & ~(1<<7);
TCCR0A = TCCR0A & ~(1<<6);

/* The delay possible = 0xff / (F_CPU/prescalar) */
// lowest delay = 0xff / (16000000 / 1) = 16us
// when prescalar == 8 --> delay = 0xff / (16000000 / 8) = 128us
// when prescalar == 64 --> delay = 0xff / (16000000 / 64) = 1.024ms
// when prescalar == 256 --> delay = 0xff / (16000000 / 256) = 4.096ms
// highest delay possible = 0xff / (16000000 / 1024) = 16.38ms

// start timer by setting the clock prescalar
// DIVIDE BY 8 use the same clock from I/O clock
```

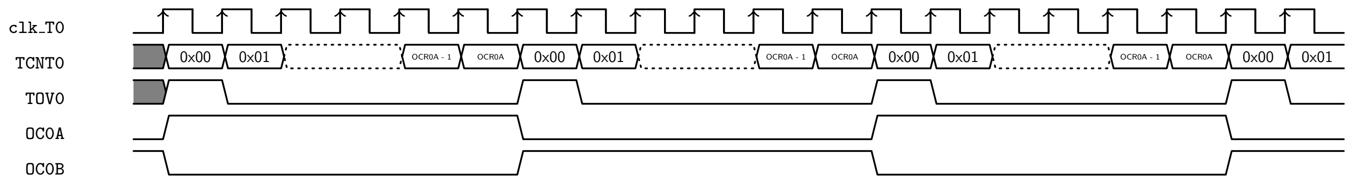
```
// DIVIDE BY 8-- CS0[2:0] === 010
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B & ~(1<<0);
TCCR0B = TCCR0B | (1<<1);
TCCR0B = TCCR0B & ~(1<<2);


// actual delaying - wait until delay happens
while((TIFR0 & 0x01) == 0x00); // checking overflow flag when overflow happns
// clearing the overflag so that we can further utilize
TIFR0 = TIFR0 | 0x01;
```

5.7.2 CTC Mode

As Timer

$$ON\_TIME = \frac{1+OCR0A}{\frac{F\_CPU}{PRESCALAR}}$$

- Depending on **OCR0A** register and PRESCALR value, we get different ON_TIME.
- First, ***WGM0[2:0]*** bits are configured as 010 for CTC Mode in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0A[1:0]*** and/or ***COM0B[1:0]*** bits are configured to make outputs *OC0A* and/or *OC0B* pins to do nothing, set, clear or toggle in **TCCR0A** register.
- Next, Interrupt is Enabled by ***OCIE01A*** (utput compare on match on **OCR0A** register enable) in **TIMSK0** reigster.
- Finally, Timer is started by setting prescalar in ***CS0[2:0]*** bits as needed prescalar of **TCR0B** reigster.
- Global Interrupt is enabled.
- A interrupt Service Routine for Timer0 Compare is Written.
- No need to clear the overflow flag as it is done by hardware.
- The timing when both pins *OC0n* are made to toggle.



- The code can be seen below,

```
// MOde of operation to CTC Mode -- WGM0[2:0] === 010
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A & ~(1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B & ~(1<<3);

/* What to do when timer reaches the OCR0A */
// toggle OC0A on each time when reaches the OCR0A
// which is reflected in PD6
// Output OC0A to toglle when reaches MAX -- COM0A[1:0] === 01
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A & ~(1<<7);
TCCR0A = TCCR0A | (1<<6);

// Output OC0B to toglle when reaches MAX -- COM0B1:0] === 01
// COM0B[1](bit7) from TCCR0A, COM0B[0](bit6) from TCCR0A
TCCR0A = TCCR0A & ~(1<<5);
TCCR0A = TCCR0A | (1<<4);
```

```
// Enable Interrupt when counter matches OCR0A Rgister
//  OCIE0A bit is enabled
TIMSK0 = TIMSK0 | (1<<1);


// setting the value till the counter should reach in OCR0A
// for toggling of OC0A pin
OCR0A = 0x32;

// start timer by setting the clock prescalar
// DIVIDE BY 8 from I/O clock
// DIVIDE BY 8-- CS0[2:0] === 010
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<1);
TCCR0B = TCCR0B & (~(1<<0) & ~(1<<2));

// enabling global interrupt
sei();
// SO ON TIME = (1 + OCR0A) / (F_CPU / PRESCALAR)
// ON TIME = 0X32 / (16000000/8) = 25.5us
// since symmetric as toggling OFF TIME = 25.5us
// hence, we get a square wave of fequency 1 / 50us = 20kHz
```

```
ISR(TIMER0_COMPA_vect)
{
    // do the thing when compare match between TCNT0 matches OCR0A.
}
```

As Counter

- Every rising/falling edge the count increases.
- So to reach required count, it would take a time of $\frac{OCR0A}{frequency@T0pin}$.
- First, ***WGM0[2:0]*** bits are configured as 010 for CTC Mode in **TCCR0A** and **TCCR0B** registers.
- Finally, Counter is started by configuring ***CS0[2:0]*** bits to 110 or 111 for external falling or rising edge on *T0 - PD4* pin.
- The code when *T0* pin is used as counter @ falling edge.

```
// MOde of operation to CTC Mode -- WGM0[2:0] === 010
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A & ~(1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B & ~(1<<3);

// Disbale Interrupt when counter matches OCR0A Rgister
//  OCIE0A bit is disabled
TIMSK0 = TIMSK0 & ~(1<<1);

//we count till OCR0A register value and reset and continue
OCR0A = 0xA;

/* to count external event -we must connect source to T0 (PD4) */
// THE CLK IS CLOCKED FROM external source
// Falling edge of T0(PD4) -- CS0[2:0] === 110
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<2);
TCCR0B = TCCR0B | (1<<1);
TCCR0B = TCCR0B & ~(1<<0);
```

Application I - Delay in ms

```
// minimum delay being 4us -- choose like that
// use PRESCALAR OF 1 -- 3us - 16us -- usage 3us - 16us -- factor=0 -- CS0[2:0]=1
// use PRESCALAR OF 8 -- 3us - 128us -- usage 17us - 128us -- factor=3 -- CS0[2:0]=2
// use PRESCALAR OF 64 -- 4us - 1.024ms -- usage 129us - 1024us -- factor=6 -- CS0[2:0]=3
// use PRESCALAR OF 256 -- 16us - 4.096ms -- usage 1025us - 4096us -- factor=8 -- CS0[2:0]=4

// MOde of operation to ctc Mode -- WGM0[2:0] === 010
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A & ~(1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B & ~(1<<3);

while(delayInMs--)
{
    // for 1ms delay
    OCR0A = 249;
    // start timer by setting the clock prescalar
    //  dived by 64 from I/O clock
    //  CS0[2:0] === 011
    // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
    TCCR0B = TCCR0B | (1<<0);
    TCCR0B = TCCR0B | (1<<1);
    TCCR0B = TCCR0B & ~(1<<2);

    // actual delaying - wait until delay happens
    while((TIFR0 & 0x02) == 0x00); // checking OCF0A (compare match flag A) flag when match happns
    // clearing the compare match flag so that we can further utilize
    TIFR0 = TIFR0 | 0x02;
}
```

5.7.3 Fast PWM Mode

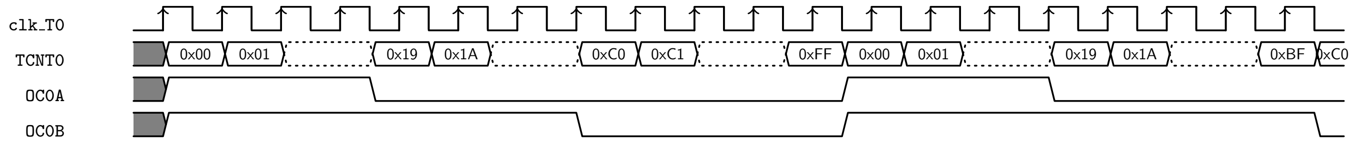
```
ISR(TIMER0_OVF_vect)
{
}
ISR(TIMER0_COMPA_vect)
{
}
ISR(TIMER0_COMPB_vect)
{
}
```

Non-Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR0A** and/or **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 011 for Fast PWM Mode with TOP at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0A[1:0]*** and/or ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make outputs *OC0A* and/or *OC0B* pins to generate PWM by comparing between **OCR0A** and/or **OCR0B** respectively. That is for Non-Inverting, ***COM0x[1:0]*** is written 10.
- Next, the duty cycle value is loaded into **OCR0A** and/or **OCR0B** register for *OC0A* and/or *OC0B* pins.
- Also, the ***OCIE0A*** and/or ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.

- The timing for PWM on 10% duty cycle *OC0A* and 75% duty cycle*OC0B* pins are shown assuming .
  - 0x19 for OCR0A.
  - 0xC0 for OCR0B.



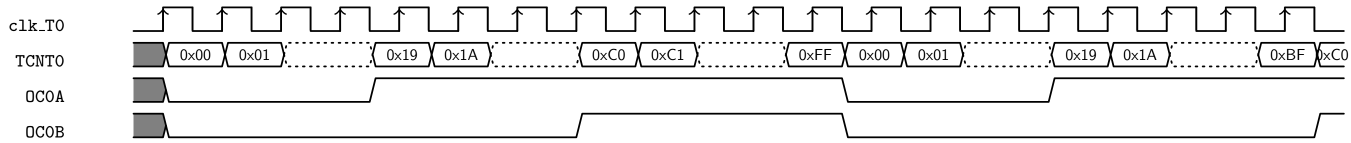

```
// MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 011
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B & ~(1<<3);

// here we set COM0A[1:0] as 10 for non-inverting
// here we set COM0B[1:0] as 10 for non-inverting

// which is reflected in PD6
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A | (1<<7);
TCCR0A = TCCR0A & ~(1<<6);

// which is reflected in PD65
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A & ~(1<<4);

// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV0 bit is enabled
TIMSK0 = TIMSK0 | (1<<0);

/* we use OCF0A flag - which is set at every time TCN0 reaches OCR0A
here we clear led(PC1),  so that we obtain the PWM when TCN0 reaches OCR0A*/
TIMSK0 = TIMSK0 | (1<<1);
/* we use OCF0B flag - which is set at every time TCN0 reaches OCR0B
here we clear led(PC2),  so that we obtain the PWM when TCN0 reaches OCR0B*/
TIMSK0 = TIMSK0 | (1<<2);


// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till max(0xFF), we can choose OCR0A and OCR0B to any value below max(0xFFF)
OCR0A = 0x19; // for 10% duty clcle
OCR0B = 0xC0; // for 75% duty clcle


// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR0A** and/or **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 011 for Fast PWM Mode with TOP at MAX in **TCCR0A** and **TCCR0B** registers.

- Next, ***COM0A[1:0]*** and/or ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make outputs *OC0A* and/or *OC0B* pins to generate PWM by comparing between **OCR0A** and/or **OCR0B** respectively. That is for Inverting, ***COM0x[1:0]*** is written 11.
- Next, the duty cycle value is loaded into **OCR0A** and/or **OCR0B** register for *OC0A* and/or *OC0B* bits.
- Also, the ***OCIE0A*** and/or ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.
- The timing for PWM on 10% duty cycle *OC0A* and 75% duty cycle *OC0B* pins are shown assuming .
  - 0x19 for OCR0A.
  - 0xC0 for OCR0B.



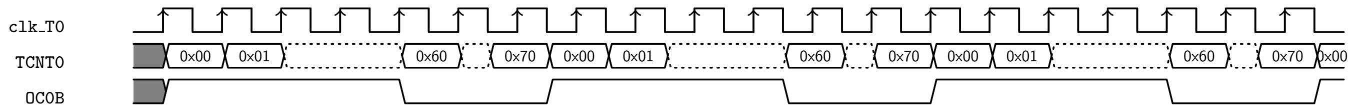

```
// MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 011
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B & ~(1<<3);

// here we set COM0A[1:0] as 11 for inverting
// here we set COM0B[1:0] as 11 for inverting

// which is reflected in PD6
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A | (1<<7);
TCCR0A = TCCR0A | (1<<6);

// which is reflected in PD65
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A | (1<<4);

// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV0 bit is enabled
TIMSK0 = TIMSK0 | (1<<0);

/* we use OCF0A flag - which is set at every time TCN0 reaches OCR0A
    here we clear led(PC1),  so that we obtain the PWM when TCN0 reaches OCR0A*/
TIMSK0 = TIMSK0 | (1<<1);
/* we use OCF0B flag - which is set at every time TCN0 reaches OCR0B
    here we clear led(PC2),  so that we obtain the PWM when TCN0 reaches OCR0B*/
TIMSK0 = TIMSK0 | (1<<2);


// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till max(0xFF), we can choose OCR0A and OCR0B to any value below max(0xFFF)
OCR0A = 0x19; // for 10% duty clcle
OCR0B = 0xC0; // for 75% duty clcle


// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
```

```
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Non-Inverting PWM with TOP at OCR0A

Frequency is chosen by **OCR0A** and Duty cycle by **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 111 for Fast PWM Mode with **OCR0A** at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make output *OC0B* pins to generate PWM by comparing between **TCNT0** and **OCR0B**. That is for Non-Inverting, ***COM0B[1:0]*** is written 10.
- The frequency of duty cycle is loaded into **OCR0A** register.
- Next, the duty cycle value is loaded into **OCR0B** register for *OC0B* bits.
- Also, the ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.
- The timing for PWM on 85% duty cycle(0x60) *OC0B* pins are shown assuming .
  - 0x70 for OCR0A.
  - 0x60 for OCR0B.

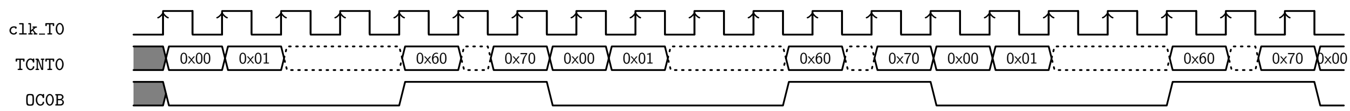



```
// MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 111
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B | (1<<3);

// here we set COM0B[1:0] as 10 for non-inverting
// which is reflected in PD5
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A & ~(1<<4);

// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
OCR0A = 0x70; // for freqeuncy
OCR0B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at OCR0A

Frequency is chosen by **OCR0A** and Duty cycle by **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 111 for Fast PWM Mode with **OCR0A** at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make output *OC0B* pins to generate PWM by comparing between **TCNT0** and **OCR0B**. That is for Inverting, ***COM0B[1:0]*** is written 11.
- The frequency of duty cycle is loaded into **OCR0A** register.
- Next, the duty cycle value is loaded into **OCR0B** register for *OC0B* bits.
- Also, the ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.
- The timing for PWM on 85% duty cycle *OC0B* pins are shown assuming .
  - 0x70 for OCR0A.
  - 0x60 for OCR0B.



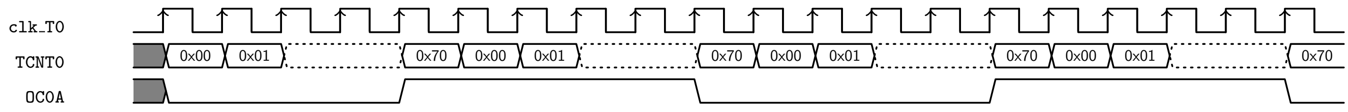

```
// MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 111
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B | (1<<3);

// here we set COM0B[1:0] as 11 for inverting
// which is reflected in PD5
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A | (1<<4);

// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
OCR0A = 0x70; // for freqeuncy
OCR0B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Toggling mode square Wave

Frequency is chosen by **OCR0A** register.

- First, ***WGM0[2:0]*** bits are configured as 111 for Fast PWM Mode with OCR0A at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0A[1:0]*** bits of **TCCR0A** register are configured to make output *OC0A* pins to generate PWM by comparing between **OCR0A**. That is for Toggling square wave ***COM0A[1:0]*** is written 01.
- The frequency of duty cycle is loaded into **OCR0A** register.

- Also, the ***OCIE0A*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.
- The timing for squared wave on *OC0A* pins are shown assuming.
  - 0x70 for OCR0A.




```
// MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 111
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A | (1<<1);
TCCR0B = TCCR0B | (1<<3);

// here we set COM0B[1:0] as 01 for toggling of OC0A
// which is reflected in PD6
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A & ~(1<<7);
TCCR0A = TCCR0A | (1<<6);

// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
OCR0A = 0x70; // for freqeuncy

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Application I - PWM generation

```
void Timer0_FastPWMGeneration(uint32_t on_time_us, uint32_t off_time_us)
{
        uint32_t total_time = on_time_us + off_time_us;

        // MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 111
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A | (1<<1);
        TCCR0B = TCCR0B | (1<<3);

        // which is reflected in PD5
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);

        if(total_time <=3)
        {
                // if total_time <= 3us -- so we stop clock

                OCR0A = 0;
                // start timer by setting the clock prescalar
```

```
                //  use the same clock from I/O clock
                //  CS0[2:0] === 001
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B & ~(1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((3 < total_time)  && (total_time <= 16))
        {
                OCR0A = ((total_time * 16) >> 0) - 1;
                OCR0B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS0[2:0] === 001
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B | (1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((16 < total_time)  && (total_time <= 128))
        {
                OCR0A = ((total_time * 16) >> 3) - 1;
                OCR0B = ((on_time_us * 16) >> 3) - 1;
                // start timer by setting the clock prescalar
                //  dived by 8 from I/O clock
                //  CS0[2:0] === 010
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B & ~(1<<0);
                TCCR0B = TCCR0B | (1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((128 < total_time)  && (total_time <= 1024))
        {
                OCR0A = ((total_time * 16) >> 6) - 1;
                OCR0B = ((on_time_us * 16) >> 6) - 1;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS0[2:0] === 011
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B | (1<<0);
                TCCR0B = TCCR0B | (1<<1);
                TCCR0B = TCCR0B & ~(1<<2);

        }
        else if((1024 < total_time)  && (total_time <= 4096))
        {
                OCR0A = ((total_time * 16) >> 8) - 1;
                OCR0B = ((on_time_us * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                //  divide by256 from I/O clock
                //  CS0[2:0] === 100
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B & ~(1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
                TCCR0B = TCCR0B | (1<<2);

        }
        else if(total_time > 4096)
        {
                // dont' cross more than 4.096ms
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
```

```
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }

        // max time = 4ms -- min freqency = 250 Hz
        //  min time = 4us -- max frequency = 250000 = 250khz
        Timer0_FastPWMGeneration(on_time_us, total_time_us - on_time_us);
}
```

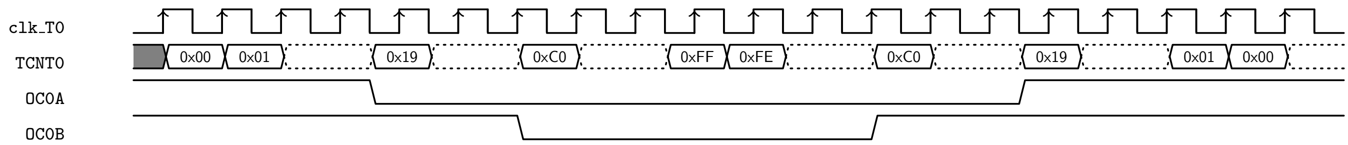
5.7.4 Phase Corrected PWM Mode

```
ISR(TIMER0_OVF_vect)
{
}
ISR(TIMER0_COMPA_vect)
{
}
ISR(TIMER0_COMPB_vect)
{
}
```

Non-Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR0A** and/or **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 001 for Phase Corrected PWM Mode with TOP at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0A[1:0]*** and/or ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make outputs *OC0A* and/or *OC0B* pins to generate PWM by comparing between **OCR0A** and/or **OCR0B** respectively. That is for Non-Inverting, ***COM0x[1:0]*** is written 10.
- Next, the duty cycle value is loaded into **OCR0A** and/or **OCR0B** register for *OC0A* and/or *OC0B* bits.
- Also, the ***OCIE0A*** and/or ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.
- The timing for PWM on 10% duty cycle *OC0A* and 75% duty cycle*OC0B* pins are shown assuming .
  - 0x19 for OCR0A.
  - 0xC0 for OCR0B.



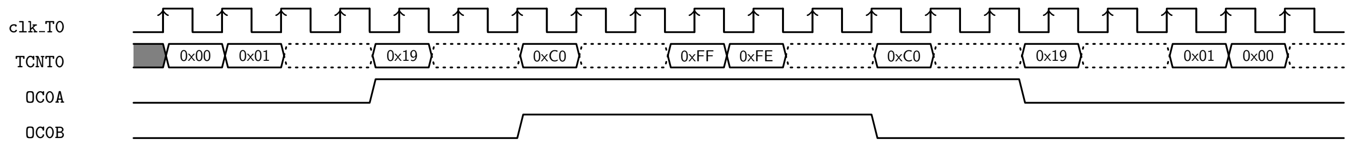

```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 001
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<3);

/* in timer0_phase_pwm_top_max, only two possiblites are there for COM0B[1:0] and COM0A[1:0] i.e)
↪  10(Inverting) and 11(Non- inverting) */
```

```
// here we set COM0A[1:0] as 10 for non-inverting
// here we set COM0B[1:0] as 10 for non-inverting

// which is reflected in PD6
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A | (1<<7);
TCCR0A = TCCR0A & ~(1<<6);

// which is reflected in PD65
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A & ~(1<<4);

/* we use overflow flag -- which is set at every time TCN0 reaches TOP here 0xFF
here, we toggle an led(PC0) at every overflow interrupt - this led(PC0) would give the frequency
↪   of PWM being generated -- done by PINC = PINC | 0X01;
Also, we set the other leds(PC1 and PC2) so that they are make one when TCN0 reaches 0x00 */
// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV0 bit is enabled
TIMSK0 = TIMSK0 | (1<<0);


// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till max(0xFF), we can choose OCR0A and OCR0B to any value below max(0xFFF)
OCR0A = 0x19; // for 10% duty clcle
OCR0B = 0xC0; // for 75% duty clcle

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR0A** and/or **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 001 for Phase Corrected PWM Mode with TOP at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0A[1:0]*** and/or ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make outputs *OC0A* and/or *OC0B* pins to generate PWM by comparing between **OCR0A** and/or **OCR0B** respectively. That is for Inverting, ***COM0x[1:0]*** is written 11.
- Next, the duty cycle value is loaded into **OCR0A** and/or **OCR0B** register for *OC0A* and/or *OC0B* bits.
- Also, the ***OCIE0A*** and/or ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.
- The timing for PWM on 10% duty cycle *OC0A* and 75% duty cycle*OC0B* pins are shown assuming .
  - 0x19 for OCR0A.
  - 0xC0 for OCR0B.



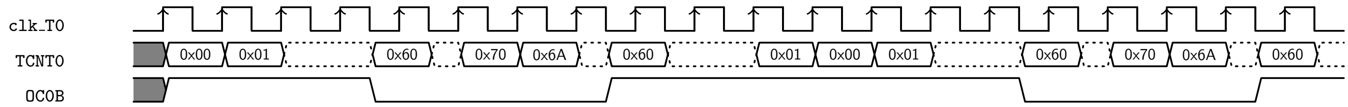

```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 001
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<3);

/* in timer0_phase_pwm_top_max, only two possiblites are there for COM0B[1:0] and COM0A[1:0] i.e)
↪  10(Inverting) and 11(Non- inverting) */

// here we set COM0A[1:0] as 11 for inverting
// here we set COM0B[1:0] as 11 for inverting

// which is reflected in PD6
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A | (1<<7);
TCCR0A = TCCR0A & ~(1<<6);

// which is reflected in PD65
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A & ~(1<<4);

/* we use overflow flag -- which is set at every time TCN0 reaches TOP here 0xFF
here, we toggle an led(PC0) at every overflow interrupt - this led(PC0) would give the frequency
↪  of PWM being generated -- done by PINC = PINC | 0X01;
Also, we set the other leds(PC1 and PC2) so that they are make one when TCN0 reaches 0x00 */
// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV0 bit is enabled
TIMSK0 = TIMSK0 | (1<<0);


// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till max(0xFF), we can choose OCR0A and OCR0B to any value below max(0xFFF)
OCR0A = 0x19; // for 10% duty clcle
OCR0B = 0xC0; // for 75% duty clcle

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Non-Inverting PWM with TOP at OCR0A

Frequency is chosen by **OCR0A** and Duty cycle by **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 101 for Phase Corrected PWM Mode with OCR0A at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make output *OC0B* pins to generate PWM by comparing between **OCR0B** respectively. That is for Non-Inverting, ***COM0B[1:0]*** is written 10.
- The frequency of duty cycle is loaded into **OCR0A** register.
- Next, the duty cycle value is loaded into **OCR0B** register for *OC0B* bits.

- Also, the ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting *CS0[2:0]* bit as needed prescalar in **TCR0B** register.
- The timing for PWM on 85% duty cycle(0x60) *OC0B* pins are shown assuming .
  - 0x70 for OCR0A.
  - 0x60 for OCR0B.

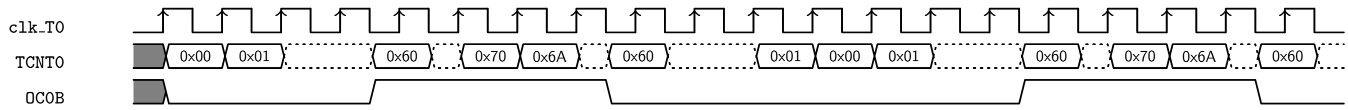



```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 101
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A & ~(1<<1);
TCCR0B = TCCR0B | (1<<3);

// here we set COM0A[1:0] as 10 for non-inverting
// which is reflected in PD5
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A & ~(1<<4);

// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
OCR0A = 0x70; // for freqeuncy
OCR0B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at OCR0A

Frequency is chosen by **OCR0A** and Duty cycle by **OCR0B** register.

- First, ***WGM0[2:0]*** bits are configured as 101 for Phase Corrected PWM Mode with OCR0A at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0B[1:0]*** bits of **TCCR0A** register are configured to make output *OC0B* pins to generate PWM by comparing between **OCR0B** respectively. That is for Inverting, ***COM0B[1:0]*** is written 11.
- The frequency of duty cycle is loaded into **OCR0A** register.
- Next, the duty cycle value is loaded into **OCR0B** register for *OC0B* bits.
- Also, the ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting *CS0[2:0]* bit as needed prescalar in **TCR0B** register.
- The timing for PWM on 85% duty cycle(0x60) *OC0B* pins are shown assuming .
  - 0x70 for OCR0A.
  - 0x60 for OCR0B.

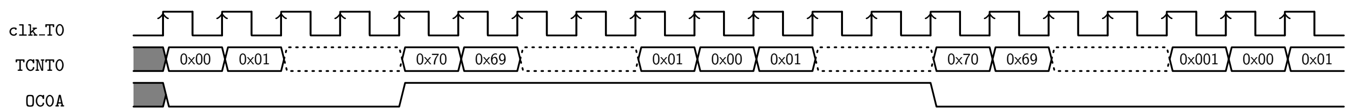



```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 101
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A & ~(1<<1);
TCCR0B = TCCR0B | (1<<3);

// here we set COM0A[1:0] as 11 for inverting
// which is reflected in PD5
// COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
TCCR0A = TCCR0A | (1<<5);
TCCR0A = TCCR0A | (1<<4);

// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
OCR0A = 0x70; // for freqeuncy
OCR0B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Toggling mode square Wave

Frequency is chosen by **OCR0A** register.

- First, ***WGM0[2:0]*** bits are configured as 101 for Phase Corrected PWM Mode with OCR0A at MAX in **TCCR0A** and **TCCR0B** registers.
- Next, ***COM0A[1:0]*** bits of **TCCR0A** register are configured to make output *OC0A* pins to generate PWM by comparing between **OCR0A**. That is for Toggling square wave ***COM0A[1:0]*** is written 01.
- The frequency of duty cycle is loaded into **OCR0A** register.
- Also, the ***OCIE0A*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS0[2:0]*** bit as needed prescalar in **TCR0B** register.
- The timing for squared wave on *OC0A* pins are shown assuming.
  - 0x70 for OCR0A.




```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 101
// WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
TCCR0A = TCCR0A | (1<<0);
TCCR0A = TCCR0A & ~(1<<1);
TCCR0B = TCCR0B | (1<<3);

// here we set COM0B[1:0] as 01 for toggling of OC0A
```

```
// which is reflected in PD6
// COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
TCCR0A = TCCR0A & ~(1<<7);
TCCR0A = TCCR0A | (1<<6);

// Next we set values for OCR0A and OCR0B
// Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
OCR0A = 0x70; // for freqeuncy

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS0[2:0] === 001
// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
TCCR0B = TCCR0B | (1<<0);
TCCR0B = TCCR0B & ~(1<<1);
TCCR0B = TCCR0B & ~(1<<2);

//enabled global interrupt
sei();
```

Application I - PWM generation

```
void Timer0_PhaseCorrectedPWMGeneration(uint32_t On_time_us, uint32_t Off_time_us)
{
        // Since, it is dual slope, the time would be doubled for one cylce, so we divide by 2
        uint32_t total_time = (On_time_us>>1) + (Off_time_us>>1);
        uint32_t on_time_us = On_time_us >> 1;

        // MOde of operation to phase_corrected_phase_top_max Mode -- WGM0[2:0] === 101
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A & ~(1<<1);
        TCCR0B = TCCR0B | (1<<3);

        // which is reflected in PD5
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);

        if(total_time <=3)
        {
                // if total_time <= 3us -- so we stop clock

                OCR0A = 0;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS0[2:0] === 001
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B & ~(1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((3 < total_time)  && (total_time <= 16))
        {
                OCR0A = ((total_time * 16) >> 0) - 1;
                OCR0B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS0[2:0] === 001
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B | (1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
```

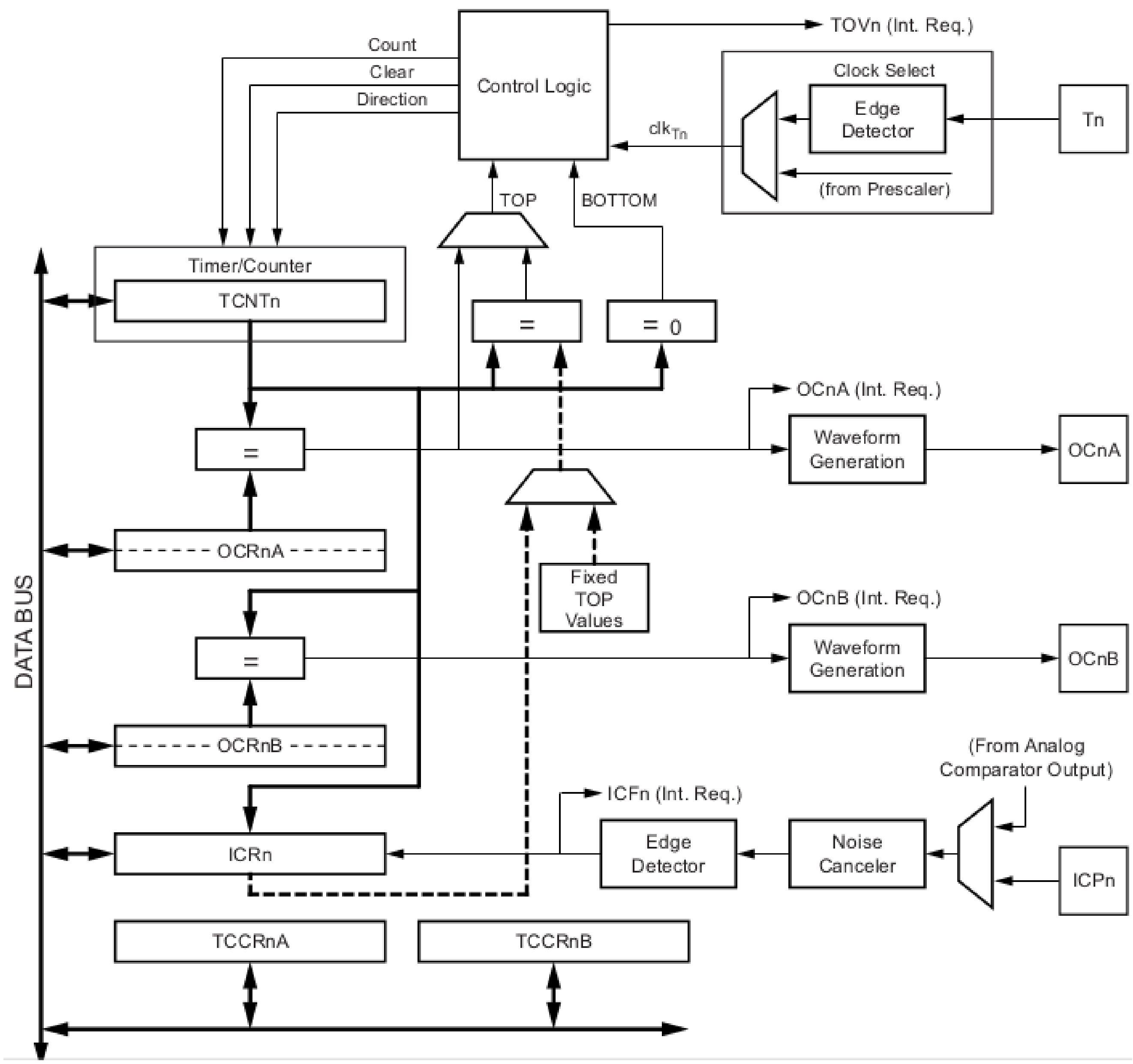
```
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((16 < total_time)  && (total_time <= 128))
        {
                OCR0A = ((total_time * 16) >> 3) - 1;
                OCR0B = ((on_time_us * 16) >> 3) - 1;
                // start timer by setting the clock prescalar
                //  dived by 8 from I/O clock
                //  CS0[2:0] === 010
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B & ~(1<<0);
                TCCR0B = TCCR0B | (1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((128 < total_time)  && (total_time <= 1024))
        {
                OCR0A = ((total_time * 16) >> 6) - 1;
                OCR0B = ((on_time_us * 16) >> 6) - 1;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS0[2:0] === 011
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B | (1<<0);
                TCCR0B = TCCR0B | (1<<1);
                TCCR0B = TCCR0B & ~(1<<2);

        }
        else if((1024 < total_time)  && (total_time <= 4096))
        {
                OCR0A = ((total_time * 16) >> 8) - 1;
                OCR0B = ((on_time_us * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                //  divide by256 from I/O clock
                //  CS0[2:0] === 100
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B & ~(1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
                TCCR0B = TCCR0B | (1<<2);

        }
        else if(total_time > 4096)
        {
                // dont' cross more than 4.096ms
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }

        // max time = 8ms -- min freqency = 125 Hz
        //  min time = 8us -- max frequency = 250000 = 125khz
        Timer0_PhaseCorrectedPWMGeneration(on_time_us, total_time_us - on_time_us);
}
```

Timer/Counter 1

6.1 Features

- General purpose 16-bit PWM/Counter module.
- Two independent output compare units and One input capture unit
- Variable PWM.
- Four independent interrupt sources (TOV1, OCF0A, OCF1B and ICF1).
- Clear timer on compare match (auto reload)

6.2 Block Diagram



6.3 Terminologies and Registers

Parameter	Description
BOTTOM	counter reaches 0x0000
MAX	ounter reaches 0xFFFF
TOP	counter reaches highest value (depends on mode of operation can be 0xFF, 0x1FF, 0x3FF, OCR1A, ICR1)

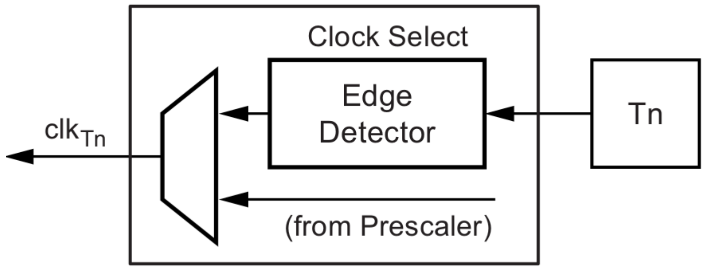
Register - 16 bit	Name
TCN10	Timer/Counter1count value
TCCR1A	Timer/Coutner1 Control Register A
TCCR1B	Timer/Coutner1 Control Register B
OCBR1A	Output compare register A
OCBR1B	Output compare register B
TIFR1	Timer Interrupt Flag Register
TIMSK1	Timer interrupt Mask Register
ICR1	Input Capture Register

Note:

- The **CNT1, OCR1A/B, ICR1** are 16-bit registers that can be accessed by the CPU via the 8-bit data bus.
- **For 16-bit write, the high byte must be written before the low byte.**
- **For 16-bit read, the low byte must be read before the high byte.**

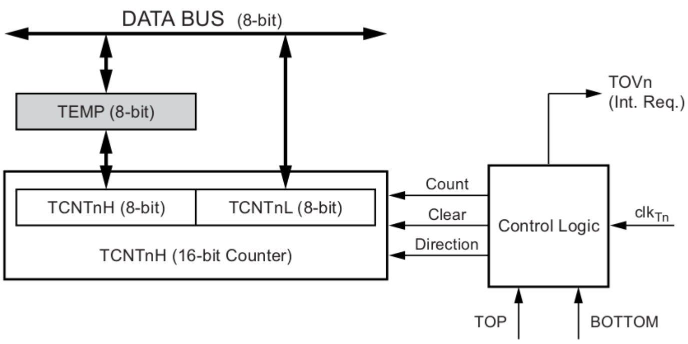
6.4 Timer/Counter1 Units

6.4.1 Clock Source/Select Unit



- The source for the Timer/Counter0 can be external or internal.
- External clock source is from *T1* pin.
- While Internal Clock source can be clocked via a prescalar.
- The output of this unit is the timer clock (clk_{T1}).
- It uses ***CS1[2:0]*** bits in **TCCR1B** register to select the source.

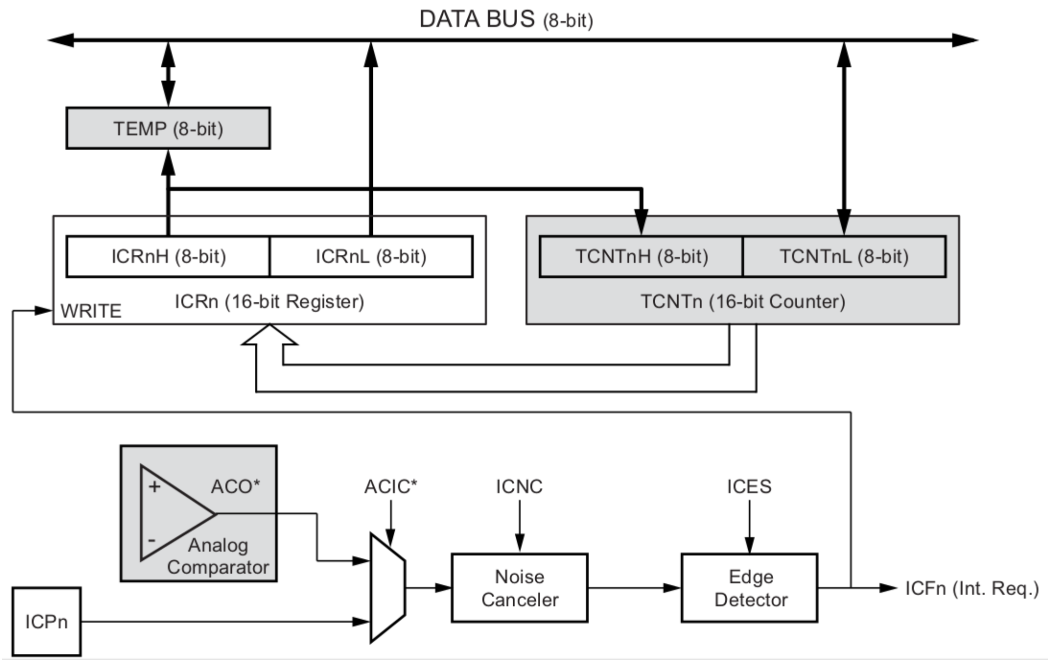
6.4.2 Counter Unit



Signal	Description
count	Increment or decrement **TCNT1** by 1
direction	Select between increment or decrement
clear	Clears **TCNT1** to 0x0000
clk_{T1}	Timer/Coutner1 clock
top	Signalize that **TCNT1** has maximum value
bottom	Signalize that **TCNT1** has minimum value(0x0000)

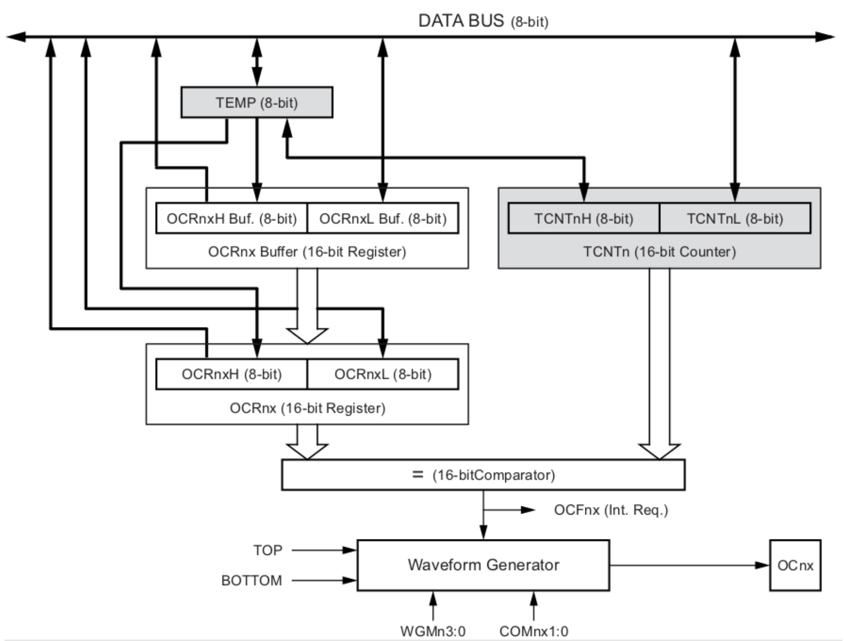
- The main part of the 16-bit Timer/Counter is the programmable bi-directional counter.
- Counter high (**TCNT1H**) containing the upper eight bits of the counter, and counter low (**TCNT1L**) containing the lower eight bits.
- Depending the mode of operation the counter is cleared, incremented, or decremented at each timer clock (clk_{T1}).
- Counting sequence is determined by ***WGM1[3:0]*** bits of **TCCR1A** -Timer/Counter1 Control register A and **TCCR1B** - Timer/Counter1 Control register B.
- The Timer/Counter1 Overflow flag ***(TOV1)*** is set and can generate interrupt according to the mode.

6.4.3 Input Capture Unit



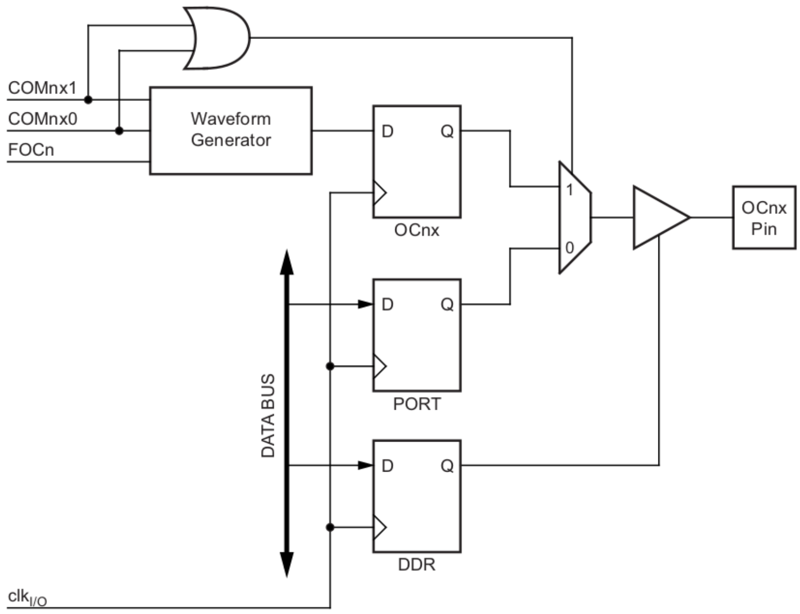
- Can capture external events and give them time-stamp indicating time of occurance.
- External signal can be from *ICP1* pin or analog-comparator unit.
- Usage : calculate frequency, duty-cycle, log of the signal
- When a change of the logic level (an event) occurs on the input capture pin (*ICP1*), or on the analog comparator output (*ACO)*, and this change confirms to the setting of the edge detector, a capture will be triggered.
- When a capture is triggered, the 16-bit value of the counter (**TCNT1**) is written to the input capture register (**ICR1**).
- The input capture flag (***ICF1***) is set at the same system clock as the **TCNT1** value is copied into **ICR1** register.
- If enabled (***ICIE1*** = 1), the input capture flag generates an input capture interrupt.
- ***ICF1*** flag is automatically cleared when the interrupt is executed and by writing on to i.
- An input capture can be triggered by software by controlling the port of the *ICP1* pin.

6.4.4 Output Compare Unit



- 16-bit comparator continuously compares **TCNT1** with both **OCR1A** and **OCR1B**.
- When **TCNT1** equals **OCR1A** or **OCR1B**, the comparator signals a match which will set the output compare flag at the next timer clock cycle.
- If interrupts are enabled, then output compare interrupt is generated.
- The waveform generator uses the match signal to generate an output according to operating mode set by the ***WGM1[3:0]*** bits and compare output mode ***COM0x[1:0]*** bits.

6.4.5 Compare Match Output Unit



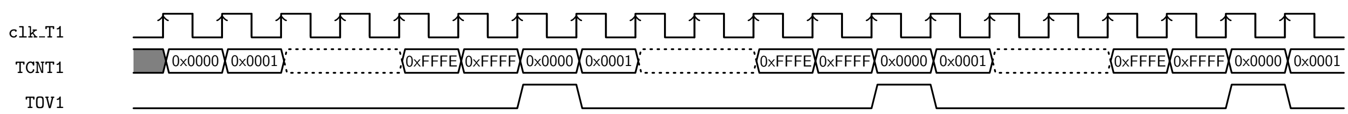
- This unit is used for changing the state of *OC1A* and *OC1B* pins by configuring the ***COM1x[1:0]*** bits.
- But, general I/O port function is overriiden by DDR reigster.

6.5 Modes of Operation

- The mode of operation can be defined by combination of waveform generation mode (***WGM1[3:0]***) and compare output mode(***COM1[1:0]***) bits.
- The waveform generation mode (***WGM1[3:0]***) bits affect the counting sequence.
- For non-PWM mode, ***COM1[1:0]*** bits control if the output should be set, cleared or toggled at a compare match.
- For PWM mode, ***COM1[1:0]*** bits control if the PWM generated should be inverted or non-inverted.

6.5.1 Normal Mode - Non-PWM Mode

- ***WGM1[3:0]*** $-->$ 000.
- Counter counts up and no counter clear.
- Overruns TOP(0XFFFF) and restarts from BOTTOM(0X0000).
- ***TOV1*** Flag is only set when overrun.
- We have to clear ***TOV1*** flag inorder to have next running.
- But, if we use interrupt we don't need to clear it as interrupt automatically clear the ***TOV1*** flag.
- The input capture unit can be used to capture events at *ICP1* pin or *ACO* pin.
- The timing can be seen below.



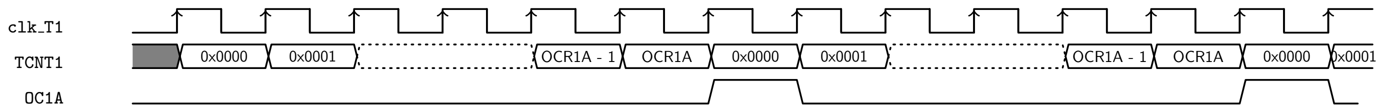
6.5.2 Clear Timer on Compare Match(CTC) Mode - Non-PWM Mode

- ***WGM1[3:0]*** −− > 0100 or 1100.
  - Counter value clears when **TCNT1** reaches **OCR1A** if ***WGM1[3:0]*** is 0100.
  - Counter value clears when **TCNT1** reaches **ICR1** if ***WGM1[3:0]*** is 1100.
- Interrupt can be generated each time **TCNT1** reaches **OCR1A** register value by ***OCF1A*** flag.
- Interrupt can be generated each time **TCNT1** reaches **ICR1** register value by ***ICF1*** flag.
- When ***COM1A[1:0]*** == 01, the *OC1A* pin output can be set to toggle its match between **TCNT1** and **OCR1A** or **ICR1** register to generate waveform.
- The frequency of the waveform its

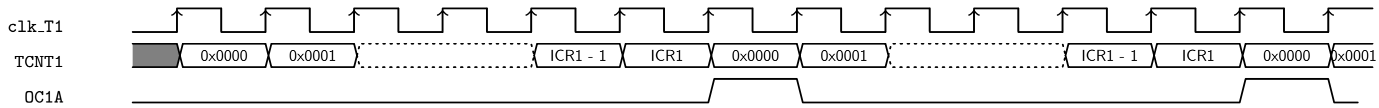
$$f_{OC1A} = \frac{f_{clkT1}}{2*N*(1+OCR1A)}$$

- Here N is prescalar factor and can be (1, 8, 64, 256, or 1024).

WGM1[3:0] == 0100



WGM1[3:0] == 1100

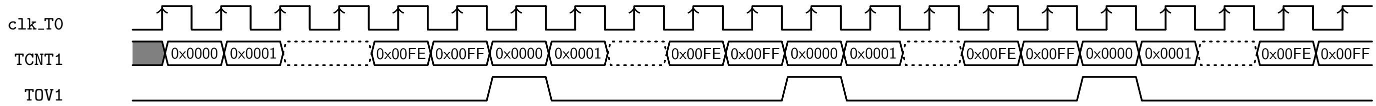


6.5.3 Fast PWM Mode

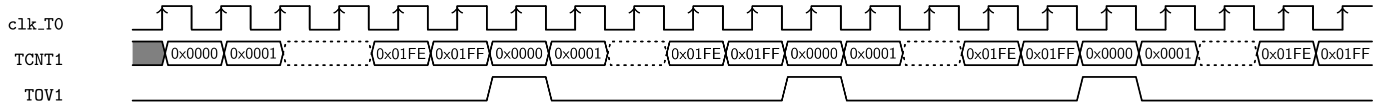
- ***WGM1[3:0]*** −− > 0101 or 0110 or 0111 or 1110 or 1111.
- Power Regulation, Rectification, DAC applications.
- Single slope operations causing high frequency PWM waveform.
- Counter starts from BOTTOM to TOP and then restarts from BOTTOM.
- TOP is defined by
  - TOP == 0x00FF if ***WGM1[3:0]*** −− > 0101
  - TOP == 0x01FF if ***WGM1[3:0]*** −− > 0110
  - TOP == 0x03FF if ***WGM1[3:0]*** −− > 0111
  - TOP == **ICR1** if ***WGM1[3:0]*** −− > 1110
  - TOP == **OCR1A** if ***WGM1[3:0]*** −− > 1111
- When ***COM1A[1:0]*** == 01, the *OC1A* pin output can be set to toggle its match between **TCNT1** and TOP to generate waveform.
  - The above is possible only when ***WGM12*** bit is set.
  - And only on *OC1A* pin and not on *OC1B* pin.
- In Inverting Compare Mode ***COM1A[1:0]*** == 10 , the *OC1A* or *OC1B* pins is made 1 on compare match between **TCNT1** and TOP and made 0 on reaching BOTTOM.
- In Non-Inverting Compare Mode ***COM1A[1:0]*** == 11 , the *OC1A* or *OC1B* pins is made 0 on compare match between **TCNT1** and TOP and 1 made on reaching BOTTOM.
- The Timer/Counter overflow flag (***TOV1***) is set each time the counter reaches TOP.
- The PWM frequency is given by

$$f_{OC1xPWM} = \frac{f_{clkT1}}{N*(1+TOP)}$$

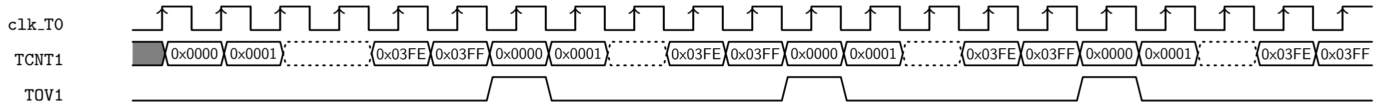
WGM1[3:0] == 0101



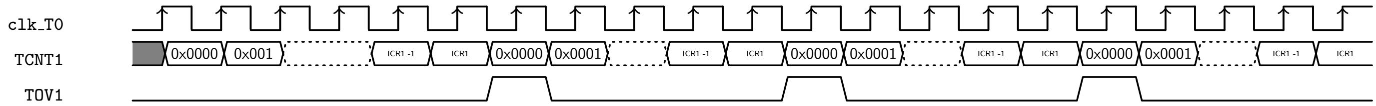
WGM1[3:0] == 0110



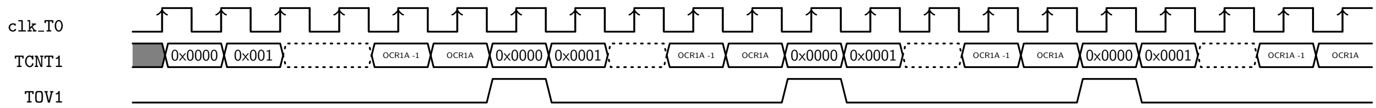
WGM1[3:0] == 0111



WGM1[3:0] == 1110



WGM1[3:0] == 1111

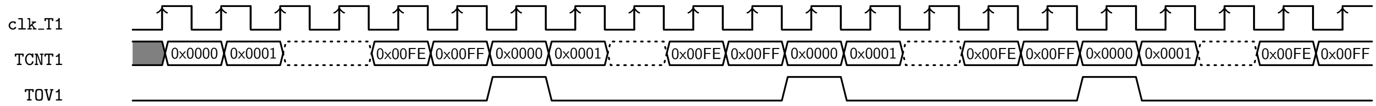


6.5.4 Phase Correct PWM Mode

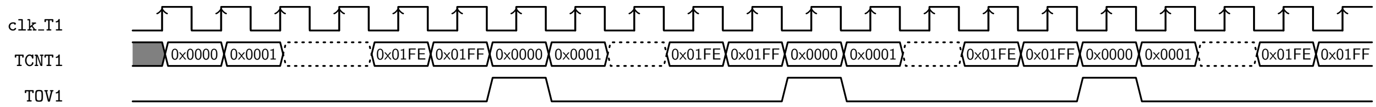
- ***WGM1[3:0]*** −− > 0001 or 0010 or 0011 or 1010 or 1011.
- High resolution phase correct PWM.
- Motor control due to symmetric features
- Dual slope operations causing ower frequency PWM waveform.
- Counter starts from BOTTOM to TOP and then from TOP to BOTTOM.
- TOP is defined by
  - TOP == 0x00FF if ***WGM1[3:0]*** −− > 0001
  - TOP == 0x01FF if ***WGM1[3:0]*** −− > 0010
  - TOP == 0x03FF if ***WGM1[3:0]*** −− > 0011
  - TOP == **ICR1** if ***WGM1[3:0]*** −− > 1010
  - TOP == **OCR1A** if ***WGM1[3:0]*** −− > 1011
- When ***COM1A[1:0]*** == 01, the *OC1A* pin output can be set to toggle its match between **TCNT1** and TOP to generate waveform.
  - The above is possible only when ***WGM12*** bit is set.
  - And only on *OC1A* pin and not on *OC1B* pin.
- In Inverting Compare Mode ***COM1A[1:0]*** == 10 , the *OC1A* or *OC1B* pins is made 1 on compare match between **TCNT1** and TOP and made 0 on reaching BOTTOM.
- In Non-Inverting Compare Mode ***COM1A[1:0]*** == 11 , the *OC1A* or *OC1B* pins is made 0 on compare match between **TCNT1** and TOP and 1 made on reaching BOTTOM.
- The Timer/Counter overflow flag (***TOV1***) is set each time the counter reaches BOTTOM..
- The PWM frequency is given by

$$f_{OC1xPWM} = \frac{f_{clkT1}}{2*N*TOP}$$

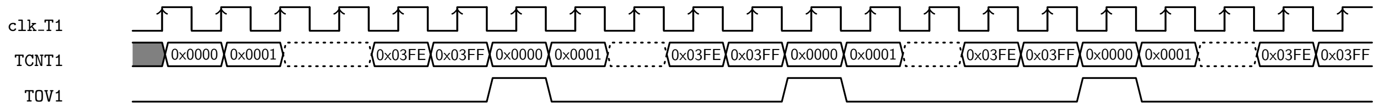
WGM1[3:0] == 0001



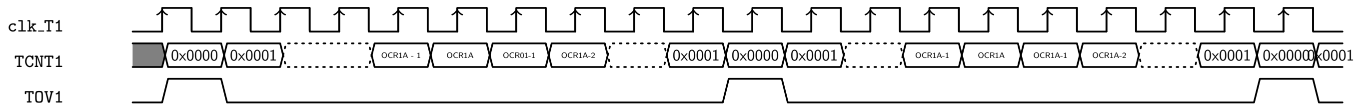
WGM1[3:0] == 0010



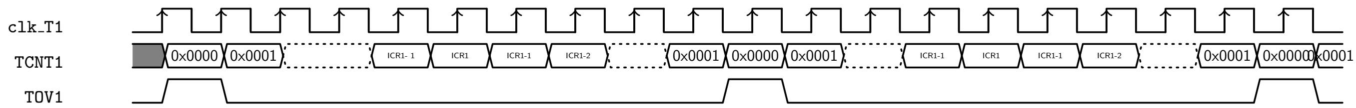
WGM1[3:0] == 0011



WGM[2:0] == 1010



WGM[2:0] == 1011



6.5.5 Phase and Frequency Corrected PWM Mode

- ***WGM1[3:0]*** −− > 1000 or 1001.
- High resolution and Phase correctd PWM.
- Dual-Slope.
- Counter counts from BOTTOM to TOP and then from TOP to BOTTOM.
  - TOP == **OCR1A** if ***WGM1[3:0]*** −− > 1001
  - TOP == **ICR1** if ***WGM1[3:0]*** −− > 1000
- In Inverting Compare Mode ***COM1x[1:0]*** == 10 the *OC0x* pins is made 1 on compare match between **TCNT1** and TOP when upcounting and made 0 on compare match between **TCNT1** and TOP when downcounting.
- In Non-Inverting Compare Mode ***COM1x[1:0]*** == 11, the *OC0x* pins is made 0 on compare match between **TCNT1** and TOP when upcounting AND made 1 on compare match between **TCNT1** and TOP when downcounting.
- The Timer/Counter overflow flag (***TOV1***) is set each time the counter reaches BOTTOM.
- The interrupt flag can be used to generate an interrupt each time the counter reaches the BOTTOM value.
- The PWM frequency is given by

$$f_{OC1xPWM} = \frac{f_{clkT1}}{2*N*TOP}$$

6.6 Register Description

TCCR1A – Timer/Counter 1 Control Register A

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10

COM1x[1:0]	**Non-PWM modes**	**Fast PWM**	**Phase Corrected PWM & Phase and Frequency Corrected PWM**
00	No output @ *PB1 - OC1A* or *PB2 - OC1B* pin	No output @ *PB1 - OC1A* or *PB2 - OC1B* pin	No output @ *PB1 - OC1A* or *PB2 - OC1B* pin
01	Toggle *PB1 - OC1A* or *PB2 - OC1B* pin on compare Match.	When *WGM[3:0]* == 1110 or 1111, Toggle *OC1A* pin on compare match	When *WGM[3:0]* == 1110 or 1111, Toggle *OC1A* pin on comapre match.
10	Clear *PB1 - OC1A* or *PB2 - OC1B* pin on compare Match.	Clear *PB1 - OC1A* or *PB2 - OC1B* on compare match and set *PB1 - OC1A* or *PB2 - OC1B* at BOTTOM	Clear *PD5 - OC0B* on compare match when up-counting and set *PB1 - OC1A* or *PB2 - OC1B* on compare match when down-counting.
11	Set *PB1 - OC1A* or *PB2 - OC1B* pin on compare Match.	Set *PB1 - OC1A* or *PB2 - OC1B* on compare match and clear *PB1 - OC1A* or *PB2 - OC1B* at BOTTOM	Set *PD5 - OC0B* on compare match when up-counting and clear *PB1 - OC1A* or *PB2 - OC1B* on compare match when down-counting.

WGM1[3:0]	**Mode of operation**	**TOP**	**TOV1 Flag set on**
0000	Normal	0xFFFF	MAX
0001	PWM Phase corrected – 8bit	0x00FF	BOTTOM
0010	PWM Phase corrected – 9bit	0x01FF	BOTTOM
0011	PWM Phase corrected – 10bit	0x03FF	BOTTOM
0100	CTC	OCR1A	MAX
0101	Fast PWM – 8bit	0x00FF	TOP
0110	Fast PWM – 9bit	0x01FF	TOP
0111	Fast PWM – 10bit	0x03FF	TOP
1000	PWM, phase and frequency corrected	ICR1	BOTTOM
1001	PWM, phase and frequency corrected	OCR1A	BOTTOM
1010	PWM, phase corrected	ICR1	BOTTOM
1011	PWM, phase corrected	OCR1A	BOTTOM
1100	CTC	ICR1	MAX
1110	Fast PWM	ICR1	TOP
1111	Fast PWM	OCR1A	TOP

TCCR1B – Timer/Counter1 Control Register B

7	6	5	4	3	2	1	0
ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

- ***ICNC1 - Input Capture Noise Canceler*** - activates the input capture noise canceler.
- ***ICES1 - Input Capture Edge Select*** - selects which edge on the input capture pin (*ICP1*) that is used to trigger a capture event. [1 - Rising edge; 0 - falling edge;]

CS1[2:0]	**Description(Prescalar)**
000	No clock source(Timer/Counter Stopped)
001	$clk_{I/O}$ – no prescaling
010	$\frac{clk_{I/O}}{8}$
011	$\frac{clk_{I/O}}{64}$
100	$\frac{clk_{I/O}}{256}$
101	$\frac{clk_{I/O}}{1024}$
110	External clock source on *T1* pin. Clock on falling edge.
111	External clock source on *T1* pin. Clock on rising edge.

TCNT1H – Timer/Counter1 Counter Higher Byte

7	6	5	4	3	2	1	0
TCNT1[15:8]							

TCNT1L – Timer/Counter1 Counter Lower Byte

7	6	5	4	3	2	1	0
TCNT1[7:0]							

OCR1AH – Output Compare Register 1 A Higher Byte

7	6	5	4	3	2	1	0
OCR1A[15:8]							

OCR1AL – Output Compare Register 1 A Lower Byte

7	6	5	4	3	2	1	0
OCR1A[7:0]							

OCR1BH – Output Compare Register 1 B Higher Byte

7	6	5	4	3	2	1	0
OCR1B[15:8]							

OCR1BL – Output Compare Register 1 B Lower Byte

7	6	5	4	3	2	1	0
OCR1B[7:0]							

ICR1H – Input Capture Register 1 Higher Byte

7	6	5	4	3	2	1	0
ICR1[15:8]							

ICR1L – Input Capture Register 1 Lower Byte

7	6	5	4	3	2	1	0
ICR1[7:0]							

TIMSK1 – Timer/Counter 1 Interrupt Mask Register

7	6	5	4	3	2	1	0
-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1

Enable interrupts for compare match between **TCNT1** and **OCR1A** or **TCNT1** and **OCR1B** or overflow in **TCNT1** or Input capture interrupt enable.

TIFR1 – Timer/Counter 1 Interrupt Flag Register

7	6	5	4	3	2	1	0
-	-	ICF1	-	-	OCIE1B	OCIE1A	TOIE1

Flag registers for interrupts on compare match between **TCNT0** and **OCR0A** or **TCNT0** and **OCR0B** or overflow in **TCNT1** or capture event occurs on the *ICP1* pin .

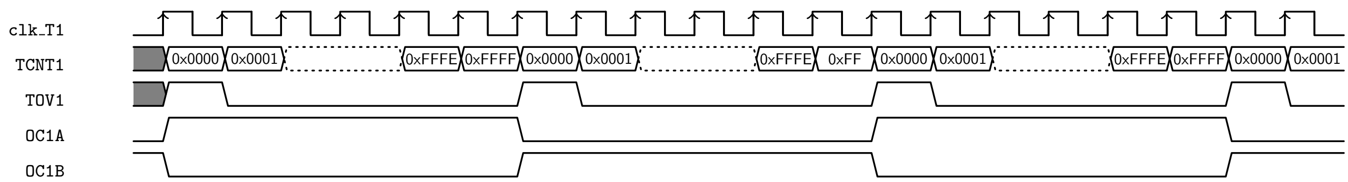
6.7 Configuring the Timer/Counter

6.7.1 Normal Mode

As Timer

$$ON\_TIME = \frac{max\_count}{\frac{F\_CPU}{PRESCALAR}}$$

- Depending on PRESCALR value, we get different ON_TIME.
- First, ***WGM1[3:0]*** bits are configured as 0000 for Normal Mode in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1A[1:0]*** and/or ***COM1A[1:0]*** bits are configured to make outputs *OC1A* and/or *OC1B* pins to do nothing, set, clear or toggle in **TCCR1A** register.
- Next, Interrupt is Enabled by ***TOIE1*** (overflow enable) in **TIMSK1** reigster.
- Finally, Timer is started by setting prescalar in ***CS1[2:0]*** bits as needed prescalar of **TCR1B** reigster.
- Global Interrupt is enabled.
- A interrupt Service Routine for Timer1 overflow is Written.
- No need to clear the overflow flag as it is done by hardware.
- The timing when both pins *OC1A* and *OC1B* are made to toggle.



- The code can be seen below,

```
// MOde of operation to Normal Mode -- WGM1[3:0] === 0000
// WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)
↪   from TCCR1A
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
```

```
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* What to do when timer reaches the MAX(0xFFFF) value */
// toggle OC1A on each time when reaches the MAX(0xFFFF)
// which is reflected in PB1
// Output OC1A to toglle when reaches MAX -- COM1A[1:0] === 01
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A & ~(1<<COM1A1);
TCCR1A = TCCR1A | (1<<COM1A0);

// toggle OC1B on each time when reaches the MAX(0xFFFF)
// which is reflected in PB2
// Output OC1B to toglle when reaches MAX -- COM1B[:0] === 01
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A & ~(1<<COM1B1);
TCCR1A = TCCR1A | (1<<COM1B0);


//Enable Interrupt of OVERFLOW flag so that interrupt can be generated
TIMSK1 = TIMSK1 | (1<<TOV1);

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

// enabling global interrupt

sei();
// SO ON TIME = max_count / (F_CPU / PRESCALAR)
// ON TIME = 0xFFFF / (16000000/1) = 4.096ms
// since symmetric as toggling OFF TIME = 4.096ms
// hence, we get a square wave of fequency 1 / 8.192ms = 122.07Hz
```

As Counter

- Every rising/falling edge the count increases.
- So to reach 0xFFFF count, it would take a time of $\frac{0xFFFF}{frequency@T1pin}$.
- First, ***WGM1[3:0]*** bits are configured as 0000 for Normal Mode in **TCCR1A** and **TCCR1B** registers.
- Finally, Counter is started by configuring ***CS1[2:0]*** bits to 110 or 111 for external falling or rising edge on *T1 - PD5*.
- The code when *T1* pin is used as counter @ falling edge.

```
// MOde of operation to Normal Mode -- WGM1[3:0] === 0000
// WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)
↪  from TCCR1A
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* to count external event -we must connect source to T1 (PD5) */
// THE CLK IS CLOCKED FROM external source
// Falling edge of T1(PD5) -- CS1[2:0] === 110
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
```

```
TCCR1B = TCCR1B & ~(1<<CS10);
TCCR1B = TCCR1B | (1<<CS11);
TCCR1B = TCCR1B | (1<<CS12);
```

As Input Capture

- Capture the value of **TCNT1** into **ICR1** register when there is rising or falling edge.
- First, ***WGM1[3:0]*** bits are configured as 0000 for Normal Mode in **TCCR1A** and **TCCR1B** registers.
- Next, the falling or rising edge for the *ICP1* pin is selected by ***ICES1*** bit in **TCCR1B**.
- The interrupts for input capture is enabled by setting the ***ICIE1*** bit in **TIMSK1**.
- A interrupt service routing is written.
- Finally, Timer is started by setting prescalar in ***CS1[2:0]*** bits as needed prescalar of **TCR1B** reigster.
- The code when *ICP1 - PB0* pin is used as capture @ rising edge.

```
// MOde of operation to Normal Mode -- WGM1[3:0] === 0000
// WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)
↪  from TCCR1A
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

// Select the edge for Input Capture
// ICES1(bit6) from TCCR1B
// Capture on Rising edge, ICES1 === 1
TCCR1B |= (1<<ICES1);

//Enable Interrupt of Input Capture Interrupt Enable so that interrupt can be generated
TIMSK1 = TIMSK1 | (1<<ICIE1);

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

// enabling global interrupt

sei();

ISR(TIMER1_CAPT_vect)
{
        if((TIFR1 & (1<<ICF1)) != 0)
        {
                capVal = ICR1L;
                capVal = (ICR1H<<8) | (capVal & 0xFF);
                // see datamemory
        }
}
```

Application I - Delay

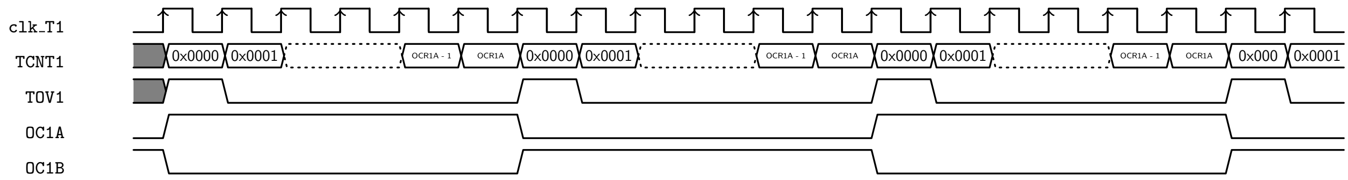
```
/* TCNT1 starts from 0X0000 goes upto 0XFFFF and restarts */
/* No possible use case as it just goes upto 0xFFFF and restarts */
// MOde of operation to Normal Mode -- WGM1[3:0] === 0000
// WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)
↪  from TCCR1A
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* What to do when timer reaches the MAX(0xFFFF) value */
// nothing should be done on OC1A for delay
// nothing  -- COM1A[1:0] === 00
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A & ~(1<<COM1A1);
TCCR1A = TCCR1A & ~(1<<COM1A0);

/* The delay possible = 0xffff / (F_CPU/prescalar) */
// lowest delay = 0xffff / (16000000 / 1) = 4.096ms
// when prescalar == 8 --> delay = 0xffff / (16000000 / 8) = 32.768ms
// when prescalar == 64 --> delay = 0xffff / (16000000 / 64) = 262.144ms
// when prescalar == 256 --> delay = 0xffff / (16000000 / 256) = 1.048576s
// highest delay possible = 0xffff / (16000000 / 1024) = 4.194304s

// start timer by setting the clock prescalar
// divede by 64 from I/O clock
// divede by 64-- CS1[2:0] === 101
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B | (1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);


// actual delaying - wait until delay happens
while((TIFR1 & 0x01) == 0x00); // checking overflow flag when overflow happns
// clearing the overflag so that we can further utilize
TIFR1 = TIFR1 | 0x01;
```

6.7.2 CTC Mode

As Timer

$$ON\_TIME = \frac{1+OCR1A}{\frac{F\_CPU}{PRESCALAR}}$$

- Depending on **OCR1A** register and/or **ICR1** register and PRESCALR value, we get different ON_TIME.
- First, ***WGM1[3:0]*** bits are configured as 0100 or 1100 for CTC Mode in **TCCR2A** and **TCCR1B** registers.
- Next, ***COM1A[1:0]*** and/or ***COM1B[1:0]*** bits are configured to make outputs *OC1A* and/or *OC1B* pins to do nothing, set, clear or toggle in **TCCR0A** register.
- Next, Interrupt is Enabled by ***OCIE1A*** (output compare on match on **OCR1A** register enable) in **TIMSK1** reigster.
- Finally, Timer is started by setting prescalar in ***CS1[2:0]*** bits as needed prescalar of **TCR1B** reigster.
- Global Interrupt is enabled.
- A interrupt Service Routine for Timer1 compare is Written.
- No need to clear the overflow flag as it is done by hardware.
- The timing when both pins *OC1n* are made to toggle.



- The code can be seen below,

```
// MOde of operation to Normal Mode -- WGM1[3:0] === 0100(TOP = OCR1A) or 1100(TOP = ICR1)
// WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)
↪  from TCCR1A
// we take TOP to be OCR1A for custom frequency
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* What to do when timer reaches the OCR1A value */
// toggle OC1A on each time when reaches the OCR1A
// which is reflected in PB1
// Output OC1A to togle when reaches OCR1A -- COM1A[1:0] === 01
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1A0);
TCCR1A = TCCR1A & ~(1<<COM1A1);

// toggle OC1B on each time when reaches the OCR1A
// which is reflected in PB2
// Output OC1B to togle when reaches OCR1A -- COM1B[1:0] === 01
// COM1B[1](bi57) from TCCR1A, COM1B[0](bit64) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1B0);
TCCR1A = TCCR1A & ~(1<<COM1B1);

// Enable Interrupt when counter matches OCR1A Rgister
//  OCIE1A  bit is enabled
TIMSK1 = TIMSK1 | (1<<OCIE1A);

// setting the value till the counter should reach in OCR1A
// for toggling of OC1A pin
OCR1A = 0x4861;

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

// enabling global interrupt

sei();
// SO ON TIME = (1 + OCR1A) / (F_CPU / PRESCALAR)
// ON TIME = 0x4861 / (16000000/1) = 1.15ms
// since symmetric as toggling OFF TIME = 1.15ms
// hence, we get a square wave of fequency 1 / 2.31ms = 431Hz
```

```
ISR(TIMER1_COMPA_vect)
{
    // do the thing when overflows.
}
```

As Counter

- Every rising/falling edge the count increases.
- So to reach required count, it would take a time of $\frac{OCR1A}{frequency@T1_{pin}}$.
- First, ***WGM1[3:0]*** bits are configured as 0100 or 1100 for CTC Mode in **TCCR2A** and **TCCR1B** registers.
- Finally, Counter is started by configuring ***CS1[2:0]*** bits to 110 or 111 for external falling or rising edge on *T1 - PD5* pin.
- The code when *T1* pin is used as counter @ falling edge.

```
// MOde of operation to Normal Mode -- WGM1[3:0] === 0100(TOP = OCR1A) or 1100(TOP = ICR1)
// WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)
↪  from TCCR1A
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* What to do when timer reaches the OCR1A value */
// toggle OC1A on each time when reaches the OCR1A
// which is reflected in PB1
// Output OC1A to toglle when reaches OCR1A -- COM1A[1:0] === 01
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1A0);
TCCR1A = TCCR1A & ~(1<<COM1A1);


//we count till OCR1A register value and toggle
// lets' count 10 pulses
OCR1A = 0x000a;

/* to count external event -we must connect source to T1 (PD5) */
// THE CLK IS CLOCKED FROM external source
// Falling edge of T1(PD5) -- CS1[2:0] === 110
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B & ~(1<<CS10);
TCCR1B = TCCR1B | (1<<CS11);
TCCR1B = TCCR1B | (1<<CS12);


// since for every rising edge the count increase
// so to reach 10 count, it would take 0xa / (frequency of input at T1 pin or PD5)
// we wave used 5kHz so it would take ==> 2ms to toggle as we have made OC1A toggle when overflows
↪  (by setting COMA[1:0])
// also we canuse TCNT1 as edge counter
```

6.7.3 Application I - Delay

```
// minimum delay being 4us -- choose like that - because, of the the delay for execution, - we get
↪  us if we use toggling of pins OC1A or OC1B
// use PRESCALAR OF 1 -- 4us - 4.096ms -- usage 4us - 4ms -- factor=0 -- CS1[2:0]=1
// use PRESCALAR OF 8 -- 4us - 32.768ms -- usage 5ms - 32ms -- factor=3 -- CS1[2:0]=2
// use PRESCALAR OF 64 -- 4us - 262.144ms -- usage 33ms - 260ms -- factor=6 -- CS0[2:0]=3
// use PRESCALAR OF 256 -- 16us - 1.048s -- usage 261ms - 1.048s -- factor=8 -- CS0[2:0]=4


/* TCNT1 starts from 0X0000 goes upto OCR1A or ICR1 and restarts */
// MOde of operation to Normal Mode -- WGM1[3:0] === 0100(TOP = OCR1A) or 1100(TOP = ICR1)
// WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)
↪  from TCCR1A
```

```
// we take TOP to be OCR1A for custom frequency
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* What to do when timer reaches the MAX(0xFFFF) value */
// nothing should be done on OC1A for delay
// nothing  -- COM1A[1:0] === 00
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A & ~(1<<COM1A1);
TCCR1A = TCCR1A & ~(1<<COM1A0);



if(delay_in_us <=3)
{
    // if delay_in_us <= 3us -- so we stop clock

    OCR1A = 0;
    // stop clcok
    // stop clcok-- CS1[2:0] === 000
    // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
    TCCR1B = TCCR1B & ~(1<<CS10);
    TCCR1B = TCCR1B & ~(1<<CS11);
    TCCR1B = TCCR1B & ~(1<<CS12);
}
else if((3 < delay_in_us)  && (delay_in_us <= 4000))
{
    OCR1A = ((delay_in_us * 16) >> 0) - 1;
    // start timer by setting the clock prescalar
    // SAME AS from I/O clock
    // same-- CS1[2:0] === 001
    // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
    TCCR1B = TCCR1B | (1<<CS10);
    TCCR1B = TCCR1B & ~(1<<CS11);
    TCCR1B = TCCR1B & ~(1<<CS12);
}
else if((4000 < delay_in_us)  && (delay_in_us <= 32000))
{
    OCR1A = ((delay_in_us * 16) >> 3) - 1;
    // start timer by setting the clock prescalar
    // divide by 8 from I/O clock
    // divide by 8 CS1[2:0] === 010
    // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
    TCCR1B = TCCR1B & ~(1<<CS10);
    TCCR1B = TCCR1B | (1<<CS11);
    TCCR1B = TCCR1B & ~(1<<CS12);
}
else if((32000 < delay_in_us)  && (delay_in_us <= 260000))
{
    OCR1A = ((delay_in_us * 16) >> 6) - 1;
    // start timer by setting the clock prescalar
    // divide by 64 from I/O clock
    // divide by 64 CS1[2:0] === 011
    // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
    TCCR1B = TCCR1B | (1<<CS10);
    TCCR1B = TCCR1B | (1<<CS11);
    TCCR1B = TCCR1B & ~(1<<CS12);
}
else if((260000 < delay_in_us)  && (delay_in_us <= 1000000))
{
    OCR1A = ((delay_in_us * 16) >> 8) - 1;
```

```
    // start timer by setting the clock prescalar
    // divide by 256 from I/O clock
    // divide by 256 CS1[2:0] === 100
    // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
    TCCR1B = TCCR1B & ~(1<<CS10);
    TCCR1B = TCCR1B & ~(1<<CS11);
    TCCR1B = TCCR1B | (1<<CS12);
}
else if(delay_in_us > 1000000)
{
    Timer1_asDelayIn_us(delay_in_us - 1000000);
    OCR1A = ((1000000 * 16) >> 8) - 1;
    // start timer by setting the clock prescalar
    // divide by 256 from I/O clock
    //divide by 256 CS1[2:0] === 100
    // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
    TCCR1B = TCCR1B & ~(1<<CS10);
    TCCR1B = TCCR1B & ~(1<<CS11);
    TCCR1B = TCCR1B | (1<<CS12);
}

// actual delaying - wait until delay happens
while((TIFR1 & 0x02) == 0x00); // checking OCF1A (compare match flag A) flag when match happns
// clearing the compare match flag so that we can further utilize
TIFR1 = TIFR1 | 0x02;
```

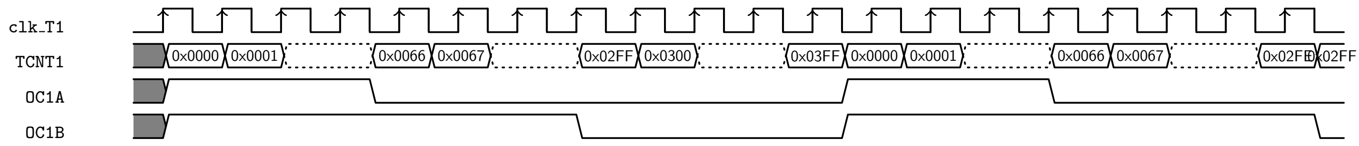
6.7.4 Fast PWM Mode

```
ISR(TIMER1_OVF_vect)
{
}
ISR(TIMER1_COMPA_vect)
{
}
ISR(TIMER1_COMPB_vect)
{
}
```

Non-Inverting PWM with TOP at MAX(0x00FF or 0x01FF or 0x03FF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR1A** and/or **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 0101 or 0110 or 0111 for Fast PWM Mode with TOP at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1A[1:0]*** and/or ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make outputs *OC1A* and/or *OC01* pins to generate PWM by comparing between **OCR1A** and/or **OCR1B** respectively. That is for Non-Inverting, ***COM1x[1:0]*** is written 10.
- Next, the duty cycle value is loaded into **OCR1A** and/or **OCR1B** register for *OC1A* and/or *OC1B* pins.
- Also, the ***OCIE1A*** and/or ***OCIE1B*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match and/or overflow.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 10% duty cycle *OC1A* and 75% duty cycle*OC1B* pins are shown assuming .
  - WGM1[3:0] === 0111 – TOP equals 0x03FF
  - 0x66 for OCR1A.
  - 0x2FF for OCR1B.

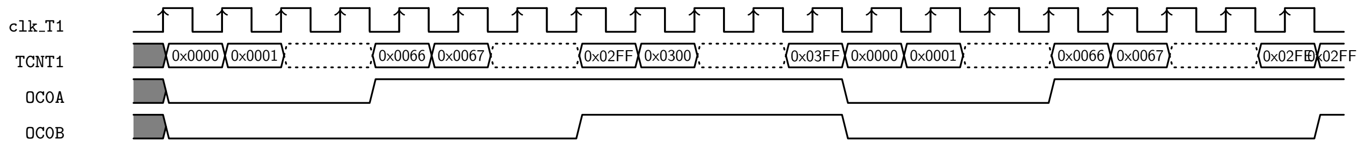



```
/* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
/* Mode of operation:
    WGM1[3:0] --> 0101 --        TOP--> 0X00FF
    WGM1[3:0] --> 0110 --        TOP--> 0x01FF
    WGM1[3:0] --> 0111 --        TOP--> 0x03FF
    WGM1[3:0] --> 1110 --        TOP--> ICR1
    WGM1[3:0] --> 1111 --        TOP--> OCR1A
*/
// we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 0111 for OCR1A as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

// here we set COM0A[1:0] as 10 for non-inverting
// here we set COM0B[1:0] as 10 for non-inverting

// which is reflected in PD6
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1A1);
TCCR1A = TCCR1A & ~(1<<COM1A0);

// which is reflected in PD65
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1B1);
TCCR1A = TCCR1A & ~(1<<COM1B0);

// Enable Interrupt when TOV1 overflows TOP - here 0x03FF
//  TOIE1 bit is enabled
TIMSK1 = TIMSK1 | (1<<TOIE1);

/* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
TIMSK1 = TIMSK1 | (1<<OCIE1A);
/* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
TIMSK1 = TIMSK1 | (1<<OCIE1B);


// Next we set values for OCR1A and OCR2B
// Since, TCNT1 goes till max(0x3FF), we can choose OCR1A and OCR1B to any value below max(0x03FF)
OCR1A = 102; // for 10% duty clcle
OCR1B = 767; // for 75% duty clcle


// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at MAX(0x00FF or 0x01FF or 0x03FF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR1A** and/or **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 0101 or 0110 or 0111 for Fast PWM Mode with TOP at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1A[1:0]*** and/or ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make outputs *OC1A* and/or *OC01* pins to generate PWM by comparing between **OCR1A** and/or **OCR1B** respectively. That is for Inverting, ***COM1x[1:0]*** is written 11.
- Next, the duty cycle value is loaded into **OCR1A** and/or **OCR1B** register for *OC1A* and/or *OC1B* pins.
- Also, the ***OCIE0A*** and/or ***OCIE0B*** bits of **TIMSK0** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match and/or overflow.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 10% duty cycle *OC1A* and 75% duty cycle*OC1B* pins are shown assuming .
  - WGM1[3:0] === 0111 – TOP equals 0x03FF
  - 0x66 for OCR1A.
  - 0x2FF for OCR1B.



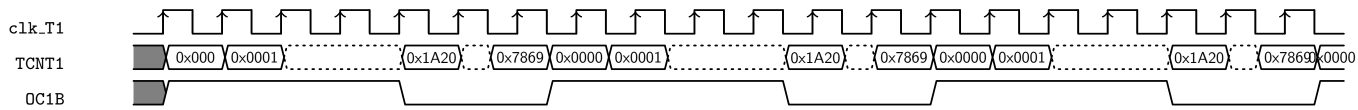

```
/* TCNT1 starts from OX0000 goes upto TOP and restarts from OX00*/
/* Mode of operation:
    WGM1[3:0] --> 0101 --        TOP--> OX00FF
    WGM1[3:0] --> 0110 --        TOP--> 0x01FF
    WGM1[3:0] --> 0111 --        TOP--> 0x03FF
    WGM1[3:0] --> 1110 --        TOP--> ICR1
    WGM1[3:0] --> 1111 --        TOP--> OCR1A
*/
// we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 0111 for OCR1A as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

// here we set COM0A[1:0] as 11 for inverting
// here we set COM0B[1:0] as 11 for inverting

// which is reflected in PD6
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1A1);
TCCR1A = TCCR1A | (1<<COM1A0);

// which is reflected in PD65
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1B1);
TCCR1A = TCCR1A | (1<<COM1B0);

// Enable Interrupt when TOV1 overflows TOP - here 0x03FF
//  TOIE1 bit is enabled
TIMSK1 = TIMSK1 | (1<<TOIE1);

/* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
TIMSK1 = TIMSK1 | (1<<OCIE1A);
/* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
TIMSK1 = TIMSK1 | (1<<OCIE1B);


// Next we set values for OCR1A and OCR2B
```

```
// Since, TCNT1 goes till max(0x3FF), we can choose OCR1A and OCR1B to any value below max(0x03FF)
OCR1A = 102; // for 10% duty clcle
OCR1B = 767; // for 75% duty clcle


// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

//enabled global interrupt
sei();
```

Non-Inverting PWM with TOP at OCR1A

Frequency is chosen by **OCR1A** and Duty cycle by **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 1110 or 1111 for Fast PWM Mode with **ICR1** or **OCR1A** at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make output *OC1B* pins to generate PWM by comparing between **TCNT1** and **OCR1B**. That is for Non-Inverting, ***COM1B[1:0]*** is written 10.
- The frequency of duty cycle is loaded into **OCR01A** register.
- Next, the duty cycle value is loaded into **OCR1B** register for *OC1B* bits.
- Also, the ***OCIE01B*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 37% duty cycle *OC1B* pins are shown assuming .
  - 0x7869 for OCR0A.
  - 0x1A20 for OCR0B.



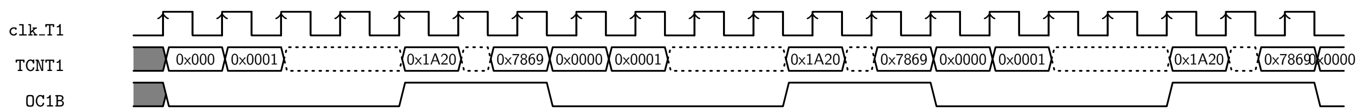

```
/* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
/* Mode of operation:
    WGM1[3:0] --> 0101 --        TOP--> 0X00FF
    WGM1[3:0] --> 0110 --        TOP--> 0x01FF
    WGM1[3:0] --> 0111 --        TOP--> 0x03FF
    WGM1[3:0] --> 1110 --        TOP--> ICR1
    WGM1[3:0] --> 1111 --        TOP--> OCR1A
*/
// we take OCR1A for custom frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B | (1<<WGM13);


// for non-inverting on  OC1B we use 10 for and COM1B[1:0]
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A & ~(1<<COM1B0);
TCCR1A = TCCR1A | (1<<COM1B1);
```

```
// Next we set values for OCR1A and OCR1B
// Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
OCR1A = 0x7869; // for freqeuncy
OCR1B = 0x1A20; // for pwm duty cylc

// Enable interrupt when count reaches the overflow value
TIMSK1 |= (1<<TOV1);

// Enable interrupt when count reaches the OCR1B
TIMSK1 |= (1<<OCF1B);

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

//e enabel globalinterrupt
sei();
```

Inverting PWM with TOP at OCR1A

Frequency is chosen by **OCR1A** and Duty cycle by **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 1110 or 1111 for Fast PWM Mode with **ICR1** or **OCR1A** at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make output *OC1B* pins to generate PWM by comparing between **TCNT1** and **OCR1B**. That is for Inverting, ***COM1B[1:0]*** is written 11.
- The frequency of duty cycle is loaded into **OCR01A** register.
- Next, the duty cycle value is loaded into **OCR1B** register for *OC1B* bits.
- Also, the ***OCIE01B*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 37% duty cycle(0x60) *OC1B* pins are shown assuming .
  - 0x7869 for OCR0A.
  - 0x1A20 for OCR0B.



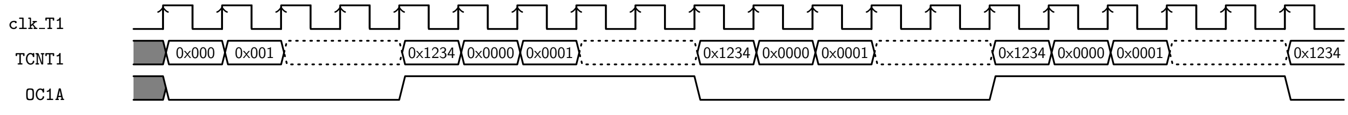

```
/* TCNT1 starts from OX0000 goes upto TOP and restarts from OX00*/
/* Mode of operation:
    WGM1[3:0] --> 0101 --        TOP--> OX00FF
    WGM1[3:0] --> 0110 --        TOP--> 0x01FF
    WGM1[3:0] --> 0111 --        TOP--> 0x03FF
    WGM1[3:0] --> 1110 --        TOP--> ICR1
    WGM1[3:0] --> 1111 --        TOP--> OCR1A
*/
// we take OCR1A for custom frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
```

```
TCCR1B = TCCR1B | (1<<WGM13);


// for ninverting on  OC1B we use 11 for and COM1B[1:0]
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1B0);
TCCR1A = TCCR1A | (1<<COM1B1);

// Next we set values for OCR1A and OCR1B
// Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
OCR1A = 0x7869; // for freqeuncy
OCR1B = 0x1A20; // for pwm duty cylc

// Enable interrupt when count reaches the overflow value
TIMSK1 |= (1<<TOV1);

// Enable interrupt when count reaches the OCR1B
TIMSK1 |= (1<<OCF1B);

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

//e enabel globalinterrupt
sei();
```

Toggling mode square Wave

Frequency is chosen by **OCR1A** register.

- First, ***WGM1[3:0]*** bits are configured as 1111 for Fast PWM Mode with **OCR1A** at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1A[1:0]*** bits of **TCCR1A** register are configured to make output *OC1A* pins to generate PWM by comparing between **OCR1A** and **TCNT1**. That is for Toggling square wave ***COM1A[1:0]*** is written 01.
- The frequency of duty cycle is loaded into **OCR1A** register.
- Also, the ***OCIE1A*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for squared wave on *OC1A* pins are shown assuming.
  - 0x1234 for OCR1A.




```
/* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
/* Mode of operation:
WGM1[3:0] --> 0101 --        TOP--> 0X00FF
WGM1[3:0] --> 0110 --        TOP--> 0x01FF
WGM1[3:0] --> 0111 --        TOP--> 0x03FF
WGM1[3:0] --> 1110 --        TOP--> ICR1
WGM1[3:0] --> 1111 --        TOP--> OCR1A
*/
// we take OCR1A for custom frequency
```

```
// choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B | (1<<WGM13);

// here we set COM1B[1:0] as 01 for toggling of OC1A
// which is reflected in PB1
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A & ~(1<<5);
TCCR1A = TCCR1A | (1<<4);

OCR1A = 0x1234; // for freqeuncy


// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

//enabled global interrupt
sei();
}
```

Application I - PWM generation

```
void Timer1_FastPWMGeneration(uint32_t on_time_us, uint32_t off_time_us)
{
        uint32_t total_time = on_time_us + off_time_us;

        /* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
        /* Mode of operation:
                WGM1[3:0] --> 0101 --        TOP--> 0X00FF
                WGM1[3:0] --> 0110 --        TOP--> 0x01FF
                WGM1[3:0] --> 0111 --        TOP--> 0x03FF
                WGM1[3:0] --> 1110 --        TOP--> ICR1
                WGM1[3:0] --> 1111 --        TOP--> OCR1A
        */

        // we take OCR1A for custom frequency and OCR1B for PWM on time(duty cycle)

        // choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);


        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B0);
        TCCR1A = TCCR1A | (1<<COM1B1);

        if(total_time <4)
        {
                // if total_time <= 3us -- so we stop clock

                OCR1A = 0;
                OCR1B = 0;
                // start timer by setting the clock prescalar
```

```
                //  use the same clock from I/O clock
                //  CS1[2:0] === 001
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B & ~(1<<0);
                TCCR1B = TCCR1B & ~(1<<1);
                TCCR1B = TCCR1B & ~(1<<2);
        }
        else if((3 < total_time)  && (total_time <= 4000))
        {
                OCR1A = ((total_time * 16) >> 0) - 1;
                OCR1B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS1[2:0] === 001
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B | (1<<0);
                TCCR1B = TCCR1B & ~(1<<1);
                TCCR1B = TCCR1B & ~(1<<2);
        }
        else if((4000 < total_time)  && (total_time <= 32000))
        {
                OCR1A = ((total_time * 16) >> 3) - 1;
                OCR1B = ((on_time_us * 16) >> 3) - 1;
                // start timer by setting the clock prescalar
                //  dived by 8 from I/O clock
                //  CS1[2:0] === 010
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B & ~(1<<0);
                TCCR1B = TCCR1B | (1<<1);
                TCCR1B = TCCR1B & ~(1<<2);
        }
        else if((32000 < total_time)  && (total_time <= 260000))
        {
                OCR1A = ((total_time * 16) >> 6) - 1;
                OCR1B = ((on_time_us * 16) >> 6) - 1;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS1[2:0] === 011
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B | (1<<0);
                TCCR1B = TCCR1B | (1<<1);
                TCCR1B = TCCR1B & ~(1<<2);

        }
        else if((260000 < total_time)  && (total_time <= 1000000))
        {
                OCR1A = ((total_time * 16) >> 8) - 1;
                OCR1B = ((on_time_us * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                //  divide by256 from I/O clock
                //  CS1[2:0] === 100
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B & ~(1<<0);
                TCCR1B = TCCR1B & ~(1<<1);
                TCCR1B = TCCR1B | (1<<2);

        }
        else if(total_time > 1000000)
        {
                // dont' cross more than 1s
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
```

```
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }

        // max time = 1S -- min freqency = 1 Hz
        //  min time = 4us -- max frequency = 250000 = 250khz
        Timer1_FastPWMGeneration(on_time_us, total_time_us - on_time_us);
}
```

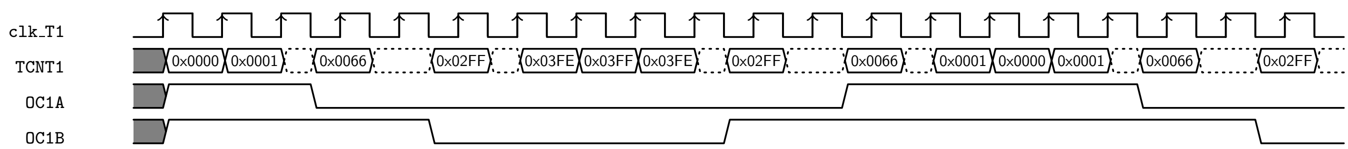
6.7.5 Phase Corrected PWM Mode

```
ISR(TIMER1_OVF_vect)
{
}
ISR(TIMER1_COMPA_vect)
{
}
ISR(TIMER1_COMPB_vect)
{
}
```

Non-Inverting PWM with TOP at MAX(0x00FF or 0x01FF or 0x03FF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR1A** and/or **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 0001 or 0010 or 0011 for Phase Corrected PWM Mode with TOP at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1A[1:0]*** and/or ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make outputs *OC1A* and/or *OC01* pins to generate PWM by comparing between **OCR1A** and/or **OCR1B** respectively. That is for Non-Inverting, ***COM1x[1:0]*** is written 10.
- Next, the duty cycle value is loaded into **OCR1A** and/or **OCR1B** register for *OC1A* and/or *OC1B* pins.
- Also, the ***OCIE1A*** and/or ***OCIE1B*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 10% duty cycle *OC1A* and 75% duty cycle*OC1B* pins are shown assuming .
  - WGM1[3:0] === 0011 – TOP equals 0x03FF
  - 0x66 for OCR1A.
  - 0x2FF for OCR1B.




```
/* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
/* Mode of operation:
    WGM1[3:0] --> 0001 --        TOP--> 0X00FF
    WGM1[3:0] --> 0010 --        TOP--> 0x01FF
    WGM1[3:0] --> 0011 --        TOP--> 0x03FF
    WGM1[3:0] --> 1010 --        TOP--> ICR1
    WGM1[3:0] --> 1011 --        TOP--> OCR1A
```

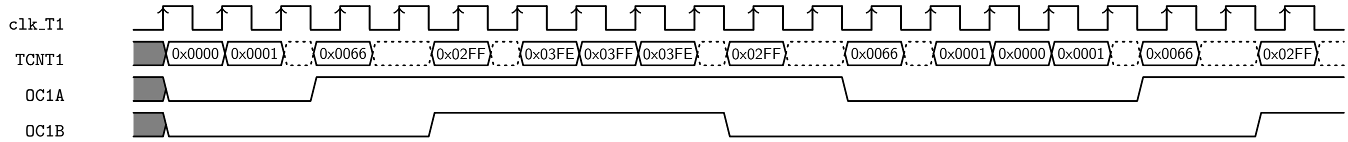
```
*/
// we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 0011 for 0x03FF as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* in timer0_phase_pwm_top_max, only two possiblites are there for COM0B[1:0] and COM0A[1:0] i.e)
↪  10(Inverting) and 11(Non- inverting) */

// here we set COM0A[1:0] as 10 for non-inverting
// here we set COM0B[1:0] as 10 for non-inverting

// which is reflected in PD6
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1A1);
TCCR1A = TCCR1A & ~(1<<COM1A0);

// which is reflected in PD65
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1B1);
TCCR1A = TCCR1A & ~(1<<COM1B0);

// Enable Interrupt when TOV1 overflows TOP - here 0x03FF
//  TOIE1 bit is enabled
TIMSK1 = TIMSK1 | (1<<TOIE1);

/* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
TIMSK1 = TIMSK1 | (1<<OCIE1A);
/* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
TIMSK1 = TIMSK1 | (1<<OCIE1B);

// Next we set values for OCR1A and OCR2B
// Since, TCNT1 goes till max(0x3FF), we can choose OCR1A and OCR1B to any value below max(0x03FF)
OCR1A = 102; // for 10% duty clcle
OCR1B = 767; // for 75% duty clcle

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at MAX(0x00FF or 0x01FF or 0x03FF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR1A** and/or **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 0001 or 0010 or 0011 for Phase Corrected PWM Mode with TOP at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1A[1:0]*** and/or ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make outputs *OC1A* and/or *OC01* pins to generate PWM by comparing between **OCR1A** and/or **OCR1B** respectively. That is for Inverting, ***COM1x[1:0]*** is written 11.
- Next, the duty cycle value is loaded into **OCR1A** and/or **OCR1B** register for *OC1A* and/or *OC1B* pins.
- Also, the ***OCIE1A*** and/or ***OCIE1B*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.

- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting *CS1[2:0]* bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 10% duty cycle *OC1A* and 75% duty cycle*OC1B* pins are shown assuming .
  - WGM1[3:0] === 0011 – TOP equals 0x03FF
  - 0x66 for OCR1A.
  - 0x2FF for OCR1B.



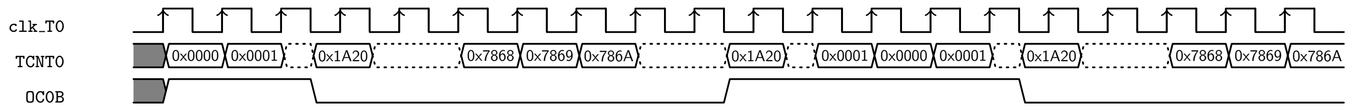

```
/* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
/* Mode of operation:
    WGM1[3:0] --> 0001 --        TOP--> 0X00FF
    WGM1[3:0] --> 0010 --        TOP--> 0x01FF
    WGM1[3:0] --> 0011 --        TOP--> 0x03FF
    WGM1[3:0] --> 1010 --        TOP--> ICR1
    WGM1[3:0] --> 1011 --        TOP--> OCR1A
*/
// we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 0011 for 0x03FF as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);

/* in timer0_phase_pwm_top_max, only two possiblites are there for COM0B[1:0] and COM0A[1:0] i.e)
↪  10(Inverting) and 11(Non- inverting) */

// here we set COM0A[1:0] as 11 for inverting
// here we set COM0B[1:0] as 11 for inverting

// which is reflected in PD6
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1A1);
TCCR1A = TCCR1A | (1<<COM1A0);

// which is reflected in PD65
// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
TCCR1A = TCCR1A | (1<<COM1B1);
TCCR1A = TCCR1A | (1<<COM1B0);

// Enable Interrupt when TOV1 overflows TOP - here 0x03FF
//  TOIE1 bit is enabled
TIMSK1 = TIMSK1 | (1<<TOIE1);

/* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
TIMSK1 = TIMSK1 | (1<<OCIE1A);
/* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
TIMSK1 = TIMSK1 | (1<<OCIE1B);

// Next we set values for OCR1A and OCR2B
// Since, TCNT1 goes till max(0x3FF), we can choose OCR1A and OCR1B to any value below max(0x03FF)
OCR1A = 102; // for 10% duty clcle
OCR1B = 767; // for 75% duty clcle

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
```

```
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);

//enabled global interrupt
sei();
```

Non-Inverting PWM with TOP at OCR1A

Frequency is chosen by **OCR1A** and Duty cycle by **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 1011 for Phase Corrected PWM Mode with OCR1A at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make output *OC1B* pins to generate PWM by comparing between **OCR1B** respectively. That is for Non-Inverting, ***COM1B[1:0]*** is written 10.
- The frequency of duty cycle is loaded into **OCR1A** register.
- Next, the duty cycle value is loaded into **OCR1B** register for *OC1B* bits.
- Also, the ***OCIE1B*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 37% duty cycle *OC1B* pins are shown assuming .
  - 0x7869 for OCR1A.
  - 0x1A20 for OCR1B.



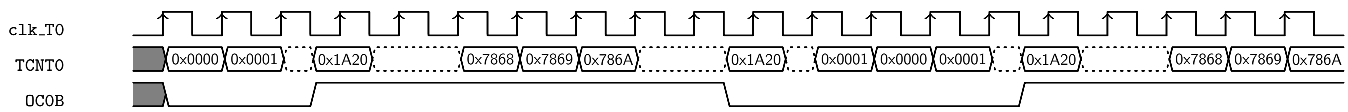

```
/* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
/* Mode of operation:
    WGM1[3:0] --> 0001 --
    TOP--> 0X00FF
    WGM1[3:0] --> 0010 --
    TOP--> 0x01FF
    WGM1[3:0] --> 0011 --
    TOP--> 0x03FF
    WGM1[3:0] --> 1010 --
    TOP--> ICR1
    WGM1[3:0] --> 1011 --
    TOP--> OCR1A
*/
// we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 1011 for OCR1A as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B | (1<<WGM13);

// here we set COM1A[1:0] as 10 for non-inverting
// which is reflected in PD5
// COM1B[1](bit5) from TCCR1A, COMOB[0](bit4) from TCCR1A
TCCR1A = TCCR1A | (1<<5);
TCCR1A = TCCR1A & ~(1<<4);

// Next we set values for OCR1A and OCR1B
// Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
OCR1A = 0x7869; // for freqeuncy
```

```
OCR1B = 0x1A20; // for pwm duty cylc

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);
//enabled global interrupt
sei();
```

Inverting PWM with TOP at OCR1A

Frequency is chosen by **OCR1A** and Duty cycle by **OCR1B** register.

- First, ***WGM1[3:0]*** bits are configured as 1011 for Phase Corrected PWM Mode with OCR1A at MAX in **TCCR1A** and **TCCR1B** registers.
- Next, ***COM1B[1:0]*** bits of **TCCR1A** register are configured to make output *OC1B* pins to generate PWM by comparing between **OCR1B** respectively. That is for Inverting, ***COM1B[1:0]*** is written 11.
- The frequency of duty cycle is loaded into **OCR1A** register.
- Next, the duty cycle value is loaded into **OCR1B** register for *OC1B* bits.
- Also, the ***OCIE1B*** bits of **TIMSK1** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS1[2:0]*** bit as needed prescalar in **TCR1B** register.
- The timing for PWM on 37% duty cycle *OC1B* pins are shown assuming .
  - 0x7869 for OCR1A.
  - 0x1A20 for OCR1B.




```
/* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
/* Mode of operation:
    WGM1[3:0] --> 0001 --
    TOP--> 0X00FF
    WGM1[3:0] --> 0010 --
    TOP--> 0x01FF
    WGM1[3:0] --> 0011 --
    TOP--> 0x03FF
    WGM1[3:0] --> 1010 --
    TOP--> ICR1
    WGM1[3:0] --> 1011 --
    TOP--> OCR1A
*/
// we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
// choose WGM1[3:0] --> 1011 for OCR1A as TOP for custom frequency
TCCR1A = TCCR1A | (1<<WGM10);
TCCR1A = TCCR1A | (1<<WGM11);
TCCR1B = TCCR1B & ~(1<<WGM12);
TCCR1B = TCCR1B | (1<<WGM13);

// here we set COM1A[1:0] as 11 for inverting
// which is reflected in PD5
// COM1B[1](bit5) from TCCR1A, COMOB[0](bit4) from TCCR1A
TCCR1A = TCCR1A | (1<<5);
```

```
TCCR1A = TCCR1A | (1<<4);

// Next we set values for OCR1A and OCR1B
// Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
OCR1A = 0x7869; // for freqeuncy
OCR1B = 0x1A20; // for pwm duty cylc

// start timer by setting the clock prescalar
// SAME AS from I/O clock
// same-- CS1[2:0] === 001
// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
TCCR1B = TCCR1B | (1<<CS10);
TCCR1B = TCCR1B & ~(1<<CS11);
TCCR1B = TCCR1B & ~(1<<CS12);
//enabled global interrupt
sei();
```

Application I - PWM generation

```
void Timer1_PhaseCorrectedPWMGeneration(uint32_t On_time_us, uint32_t Off_time_us)
{
    // Since, it is dual slope, the time would be doubled for one cylce, so we divide by 2
    uint32_t total_time = (On_time_us>>1) + (Off_time_us>>1);
    uint32_t on_time_us = On_time_us >> 1;

    /* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
    /* Mode of operation:
        WGM1[3:0] --> 0001 --
        TOP--> 0X00FF
        WGM1[3:0] --> 0010 --
        TOP--> 0x01FF
        WGM1[3:0] --> 0011 --
        TOP--> 0x03FF
        WGM1[3:0] --> 1010 --
        TOP--> ICR1
        WGM1[3:0] --> 1011 --
        TOP--> OCR1A
    */
    // we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
    // choose WGM1[3:0] --> 1011 for OCR1A as TOP for custom frequency
    TCCR1A = TCCR1A | (1<<WGM10);
    TCCR1A = TCCR1A | (1<<WGM11);
    TCCR1B = TCCR1B & ~(1<<WGM12);
    TCCR1B = TCCR1B | (1<<WGM13);

    // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
    TCCR1A = TCCR1A | (1<<COM1B0);
    TCCR1A = TCCR1A | (1<<COM1B1);

    if(total_time <4)
    {
        // if total_time <= 3us -- so we stop clock
        OCR1A = 0;
        OCR1B = 0;
        // start timer by setting the clock prescalar
        // use the same clock from I/O clock
        // CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<0);
        TCCR1B = TCCR1B & ~(1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
    }
```

```
    else if((3 < total_time) && (total_time <= 4000))
    {
        OCR1A = ((total_time * 16) >> 0) - 1;
        OCR1B = ((on_time_us * 16) >> 0) - 1;
        // start timer by setting the clock prescalar
        // use the same clock from I/O clock
        // CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<0);
        TCCR1B = TCCR1B & ~(1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
    }
    else if((4000 < total_time) && (total_time <= 32000))
    {
        OCR1A = ((total_time * 16) >> 3) - 1;
        OCR1B = ((on_time_us * 16) >> 3) - 1;
        // start timer by setting the clock prescalar
        // dived by 8 from I/O clock
        // CS1[2:0] === 010
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<0);
        TCCR1B = TCCR1B | (1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
    }
    else if((32000 < total_time) && (total_time <= 260000))
    {
        OCR1A = ((total_time * 16) >> 6) - 1;
        OCR1B = ((on_time_us * 16) >> 6) - 1;
        // start timer by setting the clock prescalar
        // dived by 64 from I/O clock
        // CS1[2:0] === 011
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<0);
        TCCR1B = TCCR1B | (1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
    }
    else if((260000 < total_time) && (total_time <= 1000000))
    {
        OCR1A = ((total_time * 16) >> 8) - 1;
        OCR1B = ((on_time_us * 16) >> 8) - 1;
        // start timer by setting the clock prescalar
        // divide by256 from I/O clock
        // CS1[2:0] === 100
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<0);
        TCCR1B = TCCR1B & ~(1<<1);
        TCCR1B = TCCR1B | (1<<2);
    }
    else if(total_time > 1000000)
    {
        // dont' cross more than 1s
    }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
    double total_time_us = (1000000.0/freqeuncy);
    double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
    if (on_time_us<1.0)
    {
        on_time_us = 1;
    }

    // max time = 8ms -- min freqency = 125 Hz
```

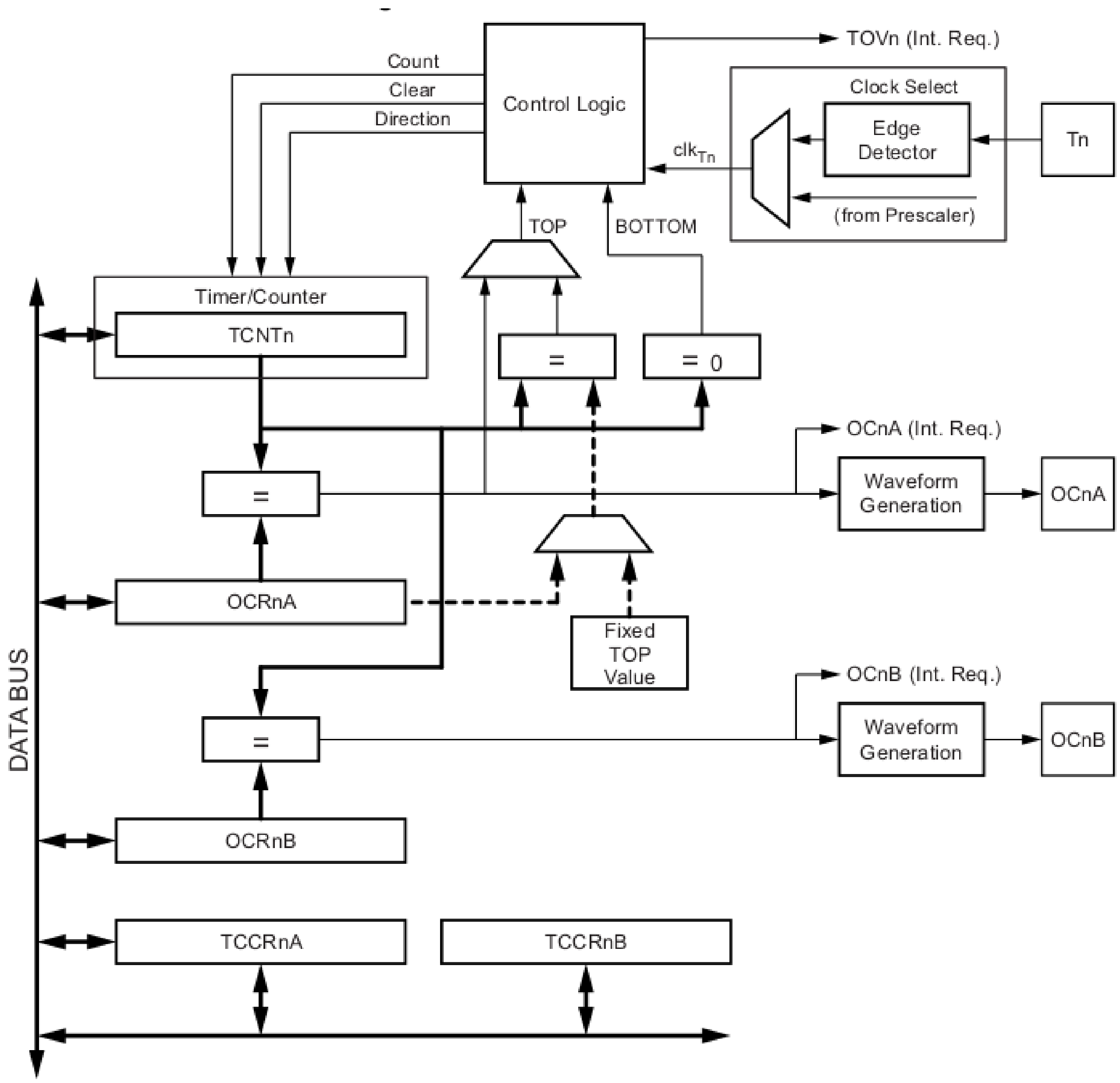
```
    //  min time = 8us -- max frequency = 250000 = 125khz
    Timer1_PhaseCorrectedPWMGeneration(on_time_us, total_time_us - on_time_us);
}
```

Timer/Counter 2

7.1 Features

- General purpose 8-bit Timer/Counter module.
- Two independent output compare units.
- Variable PWM.
- Three independent interrupt sources (TOV2, OCF2A, and OCF2B).
- Clear timer on compare match (auto reload)

7.2 Block Diagram



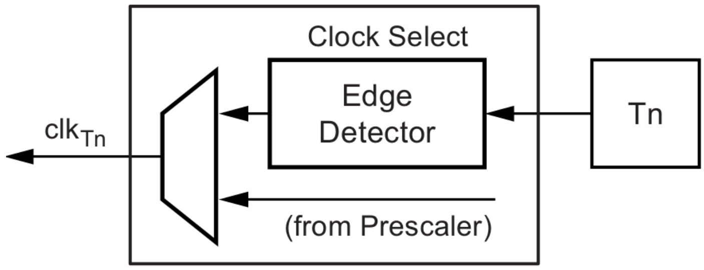
7.3 Terminologies and Registers

Parameter	Description
BOTTOM	counter reaches 0x00
MAX	ounter reaches 0xFF
TOP	counter reaches highest value (depends on mode of operation can be 0xFF, OCR2A).

Register - 8 bit	Name
TCNT2	Timer/Counter2 count value
TCCR2A	Timer/Counter2 Control Register A
TCCR2B	Timer/Counter2 Control Register B
OCBR2A	Output compare register A
OCBR2B	Output compare register B
TIFR2	Timer Interrupt Flag Register
TIMSK2	Timer interrupt Mask Register

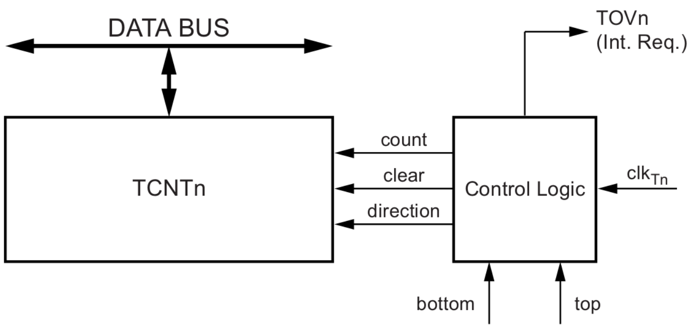
7.4 Timer/Counter2 Units

7.4.1 Clock Source/Select Unit



- The source for the Timer/Counter2 can be external or internal.
- External clock source is from *T2* pin.
- While Internal Clock source can be clocked via a prescalar.
- The output of this unit is the timer clock (clk_{T2}).
- It uses ***CS2[2:0]*** bits in **TCCR2B** register to select the source.

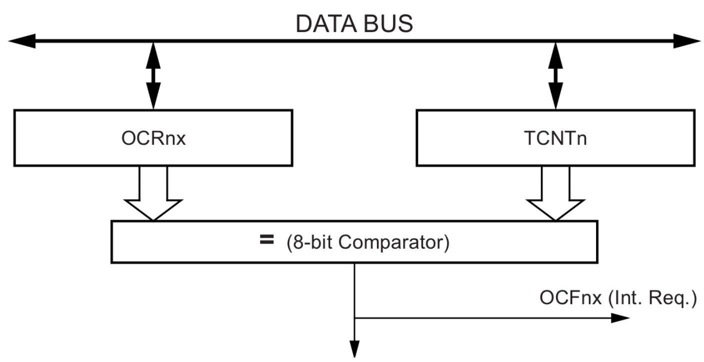
7.4.2 Counter Unit



Signal	Description
count	Increment or decrement **TCNT2** by 1
direction	Select between increment or decrement
clear	Clears **TCNT2** to 0x00
clk_{T2}	Timer/Counter2 clock
top	Signalize that **TCNT2** has maximum value
bottom	Signalize that **TCNT2** has minimum value(0x00)

- The main part of the 8-bit Timer/Counter is the programmable bi-directional counter.
- Depending the mode of operation the counter is cleared, incremented, or decremented at each timer clock (clk_{T2}).
- Counting sequence is determined by ***WGM2[1:0]*** bits of **TCCR2A** -Timer/Counter2 Control register A and ***WGM22*** bit of **TCCR2B** - Timer/Counter2 Control register B.
- The Timer/Counter2 Overflow flag ***TOV2*** is set and can generate interrupt according to the mode.

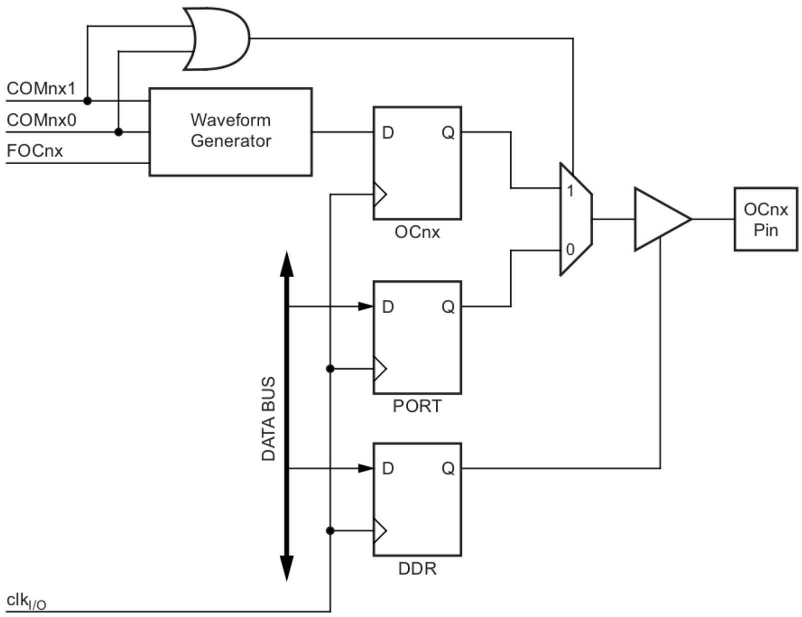
7.4.3 Output Compare Unit



- 8-bit comparator continuously compares **TCNT2** with both **OCR2A** and **OCR2B**.

- When **TCNT2** equals **OCR2A** or **OCR2B**, the comparator signals a match which will set the output compare flag at the next timer clock cycle.
- If interrupts are enabled, then output compare interrupt is generated.
- The waveform generator uses the match signal to generate an output according to operating mode set by the ***WGM2[2:0]*** bits and compare output mode ***COM2x[1:0]*** bits.

7.4.4 Compare Match Output Unit



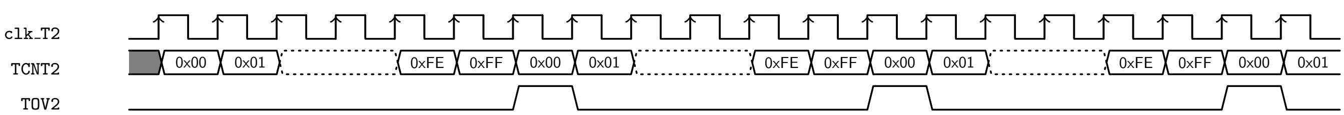
- This unit is used for changing the state of *OC2A* and *OC2B* pins by configuring the ***COM2x[1:0]*** bits.
- But, general I/O port function is overriiden by DDR reigster.

7.5 Modes of Operation

- The mode of operation can be defined by combination of waveform generation mode (***WGM2[2:0]***) and compare output mode(***COM2[1:0]***) bits.
- The waveform generation mode (***WGM2[2:0]***) bits affect the counting sequence.
- For non-PWM mode, ***COM2[1:0]*** bits control if the output should be set, cleared or toggled at a compare match.
- For PWM mode, ***COM2[1:0]*** bits control if the PWM generated should be inverted or non-inverted.

7.5.1 Normal Mode - Non-PWM Mode

- ***WGM2[2:0]*** $-->$ 000.
- Counter counts up and no counter clear.
- Overruns TOP(0XFF) and restarts from BOTTOM(0X00).
- ***TOV2*** Flag is only set when overrun.
- We have to clear ***TOV2*** flag inorder to have next running.
- But, if we use interrupt we don't need to clear it as interrupt automatically clear the ***TOV2*** flag.
- The timing can be seen below.

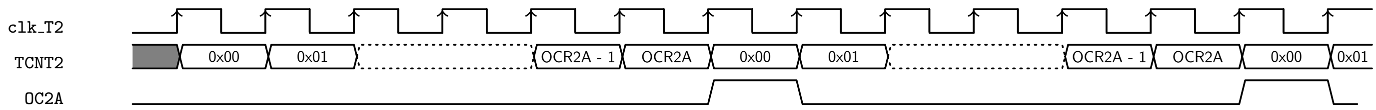


7.5.2 Clear Timer on Compare Match(CTC) Mode - Non-PWM Mode

- ***WGM2[2:0]*** $-->$ 010.
- Counter value clears when **TCNT2** reaches **OCR2A**.
- Interrupt can be generated each time **TCNT2** reaches **OCR2A** register value by ***OCF0A*** flag.
- When ***COM2A[1:0]*** == 01, the *OC2A* pin output can be set to toggle its match between **TCNT2** and **OCR2A** to generate waveform.
- The frequency of the waveform its

$$f_{OC2A} = \frac{f_{clkT2}}{2*N*(1+OCR2A)}$$

- Here N is prescalar factor and can be (1, 8, 64, 256, or 1024).

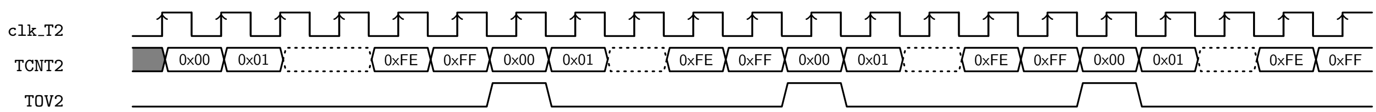


7.5.3 Fast PWM Mode

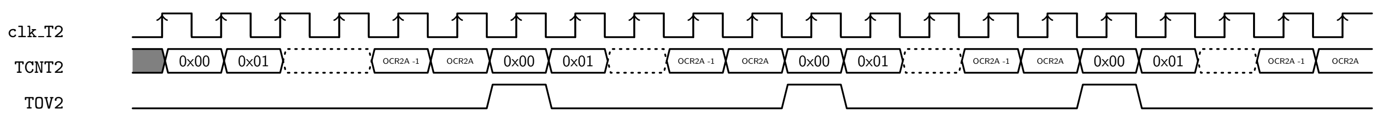
- ***WGM2[2:0]*** $-->$ 011 or 111.
- Power Regulation, Rectification, DAC applications.
- Single slope operations causing high frequency PWM waveform.
- Counter starts from BOTTOM to TOP and then restarts from BOTTOM.
- TOP is defined by
  - TOP == 0xFF if ***WGM2[2:0]*** $-->$ 011
  - TOP == **OCR2A** if ***WGM2[2:0]*** $-->$ 111
- When ***COM2A[1:0]*** == 01, the *OC2A* pin output can be set to toggle its match between **TCNT2** and TOP to generate waveform.
  - The above is possible only when ***WGM22*** bit is set.
  - And only on *OC2A* pin and not on *OC2B* pin.
- In Inverting Compare Mode ***COM2A[1:0]*** == 10 , the *OC2A* or *OC2B* pins is made 1 on compare match between **TCNT2** and TOP and made 0 on reaching BOTTOM.
- In Non-Inverting Compare Mode ***COM2A[1:0]*** == 11 , the *OC2A* or *OC2B* pins is made 0 on compare match between **TCNT2** and TOP and 1 made on reaching BOTTOM.
- The Timer/Counter overflow flag (***TOV2***) is set each time the counter reaches TOP.
- The PWM frequency is given by

$$f_{OC0xPWM} = \frac{f_{clkT2}}{N*256}$$

WGM[2:0] == 011



WGM[2:0] == 011

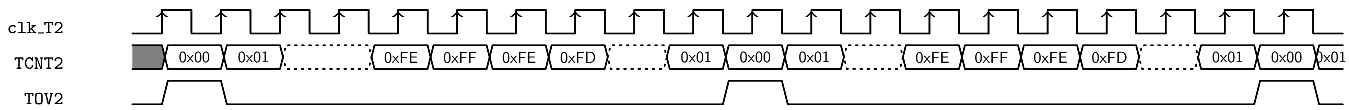


7.5.4 Phase Correct PWM Mode

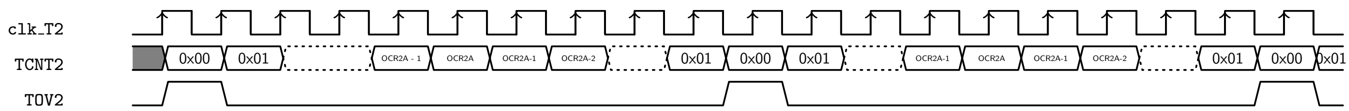
- ***WGM2[2:0]*** $-->$ 001 or 101.
- High resolution phase correct PWM.
- Motor control due to symmetric features
- Dual slope operations causing ower frequency PWM waveform.
- Counter starts from BOTTOM to TOP and then from TOP to BOTTOM.
- TOP is defined by
  - TOP == 0xFF if ***WGM2[2:0]*** $-->$ 001
  - TOP == **OCR2A** if ***WGM2[2:0]*** $-->$ 101
- When ***COM2A[1:0]*** == 01, the *OC2A* pin output can be set to toggle its match between **TCNT2** and TOP to generate waveform.
  - The above is possible only when ***WGM22*** bit is set.
  - And only on *OC2A* pin and not on *OC2B* pin.
- In Inverting Compare Mode ***COM2A[1:0]*** == 10 , the *OC2A* or *OC2B* pins is made 1 on compare match between **TCNT2** and TOP and made 0 on reaching BOTTOM.
- In Non-Inverting Compare Mode ***COM2A[1:0]*** == 11 , the *OC2A* or *OC2B* pins is made 0 on compare match between **TCNT2** and TOP and 1 made on reaching BOTTOM.
- The Timer/Counter overflow flag (***TOV2***) is set each time the counter reaches BOTTOM..
- The PWM frequency is given by

$$f_{OC0xPWM} = \frac{f_{clkT2}}{N*510}$$

WGM[2:0] == 001



WGM[2:0] == 101



7.6 Register Description

TCCR2A – Timer/Counter Control Register A

7	6	5	4	3	2	1	0
COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20

COM2B[1:0]	Non-PWM modes	Fast PWM	Phase Corrected PWM
00	No output @ *PD3 - OC2B* pin	No output @ *PD3 - OC2B*	No output @ *PD3 - OC2B*
01	Toggle *PD3 - OC2B* pin on compare Match.	Reserved	Reserved
10	Clear *PD3 - OC2B* pin on compare Match.	Clear *PD3 - OC2B* on compare match and set *PD3 - OC2B* at BOTTOM	Clear *PD3 - OC2B* on compare match when up-counting and set *PD3 - OC2B* on compare match when down-counting.
11	Set *PD3 - OC2B* pin on compare Match.	Set *PD3 - OC2B* on compare match and clear *PD3 - OC2B* at BOTTOM	Set *PD3 - OC2B* on compare match when up-counting and clear *PD3 - OC2B* on compare match when down-counting

COM2A[1:0]	Non-PWM modes	Fast PWM	Phase Corrected PWM
00	No output @ *PB3 - OC2A* pin	No output @ *PB3 - OC2A*	No output @ *PB3 - OC2A*
01	Toggle *PB3 - OC2A* pin on compare Match.	When WGM2[2] == 1, Toggle *PB3 - OC2A* pin on Compare match	Toggle *PB3 - OC2A* pin on Compare match
10	Clear *PB3 - OC2A* pin on compare Match.	Clear *PB3 - OC2A* on compare match and set *PB3 - OC2A* at BOTTOM	Clear *PB3 - OC2A* on compare match when up-counting and set *PB3 - OC2A* on compare match when down-counting.
11	Set *PB3 - OC2A* pin on compare Match.	Set *PB3 - OC2A* on compare match and clear *PB3 - OC2A* at BOTTOM	Set *PB3 - OC2A* on compare match when up-counting and clear *PB3 - OC2A* on compare match when down-counting

WGM2[2:0]	Mode of operation	TOP	TOV2 Flag set on
000	Normal	0xFF	MAX
001	PWM Phase Corrected	0xFF	BOTTOM
010	CTC	OCRA	MAX
011	Fast PWM	0xFF	MAX
101	PWM Phase Corrected	OCR2A	BOTTOM
111	Fast PWM	OCR2A	TOP

TCCR2B – Timer/Counter Control Register B

7	6	5	4	3	2	1	0
FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20

CS2[2:0]	**Description(Prescalar)**
000	No clock source(Timer/Counter Stopped)
001	$clk_{I/O}$ – no prescaling
010	$\frac{clk_{I/O}}{8}$
011	$\frac{clk_{I/O}}{64}$
100	$\frac{clk_{I/O}}{256}$
101	$\frac{clk_{I/O}}{1024}$
110	External clock source on *T2* pin. Clock on falling edge.
111	External clock source on *T2* pin. Clock on rising edge.

TIMSK2 – Timer/Counter Interrupt Mask Register

7	6	5	4	3	2	1	0
-	-	-	-	-	OCIE2B	OCIE2A	TOIE2

Enable interrupts for compare match between **TCNT2** and **OCR2A** or **TCNT2** and **OCR2B** or overflow in **TCNT2**.

TIFR2 – Timer/Counter 0 Interrupt Flag Register

7	6	5	4	3	2	1	0
-	-	-	-	-	OCIE2B	OCIE2A	TOIE2

FLag registers for interrupts on compare match between **TCNT2** and **OCR2A** or **TCNT2** and **OCR2B** or overflow in **TCNT2**.

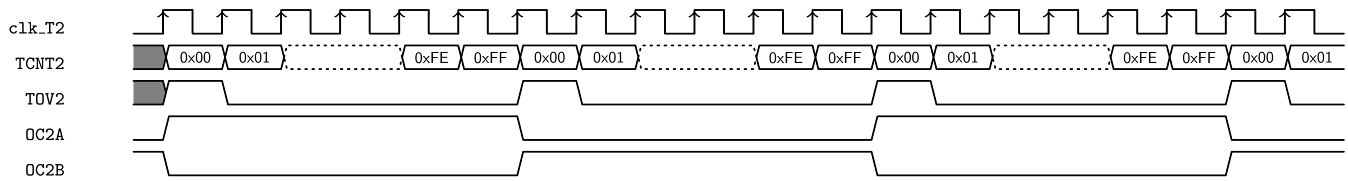
7.7 Configuring the Timer/Counter

7.7.1 Normal Mode

As Timer

$$ON\_TIME = \frac{max\_count}{\frac{F\_CPU}{PRESCALAR}}$$

- Depending on PRESCALR value, we get different ON_TIME.
- First, ***WGM2[2:0]*** bits are configured as 000 for Normal Mode in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2A[1:0]*** and/or ***COM2A[1:0]*** bits are configured to make outputs *OC2A* and/or *OC2B* pins to do nothing, set, clear or toggle in **TCCR2A** register.
- Next, Interrupt is Enabled by ***TOIE2*** (overflow enable) in **TIMSK2** reigster.
- Finally, Timer is started by setting prescalar in ***CS2[2:0]*** bits as needed prescalar of **TCR2B** reigster.
- Global Interrupt is enabled.
- A interrupt Service Routine for Timer2 overflow is Written.
- No need to clear the overflow flag as it is done by hardware.
- The timing when both pins *OC2A* and *OC2B* are made to toggle.



- The code can be seen below,

```
// MOde of operation to Normal Mode -- WGM2[2:0] === 000
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A & (~(1<<0) & ~(1<<1));
TCCR2B = TCCR2B & ~(1<<3);

/* What to do when timer reaches the MAX(0xFF) value */
// toggle OC2A and OC2B on each time when reaches the MAX(0xFF)
// which is reflected in PB3 and PD3

// Output OC2A to toglle when reaches MAX -- COM2A[1:0] === 01
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A & ~(1<<7);
TCCR2A = TCCR2A | (1<<6);

// Output OC2B to toglle when reaches MAX -- COM2B1:0] === 01
// COM2B[1](bit7) from TCCR2A, COM2B[0](bit6) from TCCR2A
TCCR2A = TCCR2A & ~(1<<5);
TCCR2A = TCCR2A | (1<<4);

//Enable Interrupt of OVERFLOW flag so that interrupt can be generated
TIMSK2 = TIMSK2 | (1<<0);

// start timer by setting the clock prescalar
// DIVIDE BY 8 from I/O clock
// DIVIDE BY 8-- CS2[2:0] === 010
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<1);
TCCR2B = TCCR2B & (~(1<<0) & ~(1<<2));

// enabling global interrupt
sei();
```

```
// SO ON TIME = max_count / (F_CPU / PRESCALAR)
// ON TIME = 0xFF / (16000000/8) = 128us
// since symmetric as toggling OFF TIME = 128us
// hence, we get a square wave of fequency 1 / 256us = 3.906kHz
```

```
ISR(TIMER2_OVF_vect)
{
    // do the thing when overflows.
}
```

Application I - Delay

```
/* TCNT2 starts from 0X00 goes upto 0XFF and restarts */
/* No possible use case as it just goes upto 0xFF and restarts */
// MOde of operation to Normal Mode -- WGM2[2:0] === 000
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A & (~(1<<0) & ~(1<<1));
TCCR2B = TCCR2B & ~(1<<3);

/* What to do when timer reaches the MAX(0xFF) value */
// nothing should be done on OC2A for delay
// nothing  -- COM2A[1:0] === 00
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A & ~(1<<7);
TCCR2A = TCCR2A & ~(1<<6);

/* The delay possible = 0xff / (F_CPU/prescalar) */
// lowest delay = 0xff / (16000000 / 1) = 16us
// when prescalar == 8 --> delay = 0xff / (16000000 / 8) = 128us
// when prescalar == 64 --> delay = 0xff / (16000000 / 64) = 1.024ms
// when prescalar == 256 --> delay = 0xff / (16000000 / 256) = 4.096ms
// highest delay possible = 0xff / (16000000 / 1024) = 16.38ms

// start timer by setting the clock prescalar
// DIVIDE BY 8 use the same clock from I/O clock
// DIVIDE BY 8-- CS2[2:0] === 010
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B & ~(1<<0);
TCCR2B = TCCR2B | (1<<1);
TCCR2B = TCCR2B & ~(1<<2);


// actual delaying - wait until delay happens
while((TIFR2 & 0x01) == 0x00); // checking overflow flag when overflow happns
// clearing the overflag so that we can further utilize
TIFR2 = TIFR2 | 0x01;
```

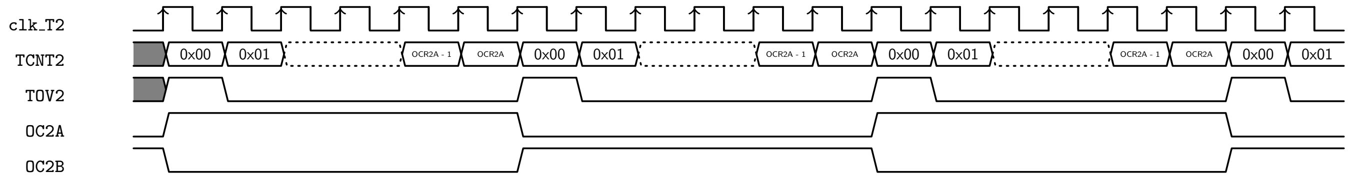
7.7.2 CTC Mode

As Timer

$$ON\_TIME = \frac{1+OCR2A}{\frac{F\_CPU}{PRESCALAR}}$$

- Depending on **OCR2A** register and PRESCALR value, we get different ON_TIME.
- First, ***WGM2[2:0]*** bits are configured as 010 for CTC Mode in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2A[1:0]*** and/or ***COM2B[1:0]*** bits are configured to make outputs *OC2A* and/or *OC2B* pins to do nothing, set, clear or toggle in **TCCR2A** register.
- Next, Interrupt is Enabled by ***OCIE01A*** (utput compare on match on **OCR2A** register enable) in **TIMSK2** reigster.

- Finally, Timer is started by setting prescalar in *CS2[2:0]* bits as needed prescalar of **TCR2B** reigster.
- Global Interrupt is enabled.
- A interrupt Service Routine for TIMER2 Compare is Written.
- No need to clear the overflow flag as it is done by hardware.
- The timing when both pins *OC0n* are made to toggle.



- The code can be seen below,

```
// MOde of operation to CTC Mode -- WGM2[2:0] === 010
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A & ~(1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B & ~(1<<3);

/* What to do when timer reaches the OCR2A */
// toggle OC2A on each time when reaches the OCR2A
// which is reflected in PB3
// Output OC2A to togle when reaches MAX -- COM2A[1:0] === 01
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A & ~(1<<7);
TCCR2A = TCCR2A | (1<<6);

// Output OC2B to togle when reaches MAX -- COM2B1:0] === 01
// COM2B[1](bit7) from TCCR2A, COM2B[0](bit6) from TCCR2A
TCCR2A = TCCR2A & ~(1<<5);
TCCR2A = TCCR2A | (1<<4);


// Enable Interrupt when counter matches OCR2A Rgister
//  OCIE2A bit is enabled
TIMSK2 = TIMSK2 | (1<<1);


// setting the value till the counter should reach in OCR2A
// for toggling of OC2A pin
OCR2A = 0x32;

// start timer by setting the clock prescalar
// DIVIDE BY 8 from I/O clock
// DIVIDE BY 8-- CS2[2:0] === 010
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<1);
TCCR2B = TCCR2B & (~(1<<0) & ~(1<<2));

// enabling global interrupt
sei();
// SO ON TIME = (1 + OCR2A) / (F_CPU / PRESCALAR)
// ON TIME = 0X32 / (16000000/8) = 25.5us
// since symmetric as toggling OFF TIME = 25.5us
// hence, we get a square wave of fequency 1 / 50us = 20kHz
```

```
ISR(TIMER2_COMPA_vect)
{
    // do the thing when compare match between TCNT2 matches OCR2A.
}
```

Application I - Delay in ms

```
// minimum delay being 4us -- choose like that
// use PRESCALAR OF 1 -- 3us - 16us -- usage 3us - 16us -- factor=0 -- CS2[2:0]=1
// use PRESCALAR OF 8 -- 3us - 128us -- usage 17us - 128us -- factor=3 -- CS2[2:0]=2
// use PRESCALAR OF 64 -- 4us - 1.024ms -- usage 129us - 1024us -- factor=6 -- CS2[2:0]=3
// use PRESCALAR OF 256 -- 16us - 4.096ms -- usage 1025us - 4096us -- factor=8 -- CS2[2:0]=4

// MOde of operation to ctc Mode -- WGM2[2:0] === 010
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A & ~(1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B & ~(1<<3);

while(delayInMs--)
{
    // for 1ms delay
    OCR2A = 249;
    // start timer by setting the clock prescalar
    //  dived by 64 from I/O clock
    //  CS2[2:0] === 011
    // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
    TCCR2B = TCCR2B | (1<<0);
    TCCR2B = TCCR2B | (1<<1);
    TCCR2B = TCCR2B & ~(1<<2);

    // actual delaying - wait until delay happens
    while((TIFR2 & 0x02) == 0x00); // checking OCF0A (compare match flag A) flag when match happns
    // clearing the compare match flag so that we can further utilize
    TIFR2 = TIFR2 | 0x02;
}
```

7.7.3 Fast PWM Mode

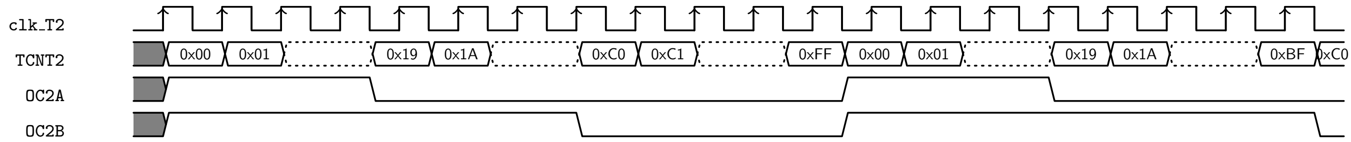
```
ISR(TIMER2_OVF_vect)
{
}
ISR(TIMER2_COMPA_vect)
{
}
ISR(TIMER2_COMPB_vect)
{
}
```

Non-Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR2A** and/or **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 011 for Fast PWM Mode with TOP at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2A[1:0]*** and/or ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make outputs *OC2A* and/or *OC2B* pins to generate PWM by comparing between **OCR2A** and/or **OCR2B** respectively. That is for Non-Inverting, ***COM2x[1:0]*** is written 10.
- Next, the duty cycle value is loaded into **OCR2A** and/or **OCR2B** register for *OC2A* and/or *OC2B* pins.
- Also, the ***OCIE2A*** and/or ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.

- The timing for PWM on 10% duty cycle *OC2A* and 75% duty cycle*OC2B* pins are shown assuming .
  - 0x19 for OCR2A.
  - 0xC0 for OCR2B.



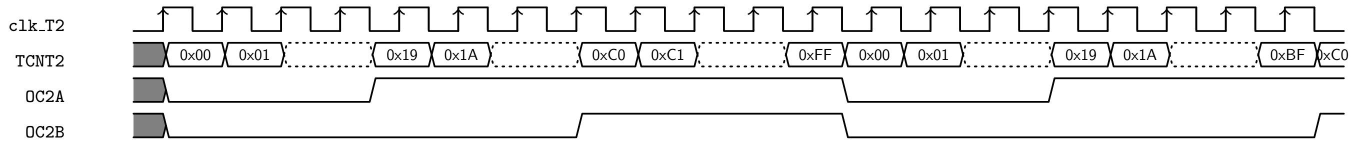

```
// MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 011
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B & ~(1<<3);

// here we set COM2A[1:0] as 10 for non-inverting
// here we set COM2B[1:0] as 10 for non-inverting

// which is reflected in PB3
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A | (1<<7);
TCCR2A = TCCR2A & ~(1<<6);

// which is reflected in PB35
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A & ~(1<<4);

// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV2 bit is enabled
TIMSK2 = TIMSK2 | (1<<0);

/* we use OCF0A flag - which is set at every time TCN0 reaches OCR2A
here we clear led(PC1),  so that we obtain the PWM when TCN0 reaches OCR2A*/
TIMSK2 = TIMSK2 | (1<<1);
/* we use OCF0B flag - which is set at every time TCN0 reaches OCR2B
here we clear led(PC2),  so that we obtain the PWM when TCN0 reaches OCR2B*/
TIMSK2 = TIMSK2 | (1<<2);


// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till max(0xFF), we can choose OCR2A and OCR2B to any value below max(0xFFF)
OCR2A = 0x19; // for 10% duty clcle
OCR2B = 0xC0; // for 75% duty clcle


// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR2A** and/or **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 011 for Fast PWM Mode with TOP at MAX in **TCCR2A** and **TCCR2B** registers.

- Next, ***COM2A[1:0]*** and/or ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make outputs *OC2A* and/or *OC2B* pins to generate PWM by comparing between **OCR2A** and/or **OCR2B** respectively. That is for Inverting, ***COM2x[1:0]*** is written 11.
- Next, the duty cycle value is loaded into **OCR2A** and/or **OCR2B** register for *OC2A* and/or *OC2B* bits.
- Also, the ***OCIE2A*** and/or ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for PWM on 10% duty cycle *OC2A* and 75% duty cycle *OC2B* pins are shown assuming .
  - 0x19 for OCR2A.
  - 0xC0 for OCR2B.



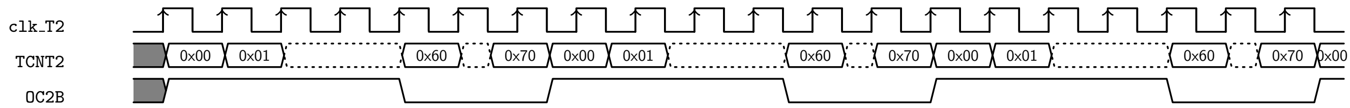

```
// MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 011
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B & ~(1<<3);

// here we set COM2A[1:0] as 11 for inverting
// here we set COM2B[1:0] as 11 for inverting

// which is reflected in PB3
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A | (1<<7);
TCCR2A = TCCR2A | (1<<6);

// which is reflected in PB35
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A | (1<<4);

// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV2 bit is enabled
TIMSK2 = TIMSK2 | (1<<0);

/* we use OCF0A flag - which is set at every time TCN0 reaches OCR2A
    here we clear led(PC1),  so that we obtain the PWM when TCN0 reaches OCR2A*/
TIMSK2 = TIMSK2 | (1<<1);
/* we use OCF0B flag - which is set at every time TCN0 reaches OCR2B
    here we clear led(PC2),  so that we obtain the PWM when TCN0 reaches OCR2B*/
TIMSK2 = TIMSK2 | (1<<2);


// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till max(0xFF), we can choose OCR2A and OCR2B to any value below max(0xFFF)
OCR2A = 0x19; // for 10% duty clcle
OCR2B = 0xC0; // for 75% duty clcle


// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
```

```
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Non-Inverting PWM with TOP at OCR2A

Frequency is chosen by **OCR2A** and Duty cycle by **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 111 for Fast PWM Mode with **OCR2A** at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make output *OC2B* pins to generate PWM by comparing between **TCNT2** and **OCR2B**. That is for Non-Inverting, ***COM2B[1:0]*** is written 10.
- The frequency of duty cycle is loaded into **OCR2A** register.
- Next, the duty cycle value is loaded into **OCR2B** register for *OC2B* bits.
- Also, the ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for PWM on 85% duty cycle(0x60) *OC2B* pins are shown assuming .
  - 0x70 for OCR2A.
  - 0x60 for OCR2B.

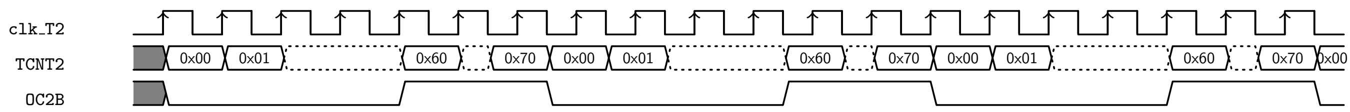



```
// MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B | (1<<3);

// here we set COM2B[1:0] as 10 for non-inverting
// which is reflected in PD3
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A & ~(1<<4);

// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
OCR2A = 0x70; // for freqeuncy
OCR2B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at OCR2A

Frequency is chosen by **OCR2A** and Duty cycle by **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 111 for Fast PWM Mode with **OCR2A** at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make output *OC2B* pins to generate PWM by comparing between **TCNT2** and **OCR2B**. That is for Inverting, ***COM2B[1:0]*** is written 11.
- The frequency of duty cycle is loaded into **OCR2A** register.
- Next, the duty cycle value is loaded into **OCR2B** register for *OC2B* bits.
- Also, the ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for PWM on 85% duty cycle *OC2B* pins are shown assuming .
  - 0x70 for OCR2A.
  - 0x60 for OCR2B.



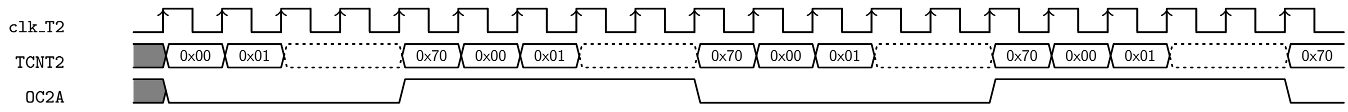

```
// MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B | (1<<3);

// here we set COM2B[1:0] as 11 for inverting
// which is reflected in PD3
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A | (1<<4);

// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
OCR2A = 0x70; // for freqeuncy
OCR2B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Toggling mode square Wave

Frequency is chosen by **OCR2A** register.

- First, ***WGM2[2:0]*** bits are configured as 111 for Fast PWM Mode with OCR2A at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2A[1:0]*** bits of **TCCR2A** register are configured to make output *OC2A* pins to generate PWM by comparing between **OCR2A**. That is for Toggling square wave ***COM2A[1:0]*** is written 01.
- The frequency of duty cycle is loaded into **OCR2A** register.

- Also, the ***OCIE2A*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for squared wave on *OC2A* pins are shown assuming.
  - 0x70 for OCR2A.




```
// MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B | (1<<3);

// here we set COM2B[1:0] as 01 for toggling of OC2A
// which is reflected in PB3
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A & ~(1<<7);
TCCR2A = TCCR2A | (1<<6);

// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
OCR2A = 0x70; // for freqeuncy

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Application I - PWM generation

```
void Timer2_FastPWMGeneration(uint32_t on_time_us, uint32_t off_time_us)
{
        uint32_t total_time = on_time_us + off_time_us;

        // MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B | (1<<3);

        // which is reflected in PD3
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A & ~(1<<4);

        if(total_time <=3)
        {
                // if total_time <= 3us -- so we stop clock

                OCR2A = 0;
                // start timer by setting the clock prescalar
```

```
                //  use the same clock from I/O clock
                //  CS2[2:0] === 001
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((3 < total_time)  && (total_time <= 16))
        {
                OCR2A = ((total_time * 16) >> 0) - 1;
                OCR2B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS2[2:0] === 001
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B | (1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((16 < total_time)  && (total_time <= 128))
        {
                OCR2A = ((total_time * 16) >> 3) - 1;
                OCR2B = ((on_time_us * 16) >> 3) - 1;
                // start timer by setting the clock prescalar
                //  dived by 8 from I/O clock
                //  CS2[2:0] === 010
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B | (1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((128 < total_time)  && (total_time <= 1024))
        {
                OCR2A = ((total_time * 16) >> 6) - 1;
                OCR2B = ((on_time_us * 16) >> 6) - 1;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS2[2:0] === 011
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B | (1<<0);
                TCCR2B = TCCR2B | (1<<1);
                TCCR2B = TCCR2B & ~(1<<2);

        }
        else if((1024 < total_time)  && (total_time <= 4096))
        {
                OCR2A = ((total_time * 16) >> 8) - 1;
                OCR2B = ((on_time_us * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                //  divide by256 from I/O clock
                //  CS2[2:0] === 100
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B | (1<<2);

        }
        else if(total_time > 4096)
        {
                // dont' cross more than 4.096ms
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
```

```
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }

        // max time = 4ms -- min freqency = 250 Hz
        //  min time = 4us -- max frequency = 250000 = 250khz
        Timer2_FastPWMGeneration(on_time_us, total_time_us - on_time_us);
}
```

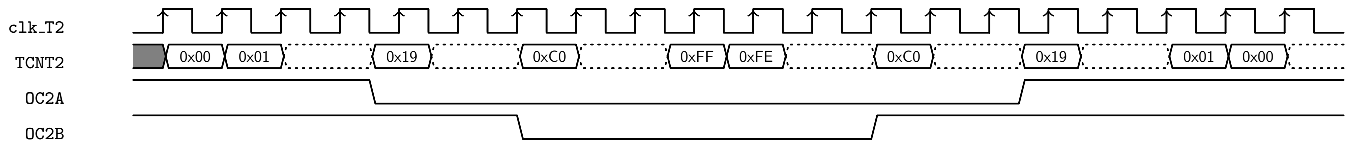
7.7.4 Phase Corrected PWM Mode

```
ISR(TIMER2_OVF_vect)
{
}
ISR(TIMER2_COMPA_vect)
{
}
ISR(TIMER2_COMPB_vect)
{
}
```

Non-Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR2A** and/or **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 001 for Phase Corrected PWM Mode with TOP at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2A[1:0]*** and/or ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make outputs *OC2A* and/or *OC2B* pins to generate PWM by comparing between **OCR2A** and/or **OCR2B** respectively. That is for Non-Inverting, ***COM2x[1:0]*** is written 10.
- Next, the duty cycle value is loaded into **OCR2A** and/or **OCR2B** register for *OC2A* and/or *OC2B* bits.
- Also, the ***OCIE2A*** and/or ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for PWM on 10% duty cycle *OC2A* and 75% duty cycle*OC2B* pins are shown assuming .
  - 0x19 for OCR2A.
  - 0xC0 for OCR2B.



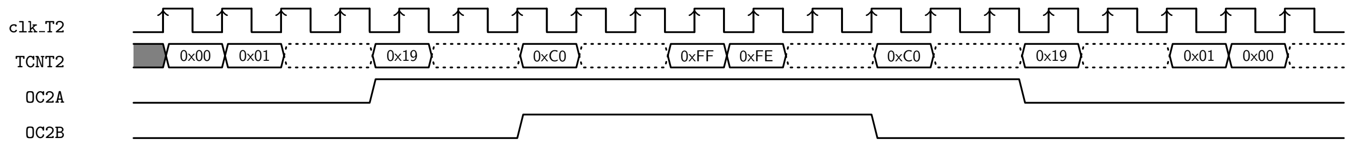

```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 001
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<3);

/* in TIMER2_phase_pwm_top_max, only two possiblites are there for COM2B[1:0] and COM2A[1:0] i.e)
↪  10(Inverting) and 11(Non- inverting) */
```

```
// here we set COM2A[1:0] as 10 for non-inverting
// here we set COM2B[1:0] as 10 for non-inverting

// which is reflected in PB3
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A | (1<<7);
TCCR2A = TCCR2A & ~(1<<6);

// which is reflected in PB35
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A & ~(1<<4);

/* we use overflow flag -- which is set at every time TCN0 reaches TOP here 0xFF
here, we toggle an led(PC0) at every overflow interrupt - this led(PC0) would give the frequency
↪  of PWM being generated -- done by PINC = PINC | 0X01;
Also, we set the other leds(PC1 and PC2) so that they are make one when TCN0 reaches 0x00 */
// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV2 bit is enabled
TIMSK2 = TIMSK2 | (1<<0);


// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till max(0xFF), we can choose OCR2A and OCR2B to any value below max(0xFFF)
OCR2A = 0x19; // for 10% duty clcle
OCR2B = 0xC0; // for 75% duty clcle

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at MAX(0xFF)

Frequency is chosen by PRESCALAR and Duty cycle by **OCR2A** and/or **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 001 for Phase Corrected PWM Mode with TOP at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2A[1:0]*** and/or ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make outputs *OC2A* and/or *OC2B* pins to generate PWM by comparing between **OCR2A** and/or **OCR2B** respectively. That is for Inverting, ***COM2x[1:0]*** is written 11.
- Next, the duty cycle value is loaded into **OCR2A** and/or **OCR2B** register for *OC2A* and/or *OC2B* bits.
- Also, the ***OCIE2A*** and/or ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for PWM on 10% duty cycle *OC2A* and 75% duty cycle*OC2B* pins are shown assuming .
  - 0x19 for OCR2A.
  - 0xC0 for OCR2B.



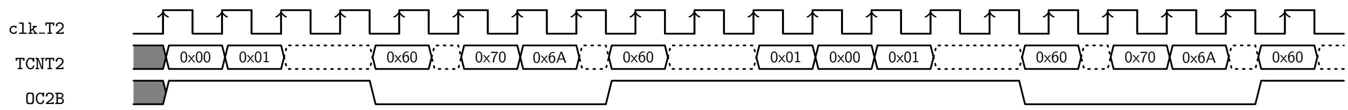

```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 001
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<3);

/* in TIMER2_phase_pwm_top_max, only two possiblites are there for COM2B[1:0] and COM2A[1:0] i.e)
↪  10(Inverting) and 11(Non- inverting) */

// here we set COM2A[1:0] as 11 for inverting
// here we set COM2B[1:0] as 11 for inverting

// which is reflected in PB3
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A | (1<<7);
TCCR2A = TCCR2A | (1<<6);

// which is reflected in PB35
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A | (1<<4);

/* we use overflow flag -- which is set at every time TCN0 reaches TOP here 0xFF
here, we toggle an led(PC0) at every overflow interrupt - this led(PC0) would give the frequency
↪  of PWM being generated -- done by PINC = PINC | 0X01;
Also, we set the other leds(PC1 and PC2) so that they are make one when TCN0 reaches 0x00 */
// Enable Interrupt when TCN0 overflows TOP - here 0xFF
//  TOV2 bit is enabled
TIMSK2 = TIMSK2 | (1<<0);


// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till max(0xFF), we can choose OCR2A and OCR2B to any value below max(0xFFF)
OCR2A = 0x19; // for 10% duty clcle
OCR2B = 0xC0; // for 75% duty clcle

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Non-Inverting PWM with TOP at OCR2A

Frequency is chosen by **OCR2A** and Duty cycle by **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 101 for Phase Corrected PWM Mode with OCR2A at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make output *OC2B* pins to generate PWM by comparing between **OCR2B** respectively. That is for Non-Inverting, ***COM2B[1:0]*** is written 10.
- The frequency of duty cycle is loaded into **OCR2A** register.
- Next, the duty cycle value is loaded into **OCR2B** register for *OC2B* bits.

- Also, the ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for PWM on 85% duty cycle(0x60) *OC2B* pins are shown assuming .
  - 0x70 for OCR2A.
  - 0x60 for OCR2B.

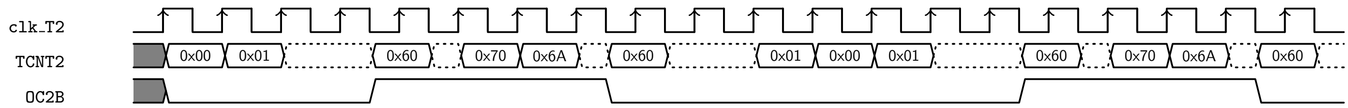



```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 101
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A & ~(1<<1);
TCCR2B = TCCR2B | (1<<3);

// here we set COM2A[1:0] as 10 for non-inverting
// which is reflected in PD3
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A & ~(1<<4);

// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
OCR2A = 0x70; // for freqeuncy
OCR2B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Inverting PWM with TOP at OCR2A

Frequency is chosen by **OCR2A** and Duty cycle by **OCR2B** register.

- First, ***WGM2[2:0]*** bits are configured as 101 for Phase Corrected PWM Mode with OCR2A at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2B[1:0]*** bits of **TCCR2A** register are configured to make output *OC2B* pins to generate PWM by comparing between **OCR2B** respectively. That is for Inverting, ***COM2B[1:0]*** is written 11.
- The frequency of duty cycle is loaded into **OCR2A** register.
- Next, the duty cycle value is loaded into **OCR2B** register for *OC2B* bits.
- Also, the ***OCIE2B*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for PWM on 85% duty cycle(0x60) *OC2B* pins are shown assuming .
  - 0x70 for OCR2A.
  - 0x60 for OCR2B.

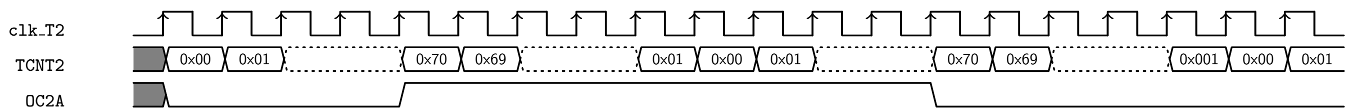



```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 101
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A & ~(1<<1);
TCCR2B = TCCR2B | (1<<3);

// here we set COM2A[1:0] as 11 for inverting
// which is reflected in PD3
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A | (1<<4);

// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
OCR2A = 0x70; // for freqeuncy
OCR2B = 0x60; // for pwm duty cylc

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Toggling mode square Wave

Frequency is chosen by **OCR2A** register.

- First, ***WGM2[2:0]*** bits are configured as 101 for Phase Corrected PWM Mode with OCR2A at MAX in **TCCR2A** and **TCCR2B** registers.
- Next, ***COM2A[1:0]*** bits of **TCCR2A** register are configured to make output *OC2A* pins to generate PWM by comparing between **OCR2A**. That is for Toggling square wave ***COM2A[1:0]*** is written 01.
- The frequency of duty cycle is loaded into **OCR2A** register.
- Also, the ***OCIE2A*** bits of **TIMSK2** register are enabled for Output Compare Interupts if needed.
- The interrupt Service routine is written if needed for compare match.
- Finally, Timer is started by setting ***CS2[2:0]*** bit as needed prescalar in **TCR2B** register.
- The timing for squared wave on *OC2A* pins are shown assuming.
  - 0x70 for OCR2A.




```
// MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 101
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A & ~(1<<1);
TCCR2B = TCCR2B | (1<<3);

// here we set COM2B[1:0] as 01 for toggling of OC2A
```

```
// which is reflected in PB3
// COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
TCCR2A = TCCR2A & ~(1<<7);
TCCR2A = TCCR2A | (1<<6);

// Next we set values for OCR2A and OCR2B
// Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
OCR2A = 0x70; // for freqeuncy

// start the timer by selecting the prescalr
//  use the same clock from I/O clock
//  CS2[2:0] === 001
// CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
TCCR2B = TCCR2B | (1<<0);
TCCR2B = TCCR2B & ~(1<<1);
TCCR2B = TCCR2B & ~(1<<2);

//enabled global interrupt
sei();
```

Application I - PWM generation

```
void Timer2_PhaseCorrectedPWMGeneration(uint32_t On_time_us, uint32_t Off_time_us)
{
        // Since, it is dual slope, the time would be doubled for one cylce, so we divide by 2
        uint32_t total_time = (On_time_us>>1) + (Off_time_us>>1);
        uint32_t on_time_us = On_time_us >> 1;

        // MOde of operation to phase_corrected_phase_top_max Mode -- WGM2[2:0] === 101
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A & ~(1<<1);
        TCCR2B = TCCR2B | (1<<3);

        // which is reflected in PD3
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A & ~(1<<4);

        if(total_time <=3)
        {
                // if total_time <= 3us -- so we stop clock

                OCR2A = 0;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS2[2:0] === 001
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((3 < total_time)  && (total_time <= 16))
        {
                OCR2A = ((total_time * 16) >> 0) - 1;
                OCR2B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS2[2:0] === 001
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B | (1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
```

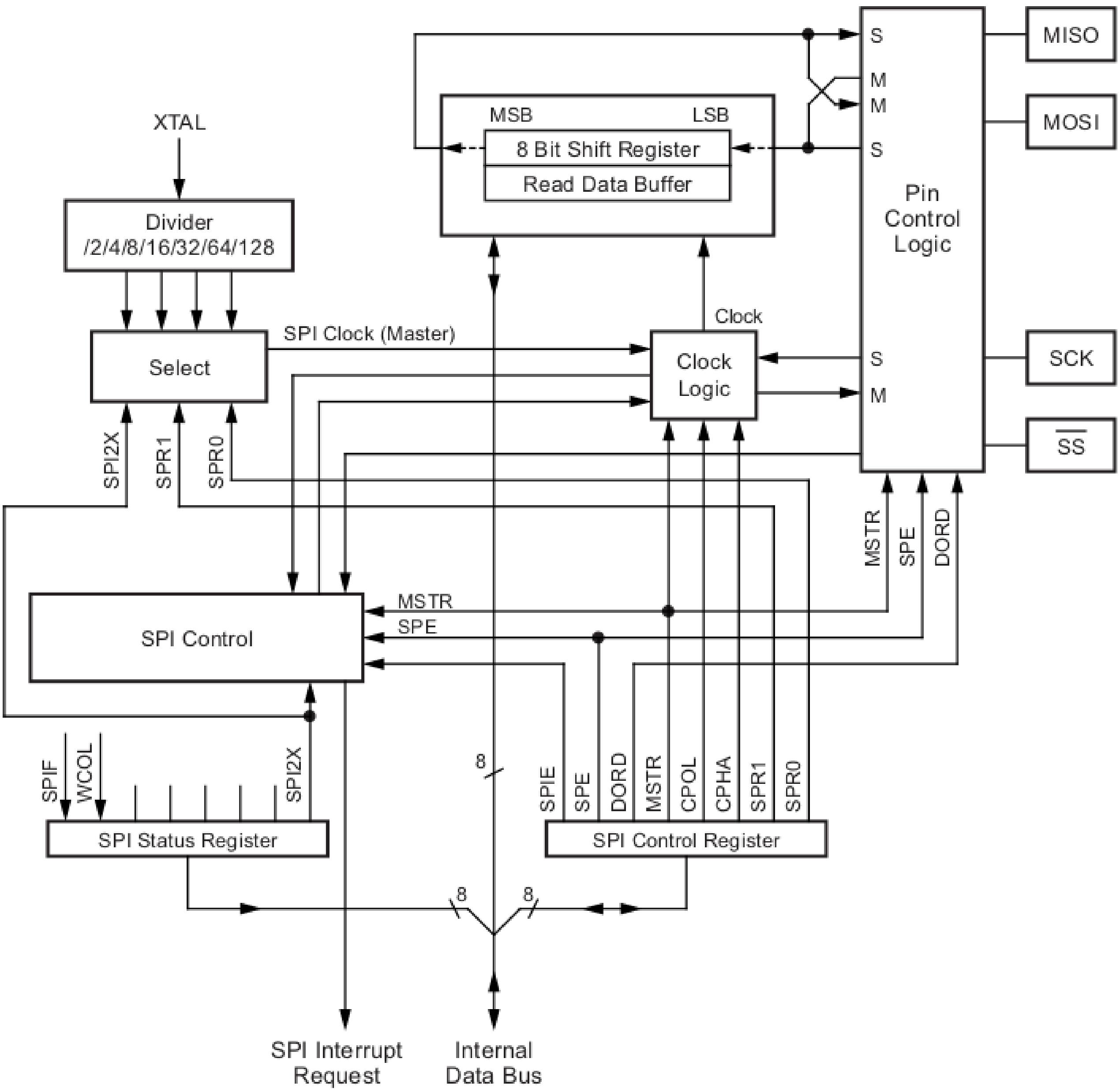
```
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((16 < total_time)  && (total_time <= 128))
        {
                OCR2A = ((total_time * 16) >> 3) - 1;
                OCR2B = ((on_time_us * 16) >> 3) - 1;
                // start timer by setting the clock prescalar
                //  dived by 8 from I/O clock
                //  CS2[2:0] === 010
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B | (1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((128 < total_time)  && (total_time <= 1024))
        {
                OCR2A = ((total_time * 16) >> 6) - 1;
                OCR2B = ((on_time_us * 16) >> 6) - 1;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS2[2:0] === 011
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B | (1<<0);
                TCCR2B = TCCR2B | (1<<1);
                TCCR2B = TCCR2B & ~(1<<2);

        }
        else if((1024 < total_time)  && (total_time <= 4096))
        {
                OCR2A = ((total_time * 16) >> 8) - 1;
                OCR2B = ((on_time_us * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                //  divide by256 from I/O clock
                //  CS2[2:0] === 100
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B | (1<<2);

        }
        else if(total_time > 4096)
        {
                // dont' cross more than 4.096ms
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }

        // max time = 8ms -- min freqency = 125 Hz
        //  min time = 8us -- max frequency = 250000 = 125khz
        Timer2_PhaseCorrectedPWMGeneration(on_time_us, total_time_us - on_time_us);
}
```

Serial Peripheral Interface

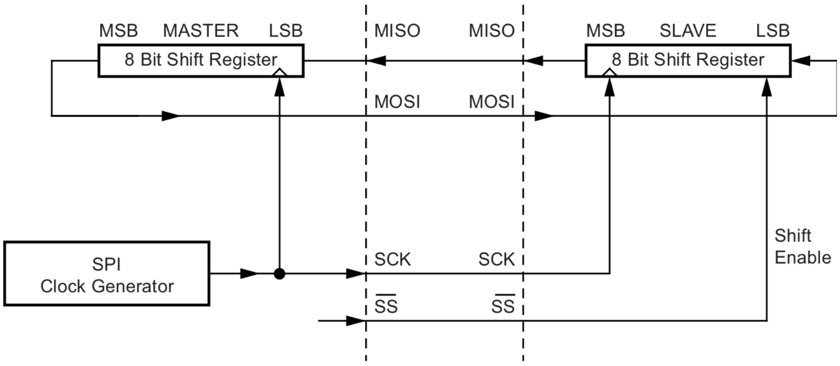
8.1 Features

- Full-duplex, three-wire synchronous data transfer
- LSB first or MSB first
- Seven Programmable bit rates
- high-speed synchronous data transfe

8.2 Block Diagram



8.3 SPI Master-Slave Interconnection



8.3.1 SPI Pins

The SPI is connected to external devices trhough four pins namely,

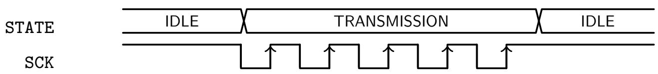
- **MISO** - Master IN / Slave OUT data - transmit data in slave mode and receive data in master mode.
- **MOSI** - Master OUT / Slave IN data - transmit data in master mode and receive data in slave mode.
- **SCK** - Serial Clock - outputs clock on SPI master mode and inputs clock on SPI slave mode.
- **NSS** - Slave Select - select the chip or the slave.

8.3.2 Basic Operation

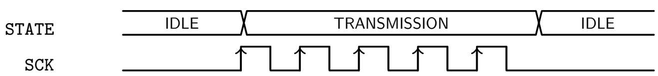
- Two shift Registers and a master clock generator.
- Initialization is done by pulling low the $\overline{SS}$ pin.
- Master generates the required clock pulses on *SCK* to interchange data.
- Using *MOSI* – Master Out Slave In – data is shifted from master to slave.
- Using *MISO* – Master In Slave Out – data is shifted from slave to master.
- After each data packet, the master will synchronize the Slave by pulling high the Slave select $\overline{SS}$ pin.

8.3.3 Clock Phase and Clock polarity

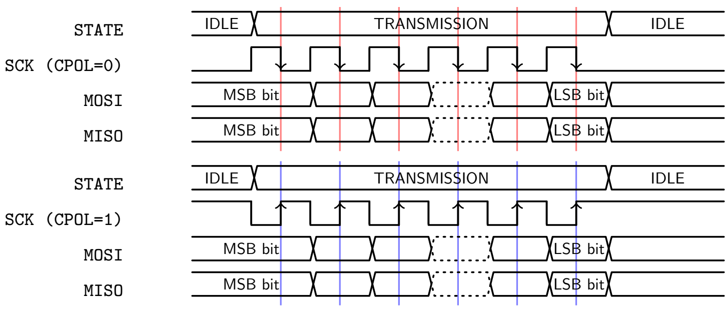
- ***CPOL*** bit controls the steady state value of *SCK* line when idle(no data is transferred).
  - ***CPOL*** $= 1$: *SCK* line is high-level idle state



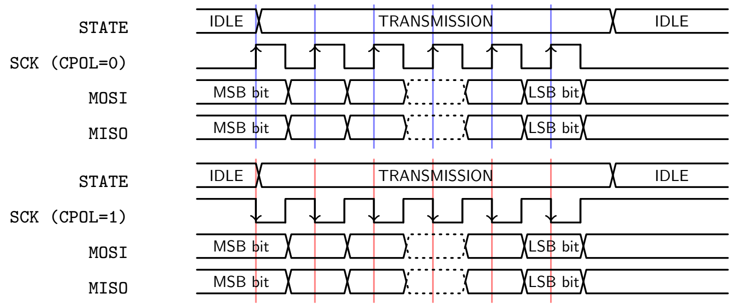
  - ***CPOL*** $= 0$: *SCK* line is low-level idle state



- ***CPHA*** bit controls the capture of datas.
  - ***CPHA*** $= 1$: MSB bit is captured on the **second edge** of *SCK* pin (falling edge if the ***CPOL*** bit is 0, rising edge if the ***CPOL*** bit is 1).



– ***CPHA*** = 0 : MSB bit is captured on the **first edge** of *SCK* pin (falling edge if the ***CPOL*** bit is 1, rising edge if the ***CPOL*** bit is 0).



8.3.4 Data Frame Format

The data can be shifted out either MSB first or LSB first.

8.4 Register Description

SPCR – SPI Control Register

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

- ***SPIE - SPI Interrupt Enable*** - Enable the SPI interrupt to be executed if ***SPIF*** bit is set in **SPSR** Register.
- ***SPE - SPI Enable*** - Enable the SPI.
- ***DORD - Data Order*** - Defines the data order being sent[1 == LSB first; 0 == MSB first]
- ***MSTR - Master/Slave Select*** - Select between Master Mode and Slave Mode[1 == Master Mode; 0 == Slave Mode]

SI2X, SPR1, SSPR0	**SCK Frequency**
000	$\frac{fosc}{4}$
001	$\frac{fosc}{16}$
010	$\frac{fosc}{64}$
011	$\frac{fosc}{128}$
100	$\frac{fosc}{2}$
101	$\frac{fosc}{8}$
110	$\frac{fosc}{32}$
111	$\frac{fosc}{64}$

SPSR – SPI Status Register

7	6	5	4	3	2	1	0
SPIF	WCOL	-	-	-	-	-	SPI2X

- ***SPIF - SPI Interrupt Flag*** - Denotes the end of serial transfter. A intterup its generated if ***SPIE*** bit in **SPCR** register is set.

SPDR – SPI Data Register

7	6	5	4	3	2	1	0
D7	D6	D5	D4	D3	D2	D1	D0

8.5 Configuring the SPI

- First, the pins *MOSI*, *MISO*, *SCK* and $\overline{SS}$ is configured to required Direction.
- Next, the $\overline{SS}$ pin is made low or high depending on the device Specs.
- The data order is selected by ***DORD*** bit in **SPCR** register.
- The Master/Slave Mode is selected by ***MSTR*** bit in **SPCR** register.
- The timing is choosen by Configuring ***CPOL*** and ***CPHL*** bit in **SPCR** register depending on the Device Specs.
- The Clock Frequency for SPI communication is choosen by Configuring the ***SPI2X***, ***SPR1*** and ***SPR*** bits of **SPCR** and **SPSR** registers.
- Interrupt is enabled by setting the ***SPIE*** bit in **SPCR** register.
- FInally, SPI is enabled by setting the **SPE** bit in **SPCR** register.
- Also, the interrupt service routing is written, when the transmission/reception completes.
- The data can be transmitted/received by writing/reading from **SPIDR** register.
- An example code is seen below,

```
// making SCK, MOSI, SS' as outptut
DDRB |= (1<<DDB2) | (1<<DDB3) | (1<<DDB5);
// making MISO as input
DDRB &= ~(1<<DDB4);

// making SCK, MOSI,  as low
PORTB &=  ~(1<<PORTB3) & ~(1<<PORTB5);
// making SS' as high
PORTB |= (1<<PORTB2);

// Select MSB first or LSB first by DORD
SPCR &= ~(1<<DORD);

// Select this as Master
SPCR |= (1<<MSTR);

// Let the clock polarity be SCK is low when idle
SPCR &= ~(1<<CPOL);

// Sampled at Rising or Falling Edge
// we choose rising edge
SPCR &= ~(1<<CPHA);

// Selecting a SCK frequnecy
// we select Fosc/4 by 000
SPSR &= ~(1<<SPI2X);
SPCR &= ~(1<<SPR1);
SPCR &= ~(1<<SPR0);
// dISBALE SPIE bit for interrupt on Serial Transfer Completion
SPCR &= ~(1<<SPIE);
// Enabling SPI
SPCR |= (1<<SPE);
```

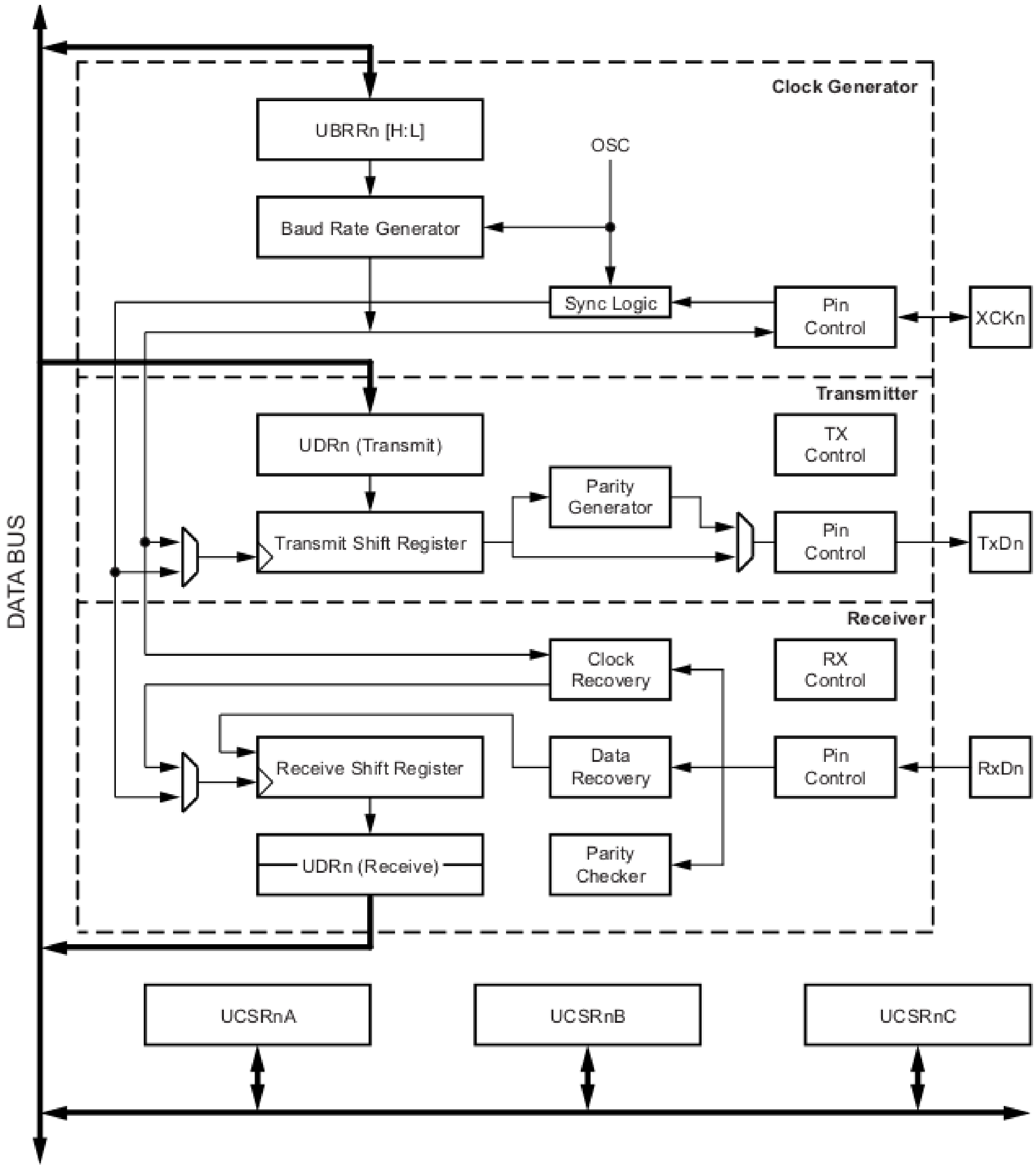
```
uint8_t SPITransferReceive(uint8_t data_)
{
    SPDR = data_;
    // wait till serial transmission is complete by checking the SPI Interrupt Flag
    while((SPSR & (1<<SPIF)) == 0 ) {};
    // return the recieved data - can use it or ignore it
    return SPDR;
}
```

Universal Synchronous and Asynchronous serial Receiver and Transmitter 0

9.1 Features

- Full duplex operation (independent serial receive and transmit registers).
- Asynchronous or synchronous operation
- High resolution baud rate generator
- Serial frame with 5,6,7,8,9 data bits and 1 or 2 stop bits
- Odd or even partiy generator and checker by hardware
- Double speed asynchronous communication mode

9.2 Block Diagram



9.2.1 Clock Generator Block

- Consist of sync. Logic for external clock input for usage in sync. slave operation
- Consist of Baud rate Generator.
- Uses the *XCKn* pin for sync. Transfer mode

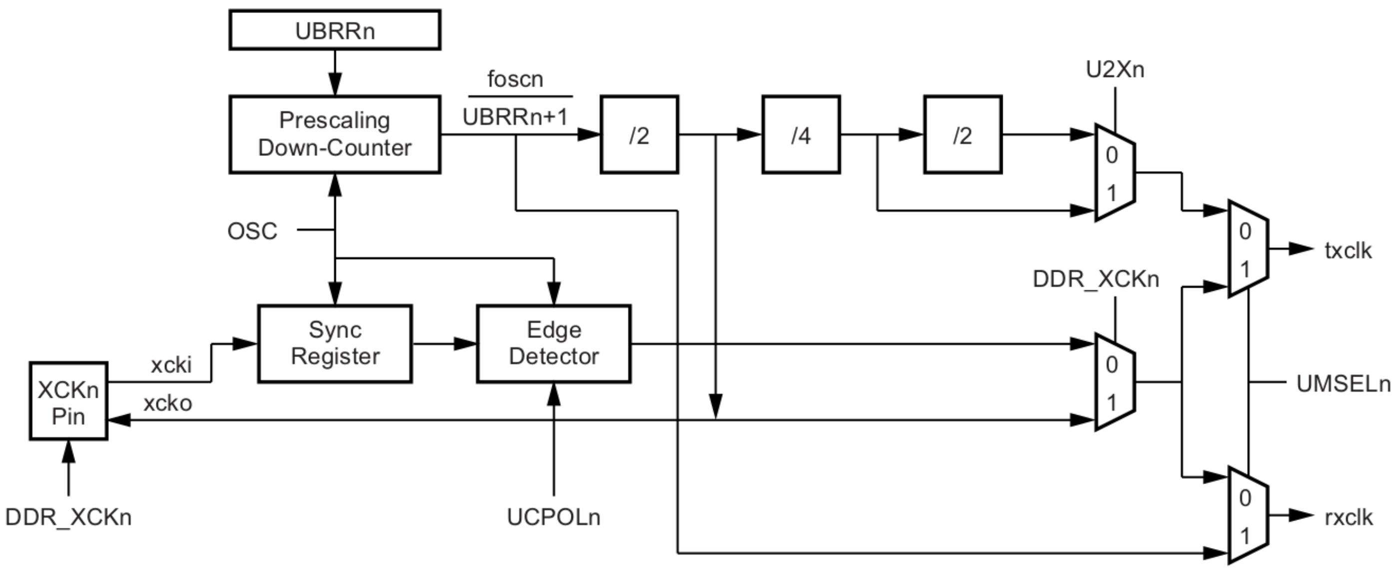
9.2.2 Transmitte Block

- Consist of singe write buffer – continuous transfer of data without delay between frames
- Consist of Serial Shift register and Parity Generator
- Also, Control logic for handling different serial frame format.

9.2.3 Receiver Block

- Consist of Clock and data recovery unit – uses for Asynchronous reception
- Consist of Parity Checker, Control Logic, Shift Register, Two level Receiver buffer
- Can support frame error, data overrun parity error

9.3 Clock Genration



- Generates Base Clock for Transmitter and Receiver.
- USART supports four modes of clock operation
  - (i) Normal Asynchronous
  - (ii) Double Speed Asynchronous
  - (iii) Master synchronous
  - (iv) Slave synchronous
- Selection between Asynchronous and Synchronous is done by ***UMSELn*** bit in ***UCSRnC*** - USART Control and Status Register C.
- The Double Speed is selected by ***U2Xn*** bit in ***UCSRnA*** - USART Control and Status Register A.
- In Synchronous Mode, the master or slave mode is selected by ***DDR_XSCn*** bit direction. [external - slave mode; internal - master mode]

Signals	**Description**
txclk	Transmitte Clock
rxclk	Receiver Base Clock
xclki	Input from *XCK* pin - used for synchronous slave operation.
xclko	Clock output to *XCK* pin - used for synchronous master operation.
fosc	*XTAL* pin freqency (System clock).

9.3.1 Internal Clock Generation - The Baud Rate Generator

- Used for Asynchronous and Synchronous Master modes of operation.
- Programmed using **UBRRn** register.

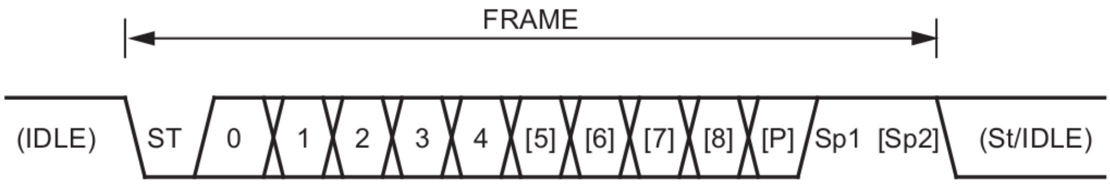
Operating Mode	**UBRRn calculation**
Asynchronous Normal Mode(***U2Xn*** == 0)	$UBRRn = \frac{fosc}{16*BAUD} - 1$
Asynchronous Double Speed Mode(***U2Xn*** == 1)	$UBRRn = \frac{fosc}{8*BAUD} - 1$
Synchronous Master Mode	$UBRRn = \frac{fosc}{2*BAUD} - 1$

9.3.2 External Clock

- Used by synchronous Slave mode.
- External clock input from *XCKn* pin is used and should

$$f_{XCK} < \frac{fosc}{4}$$

9.4 Frame Format



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.

- A serial frame is defined to be one character of data bits with synchronization bits (start and stop bits), and optionally a parity bit for error checking.
- The combinations can be
  - 1 start bit
  - 5 or 6 or 7 or 8 or 9 data bits
  - no or even or odd parity bits
  - 1 or 2 stop bits
- A frame starts with start bit followed by LSB data bits.
- Next the data bet can be from 5 to 9 ending with MSB data bits.
- Parity bits may be added if enabled.
- Finally, stop bit of 1 or 2 size is added.
- Generally, the line is idel with high Logic.

9.5 Register Description

UDRn – USART I/O Data Register n

7	6	5	4	3	2	1	0
RXB[7:0]							
TXB[7:0]							

UCSRnA – USART Control and Status Register n A

7	6	5	4	3	2	1	0
RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn

UCSRnB – USART Control and Status Register n B

7	6	5	4	3	2	1	0
RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXBn	TXB8n

UCSRnC – USART Control and Status Register n C

7	6	5	4	3	2	1	0
UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn

- ***RXCn*** - USART Receive Complete - Set when there are unread data in receive buffer.
- ***TXCn*** - USART Transmit Complete - Set when the entire frame in the transmit shift register has been shifted out and there are no new data currently present in the transmit buffer.
- ***UDREn*** - USART Data Register Empty - indicates if the transmit buffer is ready to receive new data. A one indicates buffer is expty and ready to transmit.
- ***U2Xn*** - Double the USART Transmission Speed - Affects only the asynchronous operation. One will increase the speeed of transfer rate in asynchronous opration.
- ***RXCIEn*** - RX Complete Interrupt Enable n - Writing one will enabled Receive Complete intterrupt.
- ***TXCIEn*** - TX Complete Interrupt Enable n - Writing one will enabled Transmit Complete intterrupt.
- ***UDRIen*** - USART Data Register Empty Interrupt Enable n - Enable data register empty intterrupt.
- ***RXENn*** - Receiver Enable - enable the receiever for reception.
- ***TXENn*** - Transmitter Enable - enable the Transmitter for Transmission.
- ***UCSZn[2:0]*** - Character Size n - select the number of data bits in a frame.
- ***RXB8n*** - Receive Data Bit 8 n - it's the actual 9th bit received.
- ***TXB8n*** - Transmit Data Bit 8 n - it's the actual 9th bit to tbe trasnmitted.
- ***UMSELn[1:0]*** - USART Mode Select - Select the mode.
- ***UPMn[1:0]*** - Parity Mode - Disable or set the parity mode type.
- ***USBSn*** - Stop Bit select - Selects the number of stop bits to be inserted by transmitter.

UMSELn[1:0]	**Mode**
00	Asynchronous USART
01	Synchronous USART
10	Reserved
11	Master SPI

UPMn[1:0]	**Parity Mode**
00	Disabled
01	Reserved
10	Even Parity
11	Odd Parity

USBSn	**Stop Bit(s)**
0	1-bit
0	2-bit

UCSZn[2:0]	**Character Size**
000	5-bit
001	6-bit
010	7-bit
011	8-bit
111	9-bit

UBRRnL and UBRRnH – USART Baud Rate Registers

15	14	13	12	11	10	9	8
-	-	-	-	UBRRn[11:8			
UBRRn[7:0							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**

UBRRn[11:0] - the actual 12-bit USART Baud Rate Registers.

9.6 Configurint USART

- First, the mode is selected by confguring the ***UMSEL0[1:0]*** bits in **UCSR0C** register.
- Next, the Baud rate is choosen and set in ***UBRR0[11:0]*** bits in **UBRR0H** and **UBRR0L** registers.
- Next, the frame format is set by confguring,
  - Data Length - by confguring ***UCSZ0[2:0]*** bit in **UCSR0B** and **UCSR0C** register.
  - Parity - by confguring ***UPM0[1:0]*** bit in **UCSR0C** register.
  - Stop bits - by confguring ***USBS0*** bit in **UCSR0C** register.
- Interrupt may be anabled by setting bits in ***UCSR0A*** register and ISR are wirtten.
- Finally, the Transmitter and Receiver are enabled by setting ***TXEN0*** and ***RXEN0*** bits in **UCSR0B**.
- The data can be sent by checking if the ***UDRE0*** bit is set in **UCSR0A** register and wiring the 8-bit data into **UDR0** register.
- The data can be received by checking if the ***RXC0*** bit is set in **UCSR0A** register and reading the 8-bit data from **UDR0** register.
- The code for a simple USART is seen below,

```
// Setting up the Mode
// Select the Asyncronous Master Mode.
// Setting UMSEL0[1:0] in UCSR0C to 00
UCSR0C &= ~(1<<UMSEL00);
UCSR0C &= ~(1<<UMSEL01);

// setting up the Buad rate
// Due to The Clock rate being 8MHz, for a buad rate of 9600
// UBRR0 = (fosc / (16*BAUD)) -1
// So UBRR0 = (8000000 / (16 * 9600)) - 1 = 0x33
UBRR0H = 0x00;
UBRR0L = 0x33;

// setting up the Frame Format
// Let's select 8-bit data bits, no parity, and 1 stop bit
// 8 - bit data bits
// By selecting UCSZ0[2:0] in UCSR0C and UCSR0B register to be 011
UCSR0B &= ~(1<<UCSZ02);
UCSR0C |= (1<<UCSZ01);
UCSR0C |= (1<<UCSZ00);
// No parity
// By selecting UPM0[1:0] in UCSR0C to 00
UCSR0C &= ~(1<<UPM01);
UCSR0C &= ~(1<<UPM00);
// 1 stop bit
// By selecting USBS0 in UCSR0C to 0
UCSR0C &= ~(1<<USBS0);

// Disabling any interrupts
UCSR0B &= ~(1<<7);
UCSR0B &= ~(1<<6);
UCSR0B &= ~(1<<5);

// Enabling Transmitter
UCSR0B |= (1<<TXEN0);
// Enabling Receiver
UCSR0B |= (1<<RXEN0);
```

```
void USART0sendChar(uint8_t data_)
{
        //cHECKING if transmitet buffer is empty
        while((UCSR0A & (1<<UDRE0)) == 0x00){};
        UDR0 = data_;
}
uint8_t USART0receiveChar()
{
        // wait for thedate to be recied
        while((UCSR0A & (1<<RXC0)) == 0x00){};
        return UDR0;
}
```

Two Wire Interface

10.1 Features

- 7-bit address space allows up to 128 different slave addresses
- Multi-master arbitration support
- Up to 400kHz data transfer speed
- Noise suppression circuitry rejects spikes on bus lines
- Fully programmable slave address with general call support
- Compatible with Phillips I2C

10.2 2-wire Serial Interface

- Suited for typical microcontroller applications
- Allows upto 128 different device.
- All devices connected must have individual address and method to resolved bus contention

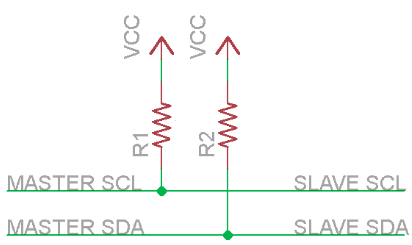
10.2.1 I^2C Pins

- Output driver consist of slew-rate limiter to confirm the TWI specification.
- Input stage consist of spike suppression unit to remove spikes shorter than 50ns.
- Internal pull-up can also be used.
- *SDA* - Serial Data - the actual serial data transfer pinFormat
- *SCL* - Serial Clock - driven by device in Master Mode

10.2.2 Terminology

Term	Descripton
Master	Device that initiates and terminates transacting and also Generates *SCL* Clock.
Slave	Device addressed by a master.
Transmitter	Device placing data on the bus.
Receiver	Device reading data on the bus.

10.2.3 Electrical Interconnection

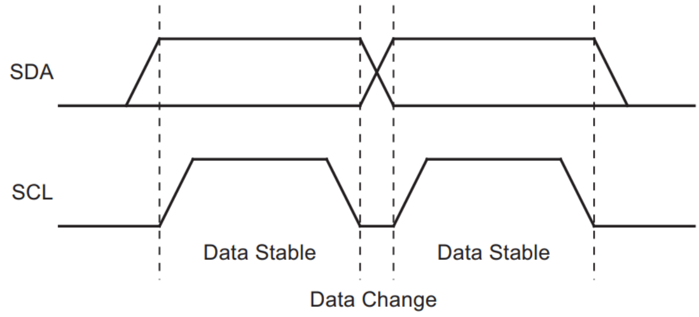


- both lines are connected to positive supply voltage through pull-up resistor

- bus driver are open-drain or open-collector
- no. of device also depends on the bus capacitance limit of 400pF

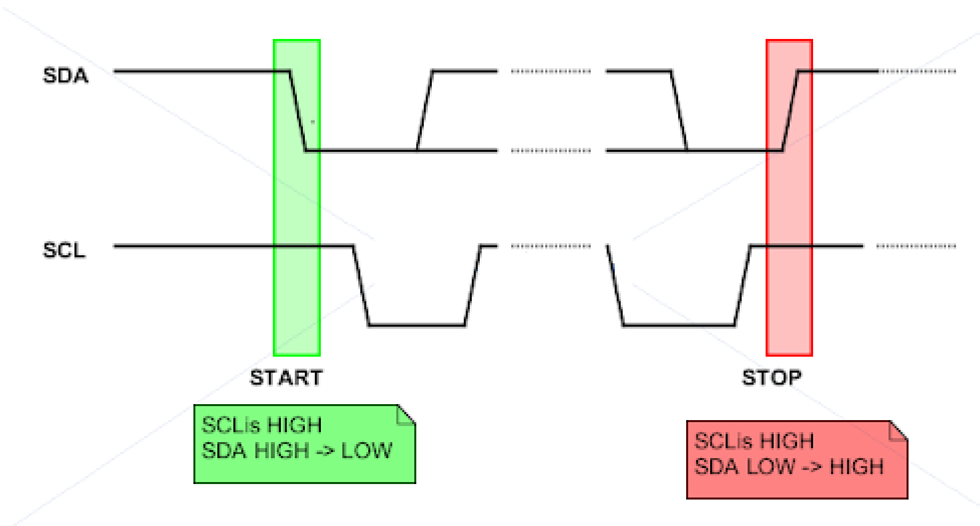
10.3 Data Transfer and Frame Format

10.3.1 Transferring Bits



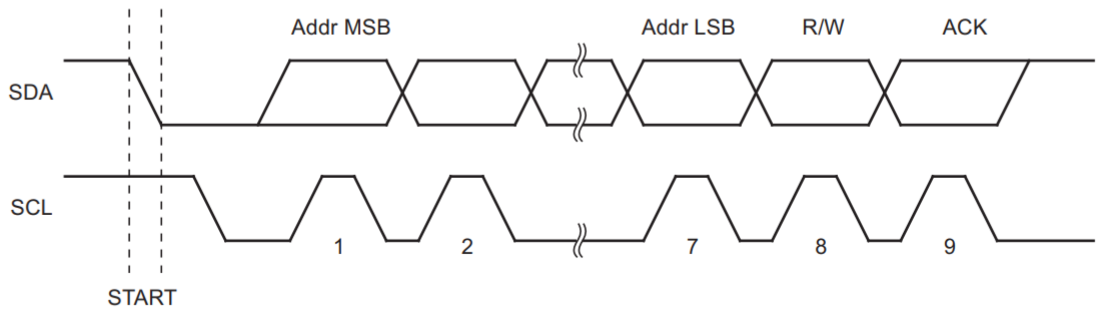
- Each data bit transferred is done by a pulse on clock line.
- Level of data line must be stable when the clock line is high.

10.3.2 START and STOP Conditions



- Both **START** and **STOP** conditions are done by changing *SDA* line when *SCL* is kept high.
- Master initiates transmission by issuing a **START** condition.
- Master terminates transmission by issuing a **STOP** condition.
- Between **START** and **STOP**, bus is busy and no other master should try to control bus.
- The same master however can issue **REPEATED START**(same as **START**) to initiate a new transfer.

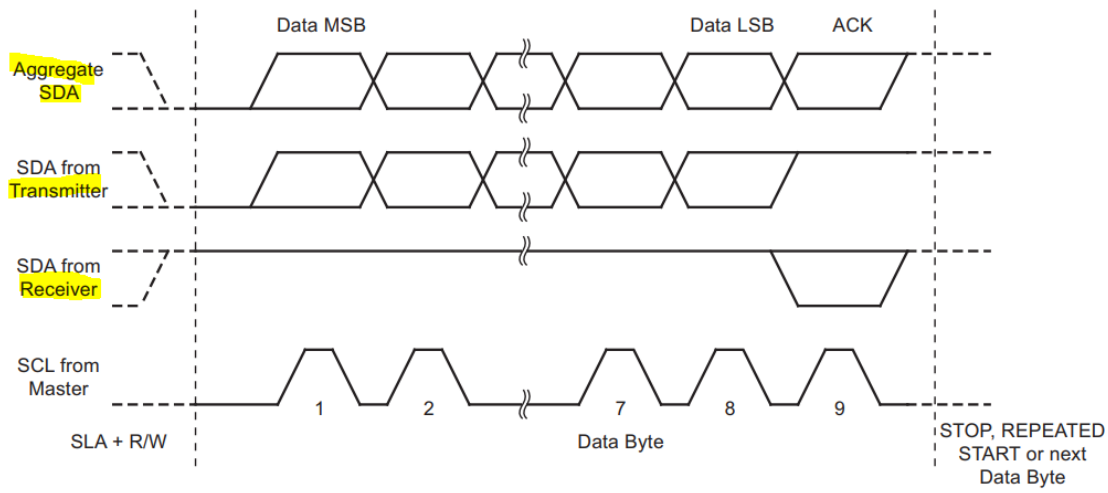
10.3.3 Address Packet format



- Addresses packets(**SLA**) are 9-bit long.
  - 7 address bits with MSB transmitted first
  - one READ(1)/WRITE(0) control bit indicating the transmitter or receiver mode respectively
  - one acknowledge bit by the Slave
- Would take 8 clock cycles by the master to send 7 address bits and one READ/WRITE control bit. **SLA+R** or **SLA+W**.
- On the 9th clock cycle, Master will leave out the control of *SDA* line (making it high due to pull-up resistor) but clocks out the 9th clock on the *SCL* line.
- On the 9th clock cycle, Slave recognizes that it is being addressed by pulling *SDA* line low making the **ACK**.
- If the Slave couldn't for some reason respond, the *SDA* line remains high in the 9th clock cycle making the **NACK**.

10.3.4 Data Packet format

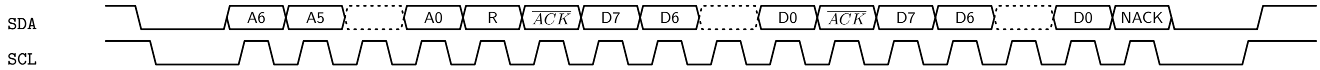
- Data packetsare 9-bit long.
  - one data byte - 8 bits with MSB first.
  - one acknowledge bit by the master or slave depending the mode.
- The transmitter(either the master or slave) send 8-bit data in 8 clock cycles.



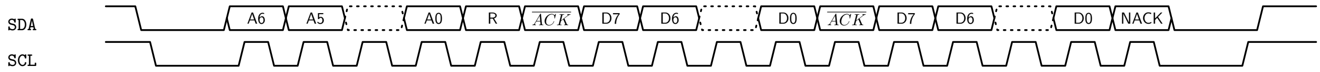
- On the 9th clock cycle, the transmitter will leave out the control of *SDA* line (making it high due to pull-up resistor).
- During the 9th clock cycle, the receiver pulls down the *SDA* line low to acknowledge the reception. – **ACK** is signaled by receiver.
- If the receiver doesn't pull down the *SDA* line for some reason, then the *SDA* line remains high. – **NACK** is signaled by receiver.
- When the receiver received the last byte or can't receive more byte, it should inform the transmitter by sending a **NACK**.

10.3.5 Overall Operation

Write Operation

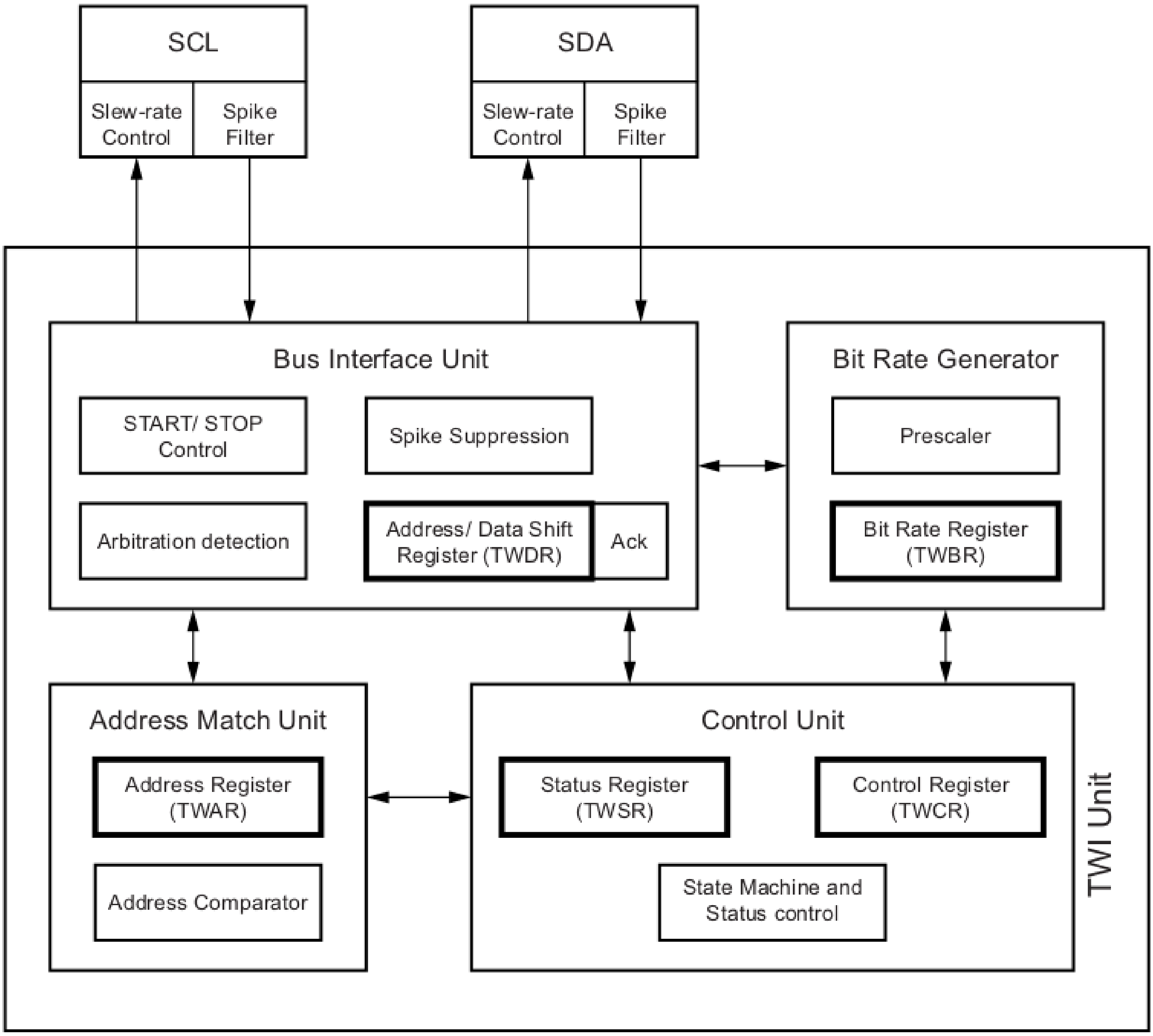


Read Operation



10.4 TWI Module

10.4.1 Block Diagram



10.4.2 Bit Rate Generation Unit

- controls *SCL* line when in Master mode
- ***TWBR*** (TWI Bit Rate Generator) and Prescalar Bits in TWI status register **TWSR** control *SCL*
- The *SCL* frequency can be

$$SCLfrequency = \frac{CPUClockFrequency}{16+2*TWBR*PrescalarValue}$$

Note: Slave's clock frequency must be atleast 16 times higher than SCL frequency.

10.4.3 Bus Interface Unit

- contains Data and adress Shift register - **TWDR** register - address or data transmitted or received
- **START/STOP** Controller - Generates and detects **START**, **STOP** and **REPEATED START**
- Register Containing the **(N)ACK** to be transmitted or received
- Arbitration detection hardware.

10.4.4 Address Match Unit

- Received address bytes matches the seven-bit address in **TWAR** (TWI Address register).
- address match results in informing the control unit

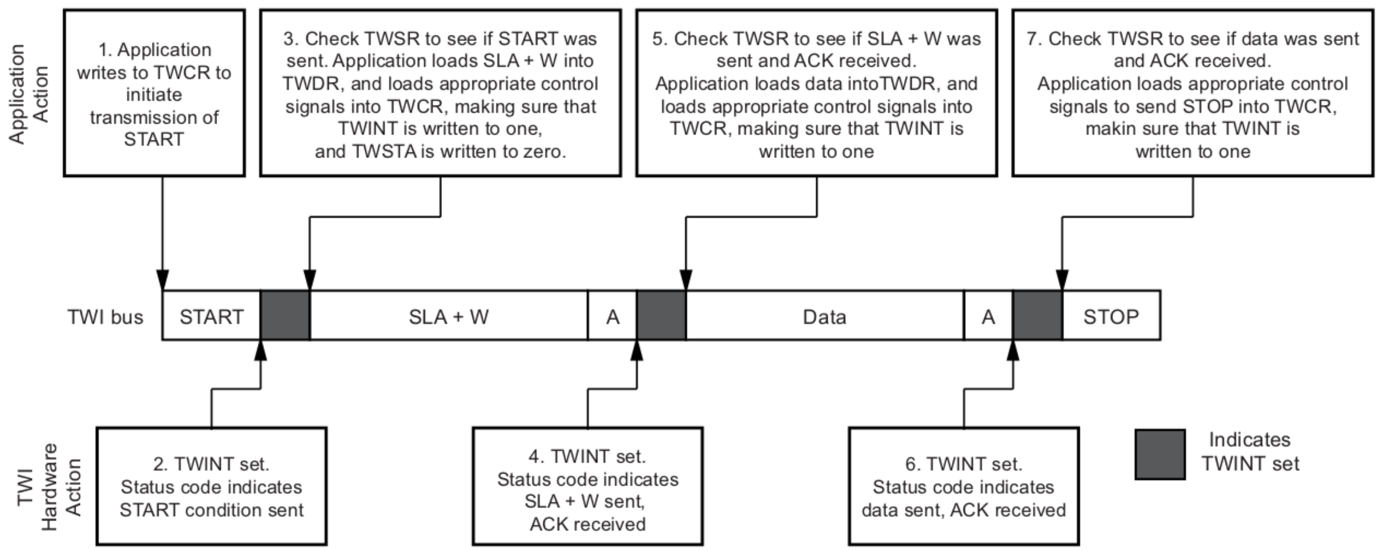
10.4.5 Control Unit

- Monitors TWI bus and generates responses based on TWI control register (**TWCR**).
- When a en even requires attention:
  - ***TWINT*** (TWI Interrupt flag) is set
  - **TWSR** (TWI Status Register) is updated with status code identifying the event.
  - when the ***TWINT*** is set, the *SCL* line is held low.
- ***TWINT*** flag is set If
  - TWI has transmitted a **START/REPEATED START** condition
  - TWI has transmitted **SLA+R/W**
  - TWI has transmitted an address byte
  - TWI has lost arbitration
  - TWI has been addressed by own slave address or general call
  - TWI has received a data bye
  - **STOP** or **REPEATED START** has been received
  - bus error occurred due to illegal **START** or **STOP**

10.5 TWI Usage

- TWI is interrupt based and so ***TWIE*** bit in **TWCR** register should be enabled; If the ***TWIE*** is diabled, then the ***TWINT*** flag must be polled.
- When the ***TWINT*** flag is asserted, TWI has finised operation and awaits for application response and **TWSR** register describes the current status of TWI bus.
- Then, application should respond by manipulating the **TWCR** and **TWDR** register.

10.5.1 An Example - Master transmits single data byte to slave



1. Transmission is started by writing specific value into **TWCR** register to transmit the **START** condition. The ***TWINT*** flag is cleared by writing Loging HIGH which initiate the transmission of **START** condition.
2. When the **START** condition has been transmitted, the TWINT flag in TWCR is set, and **TWSR** is updated with a status code indicating that the **START** condition has successfully been sent.

3. The application should now respond by examining the **TWSR** register value. If status code is as expected, the application loads **SLA+W** into **TWDR** and a specific value is written into **TWCR** register to transmit the **SLA+W** present in **TWDR**. The ***TWINT*** flag is cleared by writing logic HIGH which initiates the transmission of address packet.
4. When the address packet has been transmitted, the ***TWINT*** flag in **TWCR** is set, and **TWSR** is updated with a status code indicating that the address packet has successfully been sent. The status code will also reflect whether a slave acknowledged the packet or not.
5. The application should now respond by examining the **TWSR** register value and **ACK** bit is as expected. If status code is as expected, the application loads Data packet into **TWDR** and a specific value is written into **TWCR** register to transmit the Data packet present in **TWDR**. The ***TWINT*** flag is cleared by writing logic HIGH which initiates the transmission of data packet.
6. When the data packet has been transmitted, the ***TWINT*** flag in **TWCR** is set, and **TWSR** is updated with a status code indicating that the data packet has successfully been sent. The status code will also reflect whether a slave acknowledged the packet or not.
7. The application should now respond by examining the **TWSR** register value and **ACK** bit is as expected. If status code is as expected, the application loads a specific value is written into **TWCR** register to transmit the **STOP** Condition. The ***TWINT*** flag is cleared by writing logic HIGH which initiates the transmission of STOP condition.

10.6 Transmission Modes

There are four major Modes

(i) Master Transmitter (MT)

(ii) Master Receiver (MR)

(iii) Slave Transmitter (ST)

(iv) Slave Receiver (SR)

Status Code	**Meaning**
S	**START** Condition
Rs	**REPEATED START** Condition
R	Read bit (high level on *SDA*)
W	Write bit (low level on *SDA*)
A	Acknowledge bit (low level on *SDA*)
$\overline{A}$	Not Acknowledge bit (high level on *SDA*)
DATA	8-bit data
P	**STOP** Condition
SLA	Slave Address

10.6.1 Master Transmitter Mode (MT)

- Many number of data bytes are transmitted to Slave receiver.
- For Master, **START** Condition is transmitted
- **START** condition is sent by:
  - ***TWEN*** bit is set to enable TWI.
  - ***TWSTA*** bit is set to transmit **START** condition.
  - ***TWINT*** flag is written 1 to clear to send start bit.
  - After Tranmitting **START** condition, ***TWINT*** flag is set by hardware and status code in **TWSR** register should be 0x08 - indicating successfull transmission of **START** condition.
- To enter into Master Transmitter Mode and transmit the address:
  - Write **SLA+W** into **TWDR** register.
  - ***TWINT*** flag is written 1 to clear to transmit Address and read/write status.

- After transmitting SLA+W, an acknowledgment bit will be received, the ***TWINT*** flag is set by hardware and status code in **TWSR** register will be 0x18 (indicating SLA+W has been transmitted and ACK has been received), 0x20 (indicating SLA+W has been transmitted and NACK has been received), 0x38 (Arbitration lose in sending SLA+W).

- Data packet is transmitted by:
  - Write DATA packet into **TWDR** register.
  - ***TWINT*** flag is written 1 to clear to transmit Address and read/write status.
  - After transmitting DATA packet, an acknowledgment bit will be received, the ***TWINT*** flag is set by hardware and status code in **TWSR** register will be 0x28 (indicating DATA packet has been transmitted and ACK has been received), 0x30 (indicating DATA packet has been transmitted and NACK has been received)
  - To send further data, the above process is repeated by sending REPEATED START.
  - To stop the transmission, the STOP condition is sent.
- STOP condition is sent by:
  - ***TWSTO*** bit is set to transmit STOP condition.
  - ***TWINT*** flag is written 1 to clear to send stop bit.
- REPEATED START condition is sent by:
  - ***TWSTA*** bit is set to transmit REPEATED START condition.
  - ***TWINT*** flag is written 1 to clear to send repated start bit.
  - After Tranmitting REPEATED START condition, ***TWINT*** flag is set by hardware and status code in **TWSR** register should be 0x10 - indicating successfull transmission of REPEATED START condition.

10.6.2 Master Receiver Mode (MR)

- Many number of data bytes can be received from Slave transmitter.
- For Master, START Condition is transmitted
- START condition is sent by:
  - ***TWEN*** bit is set to enable TWI.
  - ***TWSTA*** bit is set to transmit START condition.
  - ***TWINT*** flag is written 1 to clear to send start bit.
  - After Tranmitting START condition, ***TWINT*** flag is set by hardware and status code in **TWSR** register should be 0x08 - indicating successfull transmission of START condition.
- To enter into Master Receiver Mode and transmit the address:
  - Write SLA+R into **TWDR** register.
  - ***TWINT*** flag is written 1 to clear to transmit Address and read/write status.
  - After transmitting SLA+R, an acknowledgment bit will be received, the ***TWINT*** flag is set by hardware and status code in **TWSR** register will be 0x40 (indicating SLA+R has been transmitted and ACK has been received), 0x48 (indicating SLA+R has been transmitted and NACK has been received), 0x38 (Arbitration lose in sending SLA+R).
- Data packet is received by:
  - Reading the DATA packet from **TWDR** register if ***TWINT*** flag is logic HIGH.
  - ***TWINT*** flag is written 1 to clear.
  - After receiving DATA packet, an acknowledgment bit will be returned, the ***TWINT*** flag is set by hardware and status code in **TWSR** register will be 0x58 (indicating DATA packet has been recieved and ACK has been returned), 0x50 (indicating DATA packet has been recieved and NACK has been returened)
  - To receive further data, the above process is repeated by sending REPEATED START.
  - To stop the reception, the STOP condition is sent.
- STOP condition is sent by:

- *TWSTO* bit is set to transmit STOP condition.
- *TWINT* flag is written 1 to clear to send stop bit.

- REPEATED START condition is sent by:
  - *TWSTA* bit is set to transmit REPEATED START condition.
  - *TWINT* flag is written 1 to clear to send repated start bit.
  - After Tranmitting REPEATED START condition, *TWINT* flag is set by hardware and status code in **TWSR** register should be 0x10 - indicating successfull transmission of REPEATED START condition.

10.6.3 Slave Receiver Mode (SR)

- Many number of data bytes are received from Master transmitter.
- To initiate the Slave Mode:
  - *TWA[5:0]* bits from **TWAR** regFormat is loaded with our slave address.
  - *TWSTA* and *TWSTO* are set to 0
  - *TWEN* bit is set to enable the TWI.
- TWI waits until it is addressed by its own slave address followed by data direction bit.
- After receiving own Slave address and Write bit, the *TWINT* flag is set and valid status code is available in **TWSR** register.
- If the status code is 0x60 (Own SLA+W has been recieved and ACK has been returned)
- Now, data can be read by wiring Logic HIGH on *TWINT* flag to clear and read from **TWDR** register.
- *TWEA* bit is set to acknowledge and recieve further data or *TWEA* bit is cleared and last byte is received.
- Now, *TWINT* flag is set and status code is avalaible in **TWSR**.
- If status code is 0x80 (Previously addressed with own SLA+W and data has been received and ACK has been returned) - to receive further data.
- If status code is 0x88 (Previously addressed with own SLA+W and data has been received and NACK has been returned) - last data is recieved.
- If status code is 0xA0 - A STOP condition is recieved. has been received

10.6.4 Slave Transmitter Mode (ST)

- Many number of data bytes are transmitted to Master recive.
- To initiate the Slave Mode:
  - *TWA[5:0]* bits from **TWAR** regFormat is loaded with our slave address.
  - *TWSTA* and *TWSTO* are set to 0
  - *TWEN* bit is set to enable the TWI.
- TWI waits until it is addressed by its own slave address followed by data direction bit.
- After receiving own Slave address and Read bit, the *TWINT* flag is set and valid status code is available in **TWSR** register.
- If the status code is 0xA8 (Own SLA+R has been recieved and ACK has been returned)
- Now, data to be sent is set on the **TWDR** register.
- *TWINT* flag is written 1 to clear to transmit the data.
- Now, *TWINT* flag is set and status code is avalaible in **TWSR**.
- If status code is 0xB8 -(Data byte in **TWDR** has been transmitted and ACK has been received) - can send further data.
- If status code is 0xC8 -(Data byte in **TWDR** has been transmitted and NACK has been received) - last byte send and dont' send further.

10.7 Register Desciption

TWBR – TWI Bit Rate Register

7	6	5	4	3	2	1	0
TWBR[7:0]							

The bit rate is found by,

$$SCLfrequency = \frac{CPUClockFrequency}{16+2*TWBR*PrescalarValue}$$

TWCR – TWI Control Register

7	6	5	4	3	2	1	0
TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE

- ***TWINT*** - TWI Interrupt Flag - Set by hardware when TWI has finished its current job and expects applicaion software reponse. This Flag should be cleared by software by writing login HIGH to start the operation of TWI.
- ***TWEA*** - TWI Enable Acknowledge Bit - Controls the genration of acknowledge pulse.
- ***TWSTA*** - TWI **START** condition - to generate **START** or **REPEATED START**.
- ***TWSTO*** - TWI **STOP** condition - to Generate **STOP** condition.
- ***TWEN*** - TWI Enable Bit - To enabled and active TWI interface - takes control over *SDA* and *SCL* pins, enables slew-rate limiter and spike filter.
- ***TWIE*** - TWI Interrupt ENable - to enable interrupt when TWI flag is high.

TWSR – TWI Status Register

7	6	5	4	3	2	1	0
TWS[7:3]					-	TWPS1	TWPS2

- ***TWS[7:3]*** - TWI Status - reflects the status of TWI logic and 2-wire status bus.

TWPS[1:0] **- TWI Bit rate Prescalar**	**Prescaler Value**
00	1
01	4
10	16
11	64

TWDR – TWI Data Register

7	6	5	4	3	2	1	0
TWD[7:0]							

TWAR – TWI (Slave) Address Register

7	6	5	4	3	2	1	0
TWA[6:0]							TWGCE

- ***TWA[6:0]*** - TWI Slave adress Register - contain seven bit slave address.
- ***TWGCE*** - TWI General Call Recogniction Bit - enables the recognition of general call.

10.8 Configuring the I2c

10.8.1 Master Transmitter and Receiver

The code can be seen below:

```
uint8_t status = 0;
void I2C_Master_Init()
{
        // Intialize the I2C clock frequency to 100kHz
        // let the prescalr be 1
        // f_i2c =  F_CPU / (16 + (2*xTWBR*Prescaler)) = 32
        // setting the TWBR register.
        TWBR = 32;

        // writing 1 to prscalre
        // setting the TWPS bits in TWSR to 00
        TWSR &= ~(1<<TWPS0);
        TWSR &= ~(1<<TWPS1);
}
uint8_t I2C_Master_Status()
{
        // Status value are available from TWSR[7:3]
        return TWSR & 0XF8;
}
uint8_t I2C_Master_START()
{
        // Enabling the TWI interface
        TWCR |= (1<<TWEN);
        // sending START condition
        TWCR |= (1<<TWSTA);
        // Do the transaction
        TWCR |= (1<<TWINT);
        // Checking if START condition is sent correctly
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Master_Status();
        // checking status if START condition is sent correctily
        if(status == 0x08)
        {
                // no error occured
                return 0;
        }
        else
        {
                // error occured
                return 0;
        }
}
uint8_t I2C_Master_STOP()
{
        // Removing Start condition on bit
        TWCR &= ~(1<<TWSTA);
        // sending STOP condition
        TWCR |= (1<<TWSTO);

        // Do the transaction
        TWCR |= (1<<TWINT);

        // disaabling stop and interface


        TWCR &= ~(1<<TWSTO);
        TWCR &= ~(1<<TWEN);
```

```
        return 0;
}
uint8_t I2C_Master_Mode(uint8_t slave_address, uint8_t transmiter0_receiver1)
{
        // Entering MASTER mode
        // Writing SLA+W into TWDR for transmiiter and SLA+R for receiver
        // slave address must be MSB first
        // slave address is left shifted by 1 in order to accompany the R/W bit
        TWDR = (slave_address<<1) | transmiter0_receiver1;
        // Do the transaction
        TWCR |= (1<<TWINT);
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Master_Status();
        // For transmitter the staus would have to be 0x18 and for receiver 0x40
        uint8_t status_val_checker = (transmiter0_receiver1==0) ? 0x18 : 0x40;
        if(status == status_val_checker)
        {
                // no error occured
                return 0;
        }
        else
        {
                // error occured
                return 0;
        }
}
uint8_t I2C_Master_DataTransmitByte(uint8_t data_)
{
        // Data packet is transmitted
        // Writing data intor TWDR
        TWDR = data_;
        // Do the transaction
        TWCR |= (1<<TWINT);
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Master_Status();
        if(status == 0x28)
        {
                // ACK received and still data can be sent
                return 0;
        }
        else if(status == 0x30)
        {
                // NACK received and this is the last data so stop
                return 1;
        }
        else
        {
                // error occured
                return 2;
        }
}
void I2C_Master_DataTransmitString(uint8_t *cdata)
{
        while(*cdata != '\0')
        {
                status = I2C_Master_DataTransmitByte(*cdata++) ;
                if(status == 0)
                {
                        // ACK received and still data can be sent
                        // continue
                }
                else if(status == 1)
                {
```

```
                        // NACK received and this is the last data so stop
                        return;
                }
                else
                {
                        // error occured
                        return;
                }
        }
}

uint8_t I2C_Master_DataReceiveByte()
{
        uint8_t value_ = 0;

        // Data packet is recieved
        TWCR |= (1<<TWINT);
        // Do the transaction
        while((TWCR & (1<<TWINT )) == 0x00)
        {
                value_ = TWDR;
        }


        status = I2C_Master_Status();
        if(status == 0x58)
        {
                // no error occured
                return value_;
        }
        else
        {
                // error occured
                return 1;
        }
}
void I2C_Master_DataReceiveString(uint8_t *recData,uint8_t NUMBYTE)
{
        uint8_t i=0;
        recData[NUMBYTE] = '\0';
        while(i < NUMBYTE)
        {
                // Enabling the Acknowledment bit for replying positive ACK
                TWCR |= (1<<TWEA);
                if(i==(NUMBYTE-1))
                {
                        // disbale the Acknowledment bit for replying Negatice ACK for last byte
                        TWCR &= ~(1<<TWEA);
                }
                status = I2C_Master_DataReceiveByte();
                if(status==0xFF)
                        return;
                else
                        recData[i] = status;
                i++;
        }
}
```

10.8.2 Slave Transmitter and Receiver

The code can be seen below:

```
uint8_t status = 0;
void I2C_SlaveInit(uint8_t my_address)
{
        // slave address  and last LSB 0 is for general call
        TWAR = (my_address<<1) & 0xFE;
        // Enabling the TWI interface.
        TWCR |= (1<<TWEN);
        // Disabling Start and Stop conditon bits
        TWCR &= ~(1<<TWSTA);
        TWCR &= ~(1<<TWSTO);

}
uint8_t I2C_Status()
{
        // Status value are available from TWSR[7:3]
        return TWSR & 0XF8;
}

uint8_t I2C_SlaveMode( uint8_t transmiter0_receiver1)
{
        // Acknowldege the address
        TWCR |= (1<<TWEA);
        // Watiting for the Master to call this slave
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Status();
        // For transmitter the staus would have to be 0xA8 and for receiver 0x60
        uint8_t status_val_checker = (transmiter0_receiver1==0) ? 0xA8 : 0x60;
        if(status == status_val_checker)
        {
                // Master called this slave
                return 0;
        }
        else
        {
                // error occured
                return 1;
        }
}
uint8_t I2C_Slave_DataTransmitByte(uint8_t data_)
{
        // Data packet is transmitted
        // Writing data intor TWDR
        TWDR = data_;
        // Do the transaction
        TWCR |= (1<<TWINT);
        while((TWCR & (1<<TWINT )) == 0x00);

        status = I2C_Status();
        if(status == 0xB8)
        {
                // ACK received and still data can be sent
                return 0;
        }
        else if(status == 0xC8)
        {
                // NACK received and this is the last data so stop
                return 1;
        }
        else
        {
```

```
                // error occured
                return 2;
        }
}
void I2C_Slave_DataTransmitString(char *cdata)
{
        uint8_t i = 0;
        while(cdata[i] != '\0')
        {
                status = I2C_Slave_DataTransmitByte(cdata[i]) ;
                i++;
                if(status == 0)
                {
                        // ACK received and still data can be sent
                        // continue
                }
                else if(status == 1)
                {
                        // NACK received and this is the last data so stop
                        return;
                }
                else
                {
                        // error occured
                        return;
                }
        }
}

uint8_t I2C_Slave_DataReceiveByte()
{
        uint8_t value_ = 0;

        // Data packet is recieved
        TWCR |= (1<<TWINT);
        // Do the transaction
        while((TWCR & (1<<TWINT )) == 0x00)
        {
                value_ = TWDR;
        }

        status = I2C_Status();
        if(status == 0x80)
        {
                // Data is sent and ACK has been returned
                return value_;
        }
        else if(status == 0x88)
        {
                // Data is sent and NACK has been returned for last byte
                return value_;
        }
        else
        {
                // error occured
                return 0xFF;
        }
}
void I2C_Slave_DataReceiveString(uint8_t *recData,uint8_t NUMBYTE)
{
        uint8_t i=0;
        recData[NUMBYTE] = '\0';
        while(NUMBYTE > 0)
```

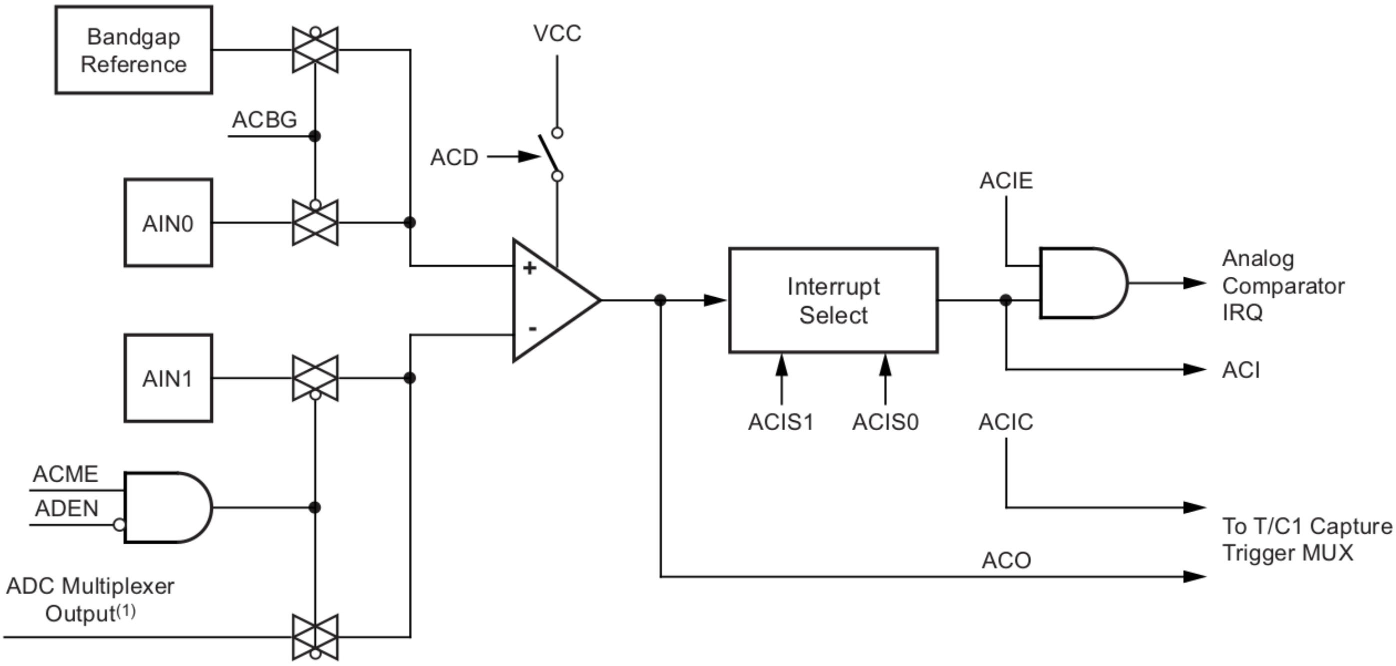
```
        {
                NUMBYTE = NUMBYTE - 1;
                // Enabling the Acknowledment bit for replying positive ACK
                TWCR |= (1<<TWEA);
                if(NUMBYTE==0)
                {
                        // disbale the Acknowledment bit for replying Negatice ACK for last byte
                        TWCR &= ~(1<<TWEA);
                }
                status = I2C_Slave_DataReceiveByte();
                if(status==0xFF)
                        return;
                else
                        recData[i] = status;
                i++;
        }
}
```

Analog Comparator

11.1 Overview

- The analog comparator compares the input values on the positive pin *AIN0* and negative pin *AIN1*.
- When the voltage on the positive pin *AIN0* is higher than the voltage on the negative pin *AIN1*, the analog comparator output, ***ACO*** bit is set.
- The comparator's output can be set to trigger the Timer/Counter1 input capture function.
- In addition, the comparator can trigger a separate interrupt, exclusive to the analog comparator.

11.2 Block Diagram



11.3 Analog Compartors Input

- One input is either be *AIN0* positive pin or Bandgap reference selected by ***ACBG*** bit.
- The other input can be either *AIN1* negative pin or any one of ADC multiplexed output selected by ***ACME***, ***ADEN*** and ***MUX[2:0]*** pins.

ACME	*ADEN*	*MUX[2:0]*	**Analog Compartor Negative Input**
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6
1	0	111	ADC7

11.4 Register Description

ADCSRB – ADC Control and Status Register B

7	6	5	4	3	2	1	0
-	ACME	-	-	-	ADTS2	ADTS1	ADTS0

ACSR – Analog Comparator Control and Status Register

7	6	5	4	3	2	1	0
ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

- ***ACD*** - Analog Comparator Disable - The power to analog comparator is switched off when this bit is set to one.
- ***ACBG*** - Analog Comparator Bandgap Select - [1 - Selects Bandgap reference as positive input to analog comparator; 0 - Selects *AIN0* as positive input to analog comparator]
- ***ACO*** - Analog Comparator Output - The actual output of Analog Comparator.
- ***ACI*** - Analog Comparator interrupt Flag - Set by hardware when compartor output event triggers the interrupt mode.
- ***ACIE*** - Analog Comparator interrupt Enable - Enabled the analog comparator interrupt.
- ***ACIC*** - Analog Comparator Input Capture Enable - Enables the input capture function in Timer/Counter1 to be triggered by analog comparator.

ACIS[1:0] **- Analog Comparator Interrupt Mode Select**	**Interrupt Mode**
00	Comparator interrupt on output toggle.
01	Reserved
10	Comparator interrupt on falling output edge.
11	Comparator interrupt on rising output edge.

11.5 Configuring the Analog Comparator

11.5.1 Using AIN1 as positive input and AIN0 as Negative Input

- First, the Analog Comparator Multiplexer Enable bit (***ACME***) in **ADCSRB** Register is diabled to select *AIN1* pin as positive input.
- Next, the Analog Comparator Bandgap Select bit (***ACBG***) in **ADCSRB** Register is diabled to select *AIN* pin as negative input.
- Next, the interrupt mode is selected by Configuring the ***ACIS[1:0]*** bit in **ADCSRB** register.
- The interupt for analog comparator is enabled by setting the ***ACIE*** bit in **ADCSRB** register.
- Finally, the Analog Comparator is swithched on by clearing the ***ACD*** bit in **ADCSRB** register.

- Also, the ISR is written for handling the interrupt.
- The code can be seen below:

```
// Disabling the Analog Comparator Multiplexer Enable bit so that AIN1 is selected as positive
↪ input
ADCSRB &= ~(1<<ACME);

// Disabling the Analog Comparator Bandgap Select bit so that AIN0 is selected as negative input
ACSR &= ~(1<<ACBG);

// Choosing the interrupt mode to toggle ACO bit
// By selecting 00 to ACIS[1:0]
ACSR &= ~(1<<ACIS1);
ACSR &= ~(1<<ACIS0);

// Enabling the Analog Comparator interrupt Enable to see the output
ACSR |= (1<<ACIE);

// enabling the Analog Comparator by clearing the Analog Comparator Disable bit
ACSR &= ~(1<<ACD);
sei();

ISR(ANALOG_COMP_vect)
{
    PINC |= (1<<0);
}
```

Analog to Digital Converter

12.1 Features

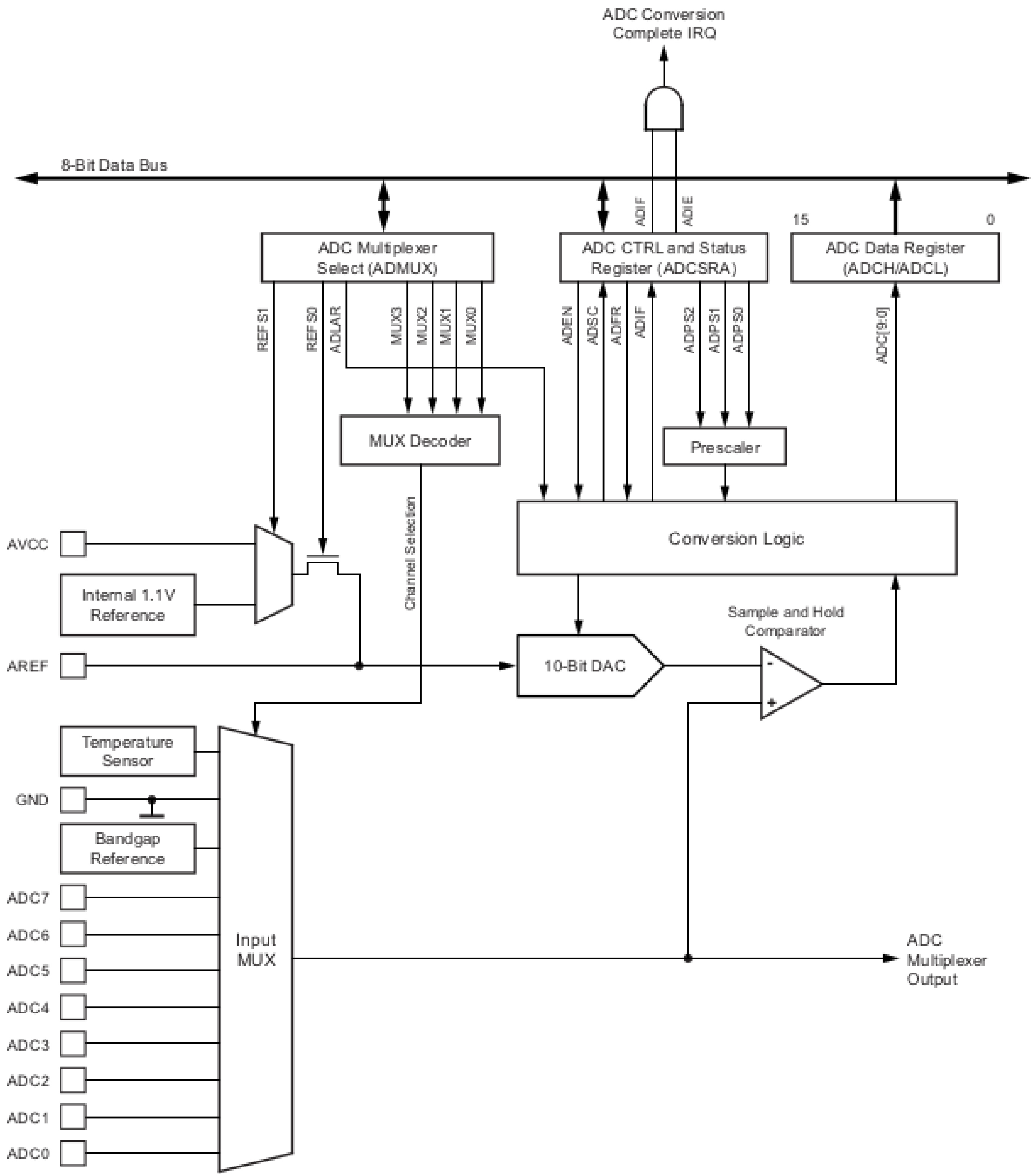
- 10-bit successive approximation ADC
- 65 to 260 μs conversion time
- 15 kilo samples per second
- 6 Multiplexed single ended input channels
- 2 Additional multiplexed single ended input channels depending on the package
- Temperature Sensor input Channel
- Selectable 1.1V ADC reference voltage
- Free running or singe conversion mode
- Interrupt on ADC conversion complete

12.2 Overview

- Minimum value = 0V and Maximum value = $V_{REF} - 1$ LSB
- AV_{CC} can should be $V_{CC}\pm$ 0.3V.
- The ***MUX*** bits in **ADMUX** register is used to select either ADC input pins or GND or Temperature Sensor or fixed band gap voltage reference(1.1V) for single ended input of ADC.
- The input clock frequency of ADC must be between 50kHz and 200kHz for max. resolution.
- Normal conversion takes 13 ADC clock cycles.
- The adc output are stored in **ADCH** and **ADCL** register.
- Can choose output between left or right adjusted by ***ADLAR*** bit in ADMUX.

Notes : First read **ADCH** and then read **ADCL**.

12.3 Block Diagram



12.4 Starting Conversion

12.4.1 Single Conversion

- Disabling the power reduction ADC bit (***PRADC***).
- Writing logical one to ADC start conversion bit (***ADSC***).
- This Start conversion bit is cleared by hardware when ADC completes conversion.

12.4.2 Triggered Conversion

- Many sources can be used to trigger.

- Auto trigger is enabled by setting ADC auto trigger enable bit(***ADATE***) in **ADCSRA** register.
- Trigger source is selected by ADC trigger select bits (***ADTS***) in **ADCSRB** register.
- When positve edge occur on selected trigger signal, the ADC starts conversion.
- Until the ADC conversion ends and another positve edge occur on selected trigger source, the next conversion wont's tart.

Free Running Mode

- Using ADC interrupt Flag as trigger source makes the ADC start new conversion as soon as ongoing conversion ends.
- This is the free running mode, when constant sampling and updating is done.
- The first conversion is started by setting the ***ADSC*** bit **ADCSRA** register.
- No need to clear interupt flag.

Note : The ***ADSC*** bit can be used to check if the conversion is going on or not independent of the mode.

12.5 Register Description

ADMUX – ADC Multiplexer Selection Register

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0

- ***ADLAR*** - ADC Left Adjust Result - presentation of ADC conversion results.[1 - Left adjusted; 0 - Right adjusted]

REFS[1:0]	Voltage Reference
00	AREF - the actual reference voltage
01	AV_{CC}
10	Reserved
11	Internal 1.1V

MUX[3:0]	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	Temperature Sensor
1110	1.1V Internal Voltage Reference
111	0V

ADCSRA – ADC Control and Status Register A

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0

- ***ADEN*** - ADC Enable - enabled the ADC.
- ***ADSC*** - ADC Start Conversion - starts the conversion in singe conversion mode and start first conversion in free running mode.
- ***ADATE*** - ADC Auto Trigger Enable - auto triggering the ADC on positve edge of selected trigger signal.
- ***ADIF*** - ADC Interrupt Flag - indicates the End of conversion.
- ***ADIE*** - ADC Interrupt Enable- enables the ADC conversion complete interrupt.

ADTS[2:0] - ADC Auto Trigger Source Selections	Trigger Source
000	Free running mode
001	Analog comparator
010	External interrupt request 0
011	Timer/Counter0 compare match A
100	Timer/Counter0 overflow
101	Timer/Counter1 compare match B
110	Timer/Counter1 overflow
111	Timer/Counter1 capture event

ADPS[2:0] - ADC Prescaler Select	Division Factor
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

ADCSRB – ADC Control and Status Register B

7	6	5	4	3	2	1	0
–	ACME	-	-	-	ADTS2	ADTS1	ADTS0

ADCL and ADCH – The ADC Data Register

ADLAR=0

15	14	13	12	11	10	9	8
-	-	-	-	-	-	ADC[9:8]	
ADC[7:0]							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**

ADLAR=1

15	14	13	12	11	10	9	8
ADC[9:2]							
ADC[1:0]		-	-	-	-	-	-
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**

12.6 Configuring the ADC

12.6.1 Single Conversion

- First, Voltage Reference is choosen by configuring the ***REFS[1:0]*** bits in **ADMUX** register.
- Next, the ADC output presentation either left or right adjusting is choosen by configuring the ***ADLAR*** bit in **ADMUX** register.
- Next, the channel is choosen by configuring the ***MUX[3:0]*** bits in **ADMUX** register.
- Next, for single conversion, the ***ADATE*** - ADC auto trigger bit is cleared in **ADCSRA** register.
- Interrupt is disbaled, as we use single conversion every time in program by clearing the ***ADIE*** bit in **ADCSRA** register.

- The Prescaler for ADC clock is choosen so that the clock is between 50kHz and 200kHz by Configuring the ***ADPS[2:0]*** bits in **ADCSRA** register.
- ADC is enabled by seting the ***ADEN*** bit in **ADCSRA** register.
- Finally, the ADC conversion is started by setting the ***ADSC*** bit in **ADCSRA** register.
- Next, we check the ***ADSC*** flag for end of conversion.
- We can read the output from ***ADC*** register.

```
DDRC &= ~(1<<channel_no);

// Selecting Voltage Referece
// Lets use AREF pin
// REFS[1:0] -- 00
ADMUX &= ~(1<<REFS0);
ADMUX &= ~(1<<REFS1);

// Selecting the Presentation of ADC output
// Right adjust - ADLAR == 0
ADMUX &= ~(1<<ADLAR);

// SELECTINT the channel for ADC
// LET'S select channel_no
// MUX[3:0]&0xF0 | channel_no
ADMUX = (ADMUX & 0XF0) | channel_no;

// for single conversion - disabling ADC auto trigger
// ADATE == 0
ADCSRA &= ~(1<<ADATE);

// disable the interrrupt by disbaling ADIE bit
// ADIE == 0
ADCSRA &= ~(1<<ADIE);

//  Prescaler be 64 so that we get 8Mhz/64 = 125kHz
// ADPS[2:0] -- 110
ADCSRA |= (1<<ADPS2) | (1<<ADPS1);
ADCSRA &= ~(1<<ADPS0);

// ENABLING adc
ADCSRA |= (1<<ADEN);

// STARTING CONVERSIOn
ADCSRA |= (1<<ADSC);

    // since single conversion, we can check start conversion bit
while((ADCSRA & (1<<ADSC)))
{

}
// RESSETTING THE Flag
// ADCSRA |= (1<<ADIF);
return ADC;
```

12.6.2 Free Running Conversion

- First, Voltage Reference is choosen by configuring the ***REFS[1:0]*** bits in **ADMUX** register.
- Next, the ADC output presentation either left or right adjusting is choosen by configuring the ***ADLAR*** bit in **ADMUX** register.
- Next, the channel is choosen by configuring the ***MUX[3:0]*** bits in **ADMUX** register.

- Next, the trigger source of auto trigger is choosen by selecting 000 (free running) in ***ADTS[2:0]*** bits in **ADCSRA** register.
- Next, for Free Running conversion, the ***ADATE*** - ADC auto trigger bit is set in **ADCSRA** register.
- Interrupt is enabled by setting the ***ADIE*** bit in **ADCSRA** register.
- The Prescaler for ADC clock is choosen so that the clock is between 50kHz and 200kHz by Configuring the ***ADPS[2:0]*** bits in **ADCSRA** register.
- ADC is enabled by seting the ***ADEN*** bit in **ADCSRA** register.
- Finally, the ADC conversion is started by setting the ***ADSC*** bit in **ADCSRA** register.
- Next, we write a ISR for handling the End of conversion.

```
DDRC &= ~(1<<channel_no);

// Selecting Voltage Referece
// Lets use AREF pin
// REFS[1:0] -- 00
ADMUX &= ~(1<<REFS0);
ADMUX &= ~(1<<REFS1);

// Selecting the Presentation of ADC output
// Right adjust - ADLAR == 0
ADMUX &= ~(1<<ADLAR);

// SELECTINT the channel for ADC
// LET'S select channel_no
// MUX[3:0]&0xF0 | channel_no
ADMUX = (ADMUX & 0XF0) | channel_no;

// Select the Auto Trigger source
// for free running, use 000 for ADTS[2:0] in ADCSRB
ADCSRB &= ~(1<<ADTS2);
ADCSRB &= ~(1<<ADTS1);
ADCSRB &= ~(1<<ADTS0);

// for free runing conversion - enable ADC auto trigger
// ADATE == 1
ADCSRA |= (1<<ADATE);

// enable the interrrupt by enabling ADIE bit
// ADIE == 1
ADCSRA |= (1<<ADIE);

//  Prescaler be 64 so that we get 8Mhz/64 = 125kHz
// ADPS[2:0] -- 110
ADCSRA |= (1<<ADPS2) | (1<<ADPS1);
ADCSRA &= ~(1<<ADPS0);

// ENABLING adc
ADCSRA |= (1<<ADEN);

// STARTING CONVERSIOn
ADCSRA |= (1<<ADSC);

sei();


ISR(ADC_vect)
{
        free_running_value = ADC;
        // ADCSRA |= (1<<ADIF);
}
```

Miscellaneous

"1" - Unprogrammed — "0" - Programmed

13.1 Fuse Bits

- Atmeta328P Has three fuse Byte.

13.1.1 Extended Fuse Byte

7	6	5	4	3	2	1	0
-	-	-	-	-	BODLEVEL2	BODLEVEL1	BODLEVEL0

- V_{BOT} - Brown-out Threshold Voltage

BODLEVEL[2:0]	**Min** V_{BOT}	**Typ** V_{BOT}	**Max** V_{BOT}
111	BOD disabled		
101	2.5	2.7	2.9
100	4.0	4.3	4.6

13.1.2 High Fuse Byte

7	6	5	4	3	2	1	0
RSTDISBL	DWEN	SPIEN	WDTON	EESAVE	BOOTSZ1	BOOTSZ0	BOOTRST

High Fuse Byte	Description	Default Value
RSTDISBL	External reset disable - disable external reset pin and use as input or output pin	1
DWEN	debugWIRE enable - enabled debugWIRE interface	1
SPIEN	Enable serial program and data downloading	0 - enable SPI programming
WDTON	Watchdog timer always On	1
EESAVE	EEPROM memory is preserved through chip erase	1
BOOTSZ1	Select boot size	0
BOOTSZ0	Select boot size	0
BOOTRST	Select reset vector - select reset vector location	1

BOOTSZ[1:0]	Boot size	Pages	Application Flash Section	Boot Loader Falash Section
11	256 words	4	0x0000 - 0x3EFF	0x3F00 - 0x3FFF
10	512 words	8	0x0000 - 0x3DFF	0x3E00 - 0x3FFF
01	1024 words	16	0x0000 - 0x3BFF	0x3C00 - 0x3FFF
00	2048 words	32	0x0000 - 0x37FF	0x3800 - 0x3FFF

13.1.3 Low Fuse Byte

7	6	5	4	3	2	1	0
CKDIV8	CKOUT	SUT1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0

High Fuse Byte	Description	Default Value
CKDIV8	Divide clock by 8	0 - Divide clock by 8 and use as CPU clock
CKOUT	Clock output	1
SUT1	Select start-up time	1
SUT0	Select start-up time	0
CKSEL3	Select clock source	0
CKSEL2	Select clock source	0
CKSEL1	Select clock source	1
CKSEL0	Select clock source	1

CKSEL[3:0]	Device Clocking Option
1111 - 1000	Low power crystall oscillator
0111 - 0110	Full swing crystal oscillator
0101 - 0100	Low frequency crystal oscillator
0011	Internal 128kHz RC oscillator
0010	Calibrated internal RC oscillator
0000	External clock

13.2 Signature Bytes

- All Atmel microcontrollers have a three-byte Signature code which identifies the devices.

Part	Sinature Bytes Address		
	0x000	0x001	0x002
ATmega328P	0x1E	0x95	0x0F

13.3 Calibration Byte

- The Atmel ATmega328P has a byte calibration value for the internal RC oscillator.
- This byte resides in the high byte of address 0x000 in the signature address space.
- During reset, this byte is automatically written into the **OSCCAL** register to ensure correct frequency of the calibrated RC oscillator.

Bibliography

[1] AVR rogramming. https://ccrma.stanford.edu/wiki/AVR_Programming.

[2] AVRDUDE. https://www.nongnu.org/avrdude/user-manual/avrdude.html.

[3] Compiler Optimize Options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[4] Tool Chain Overview. https://www.nongnu.org/avr-libc/user-manual/overview.html.