Counter/Timers in the Mega8

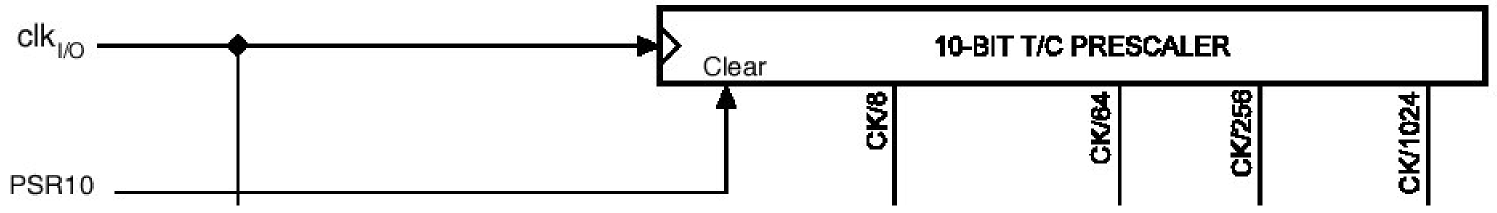
The mega8 incorporates three counter/timer devices. These can:

- Be used to count the number of events that have occurred (either external or internal)
- Act as a clock
- Trigger an interrupt after a specified number of events

Timer 0

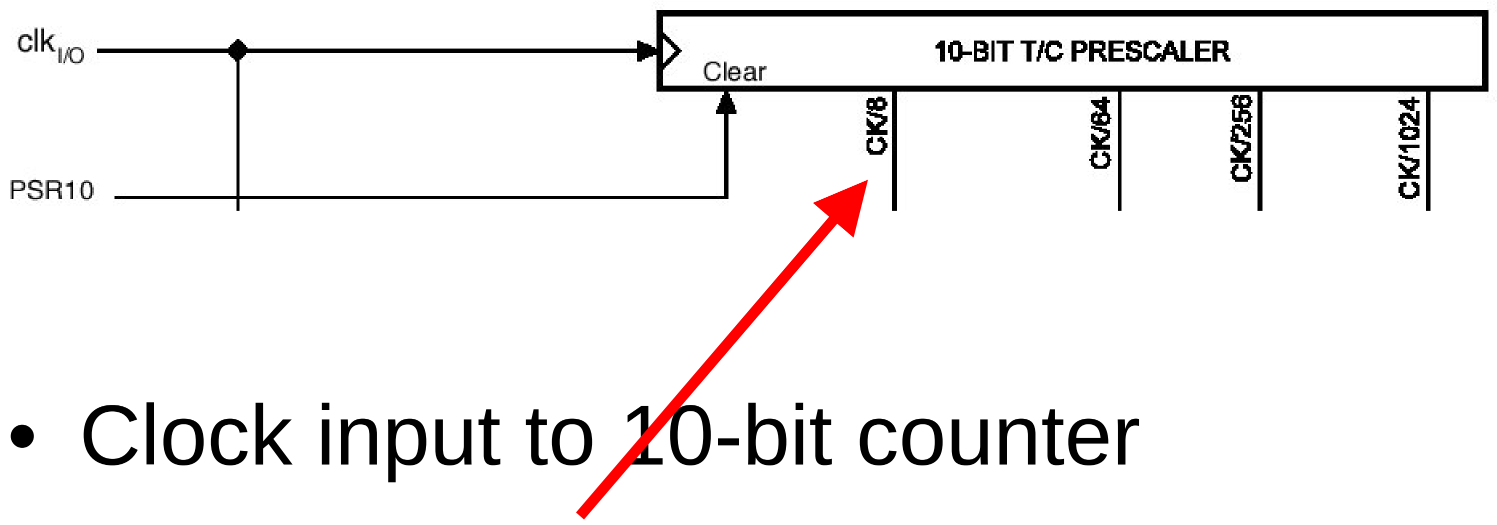
- Possible input sources:
  - Pin T0 (PD4)
  - System clock
    - Potentially divided by a “prescaler”
- 8-bit counter
- When the counter turns over from 0xFF to 0x0, an interrupt can be generated

Timer 0 Implementation



- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

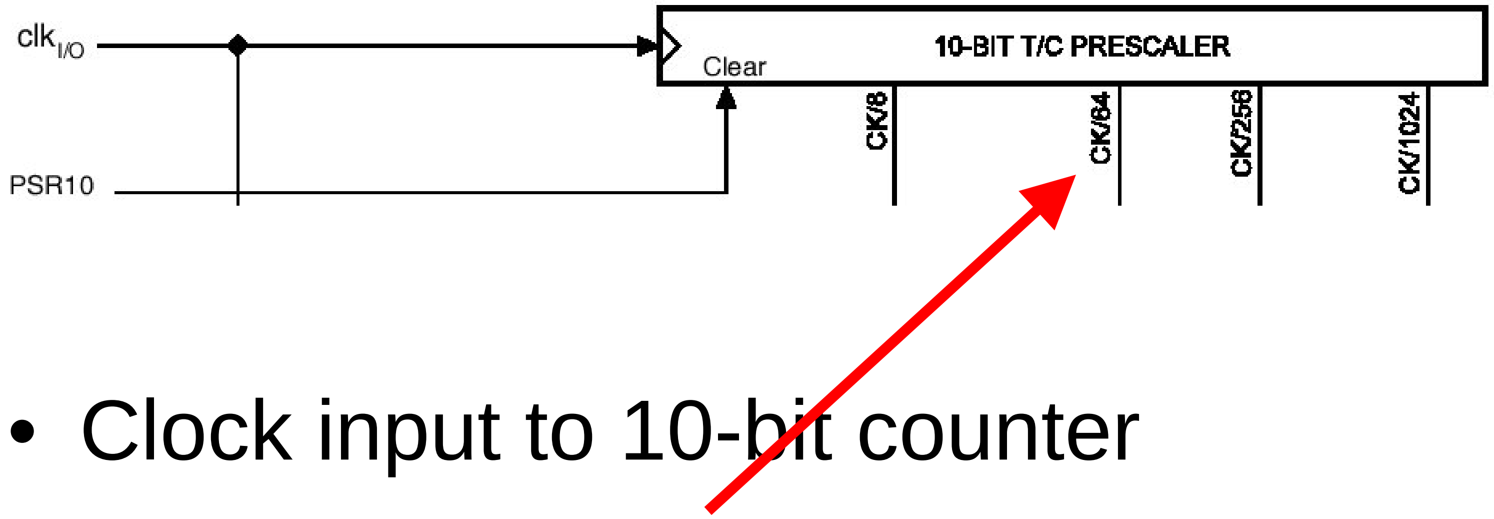
Timer 0 Implementation



- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

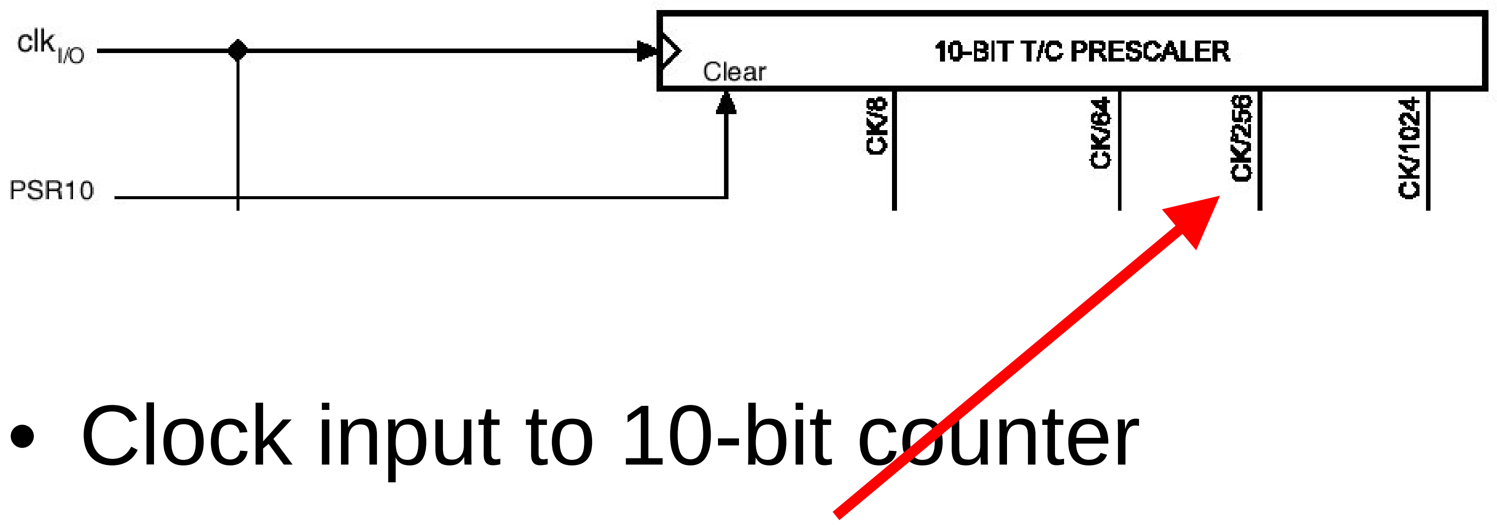
  (counting from 1)

Timer 0 Implementation



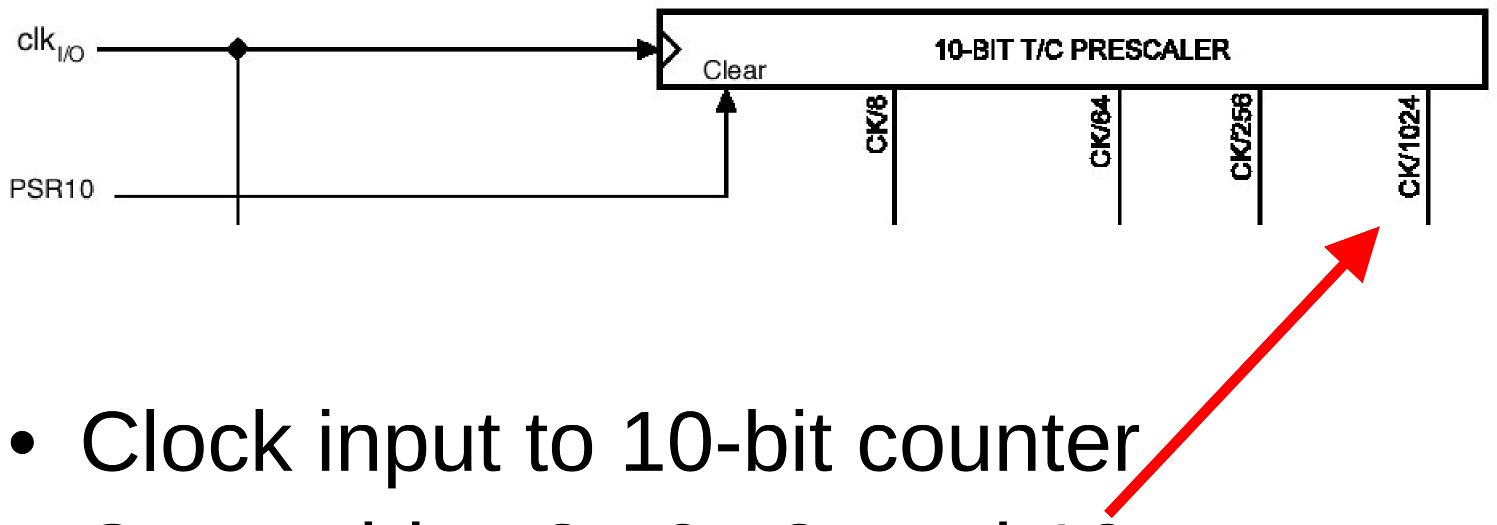
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



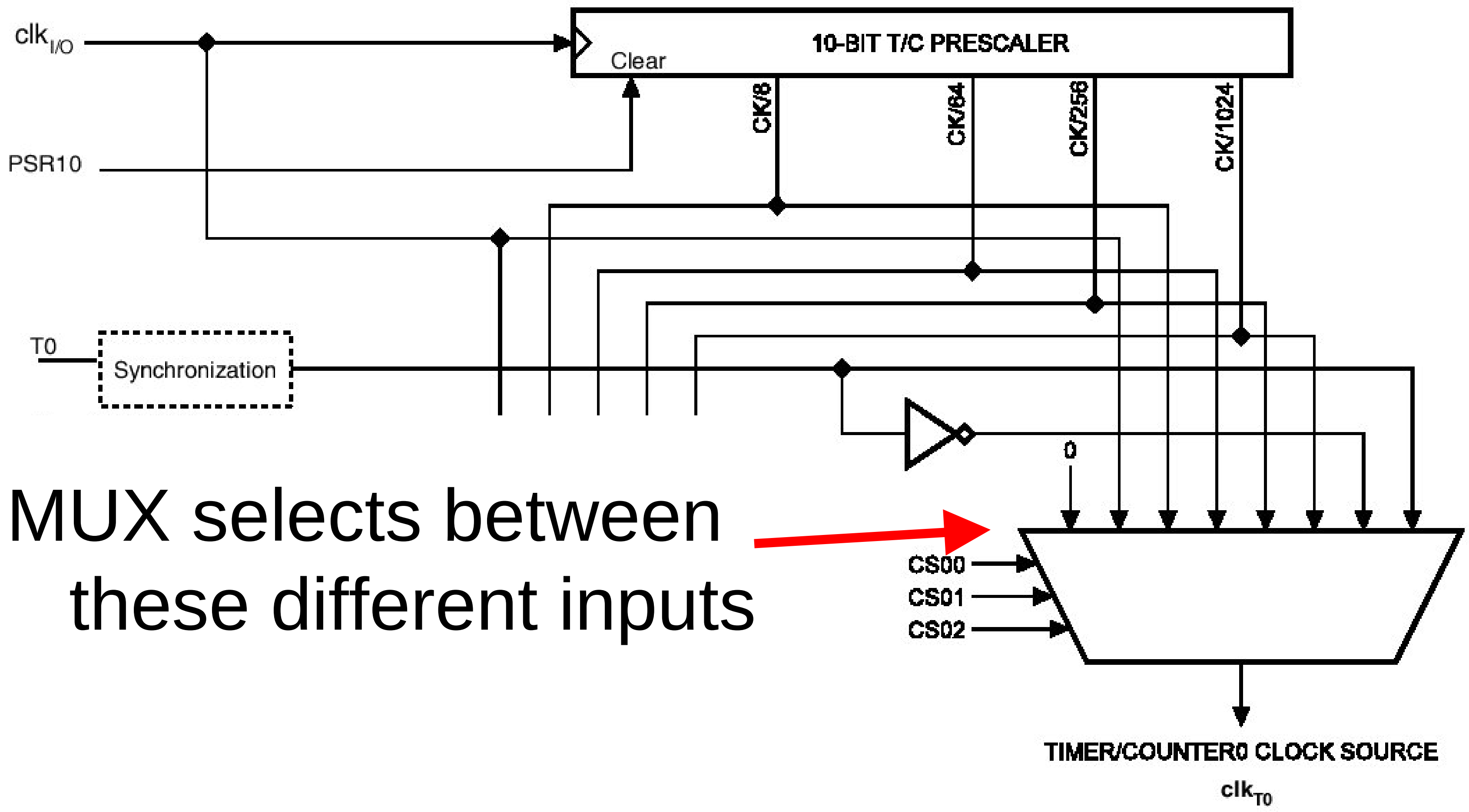
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



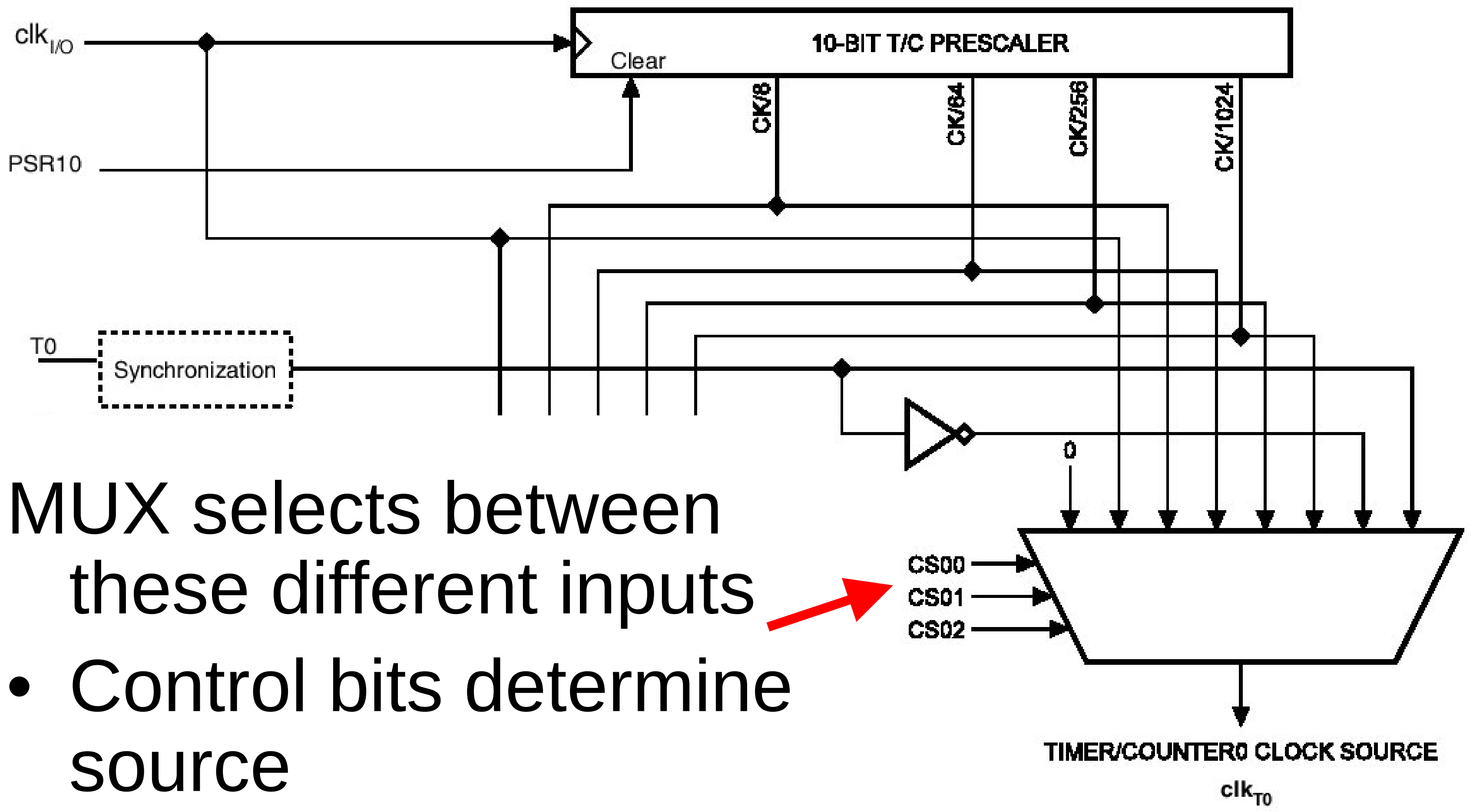
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10
  - These serve to divide the clock by the specified number of counts

Timer 0 Implementation



MUX selects between these different inputs

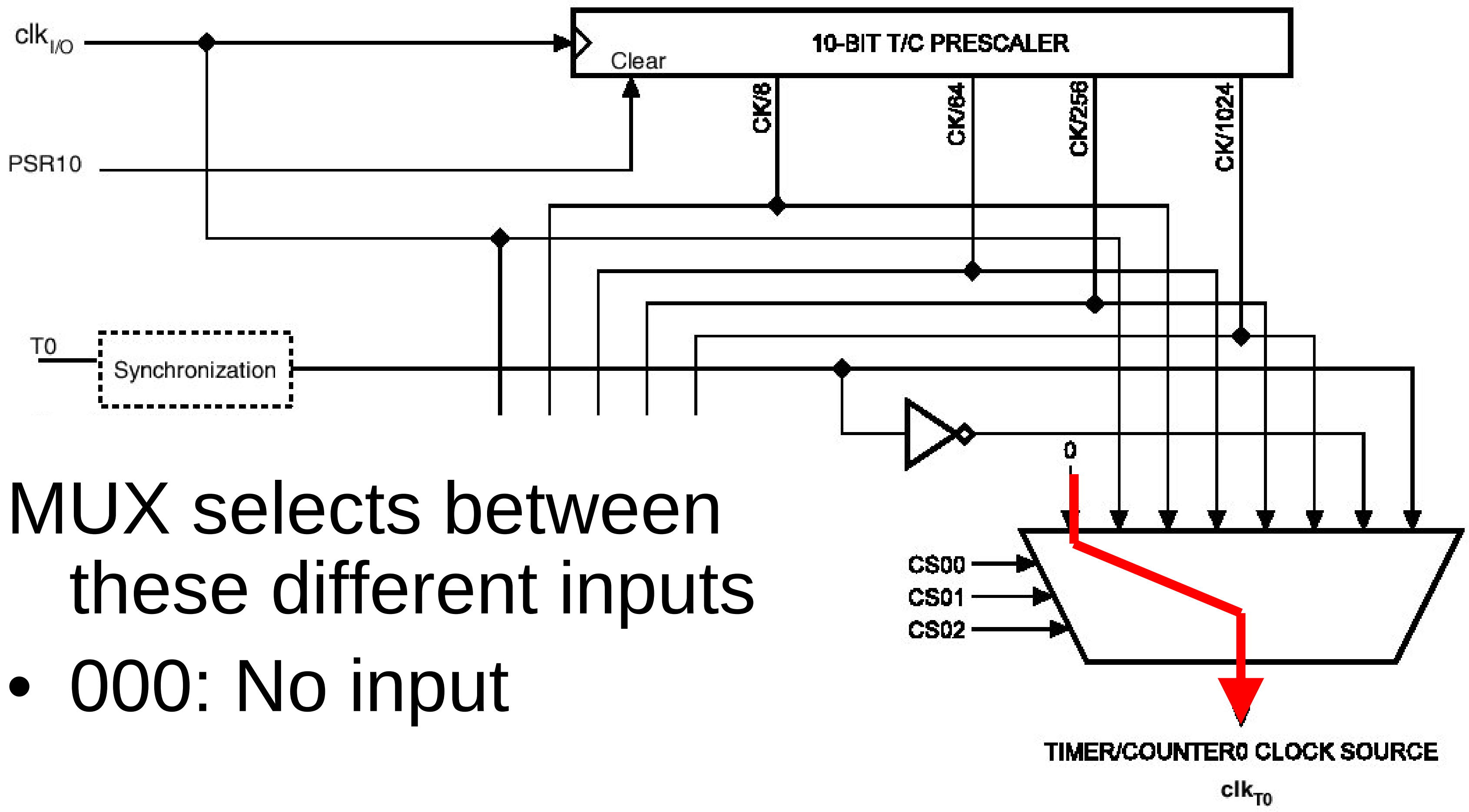
Timer 0 Implementation



MUX selects between these different inputs

- Control bits determine source

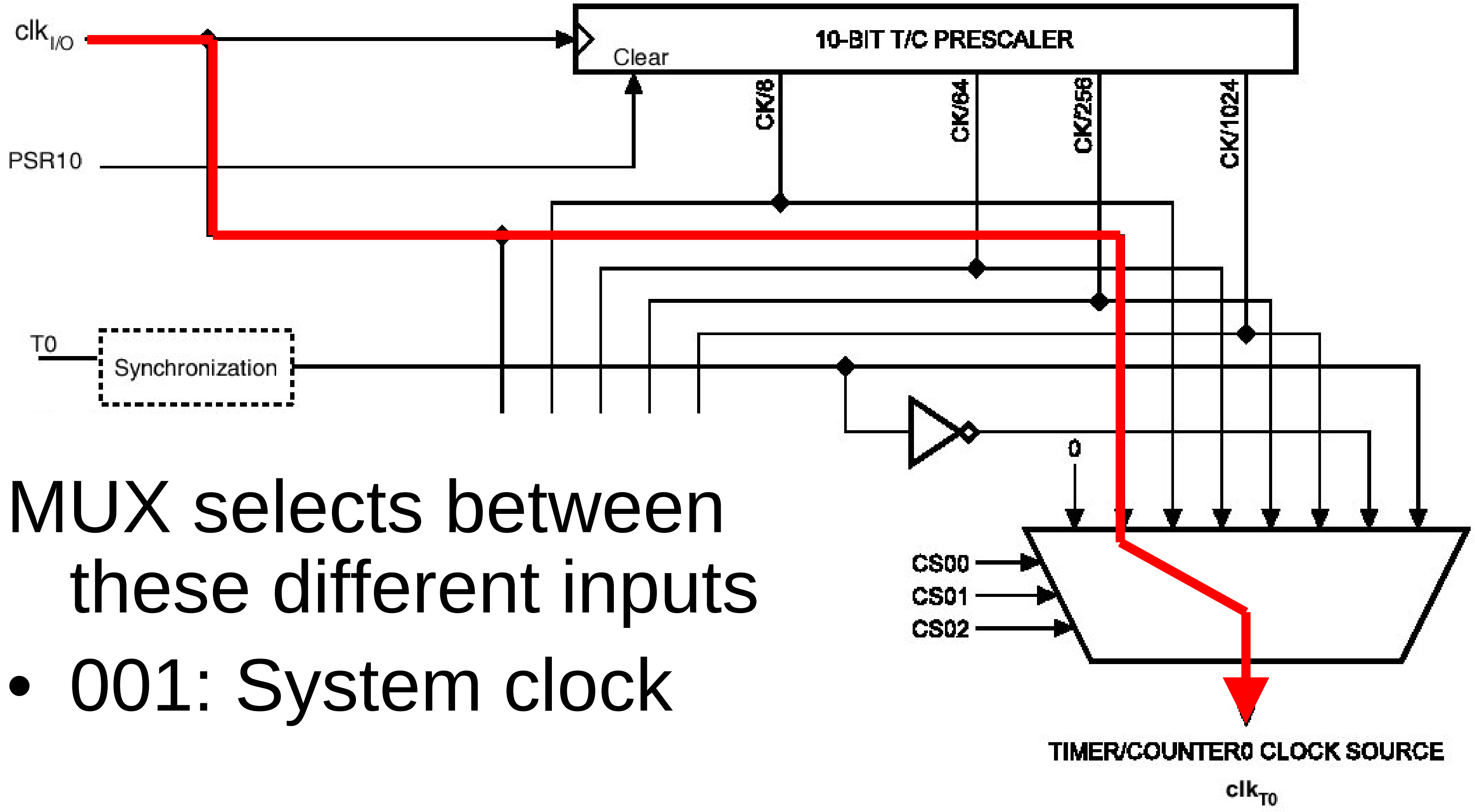
Timer 0 Implementation



MUX selects between these different inputs

- 000: No input

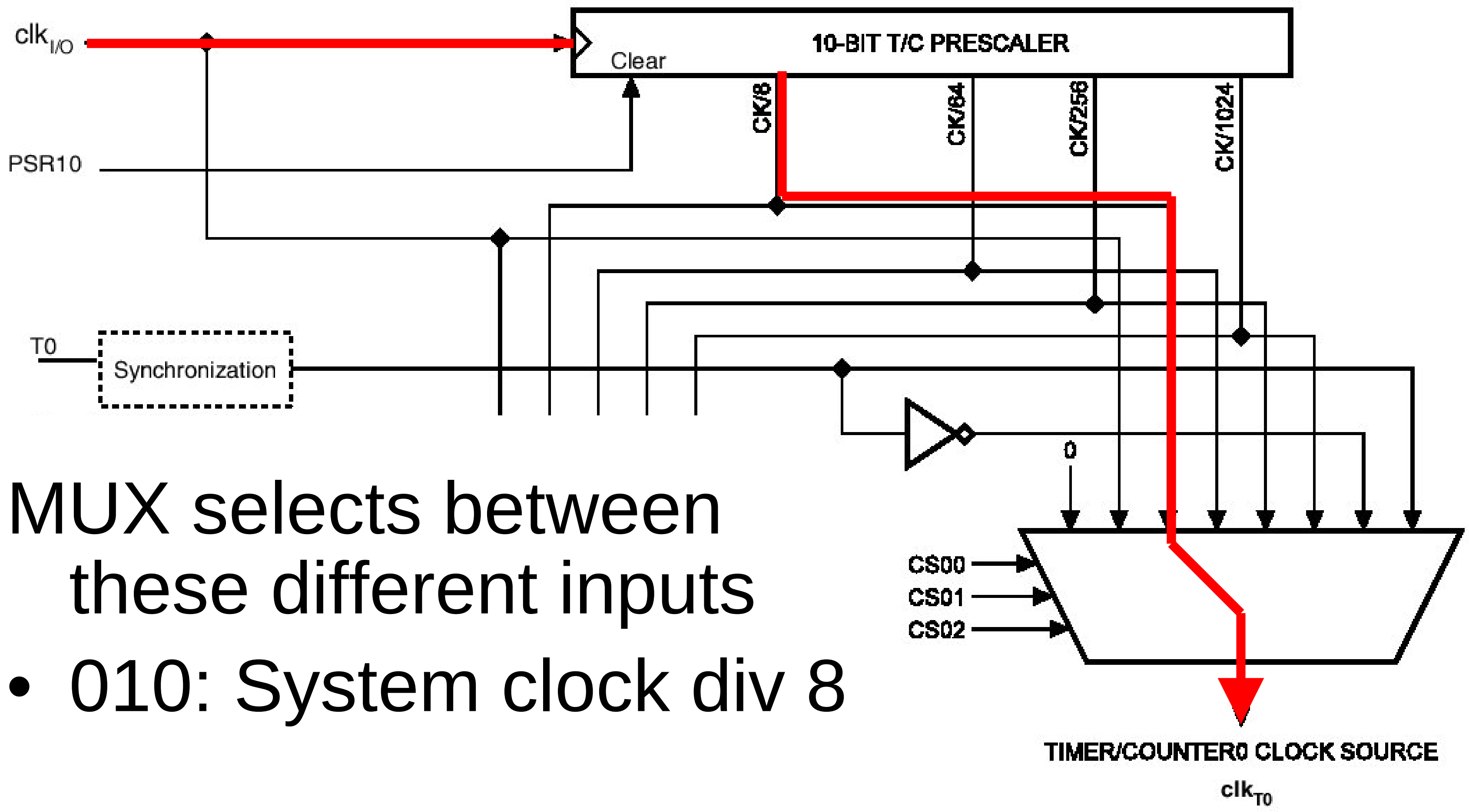
Timer 0 Implementation



MUX selects between these different inputs

- 001: System clock

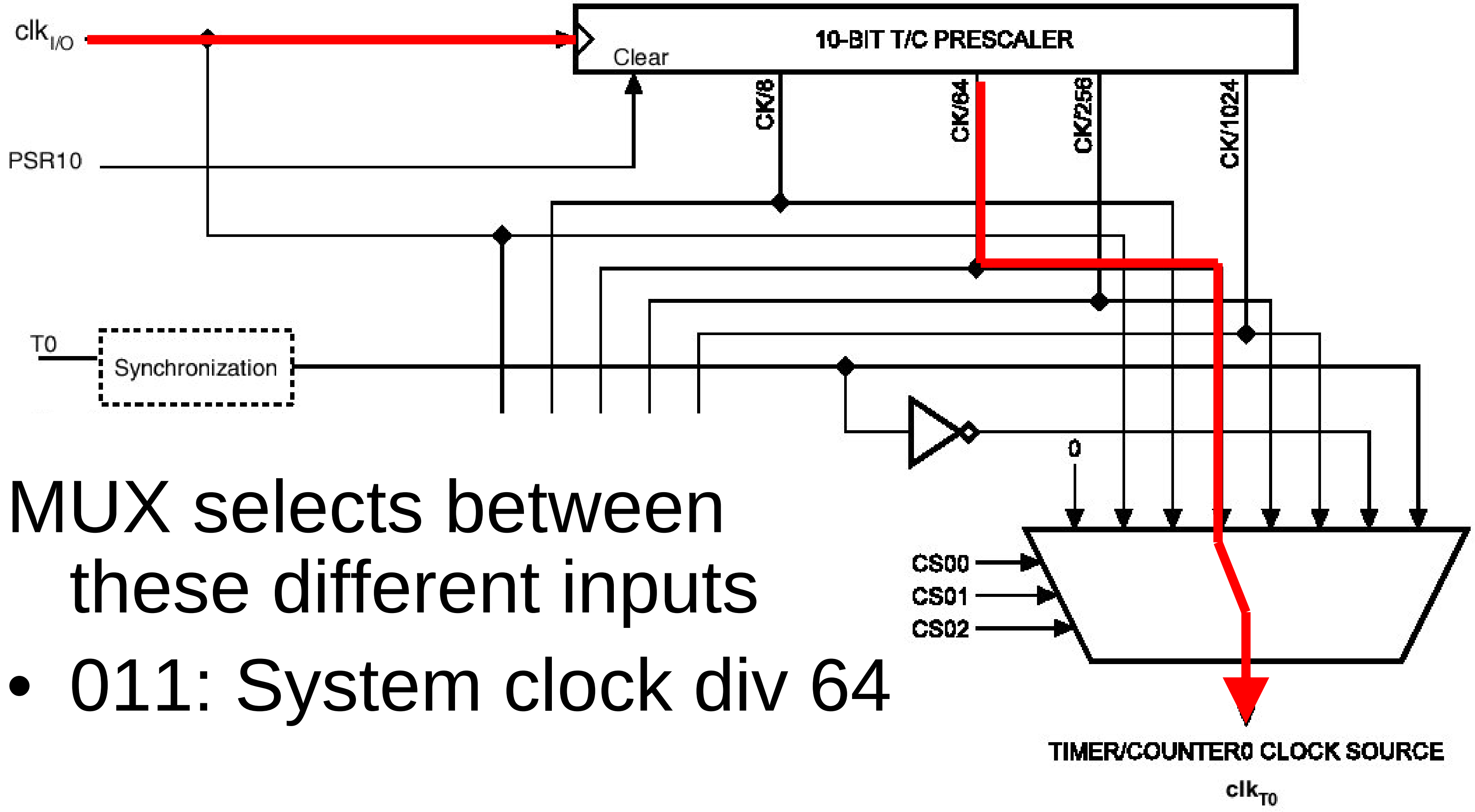
Timer 0 Implementation



MUX selects between these different inputs

- 010: System clock div 8

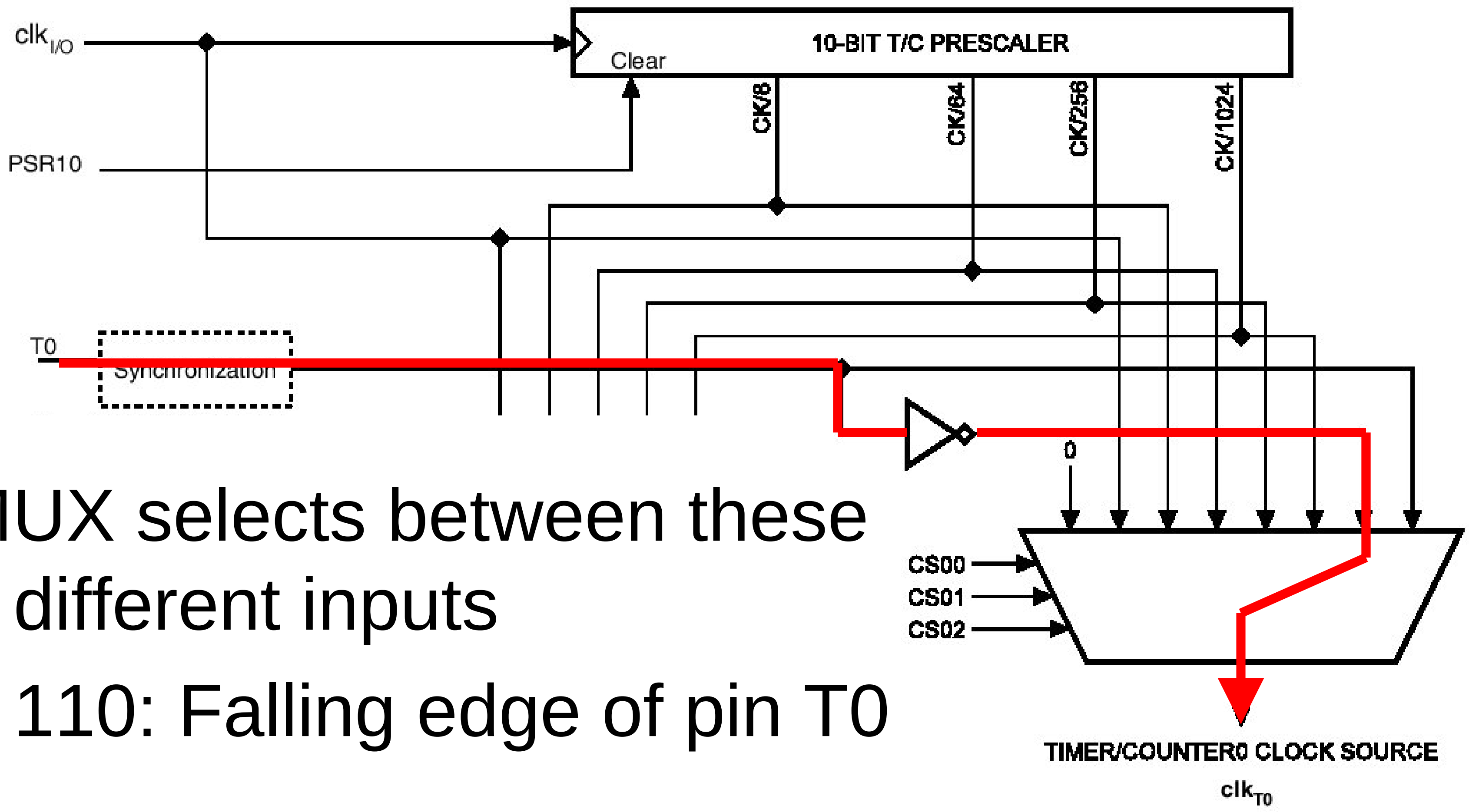
Timer 0 Implementation



MUX selects between these different inputs

- 011: System clock div 64

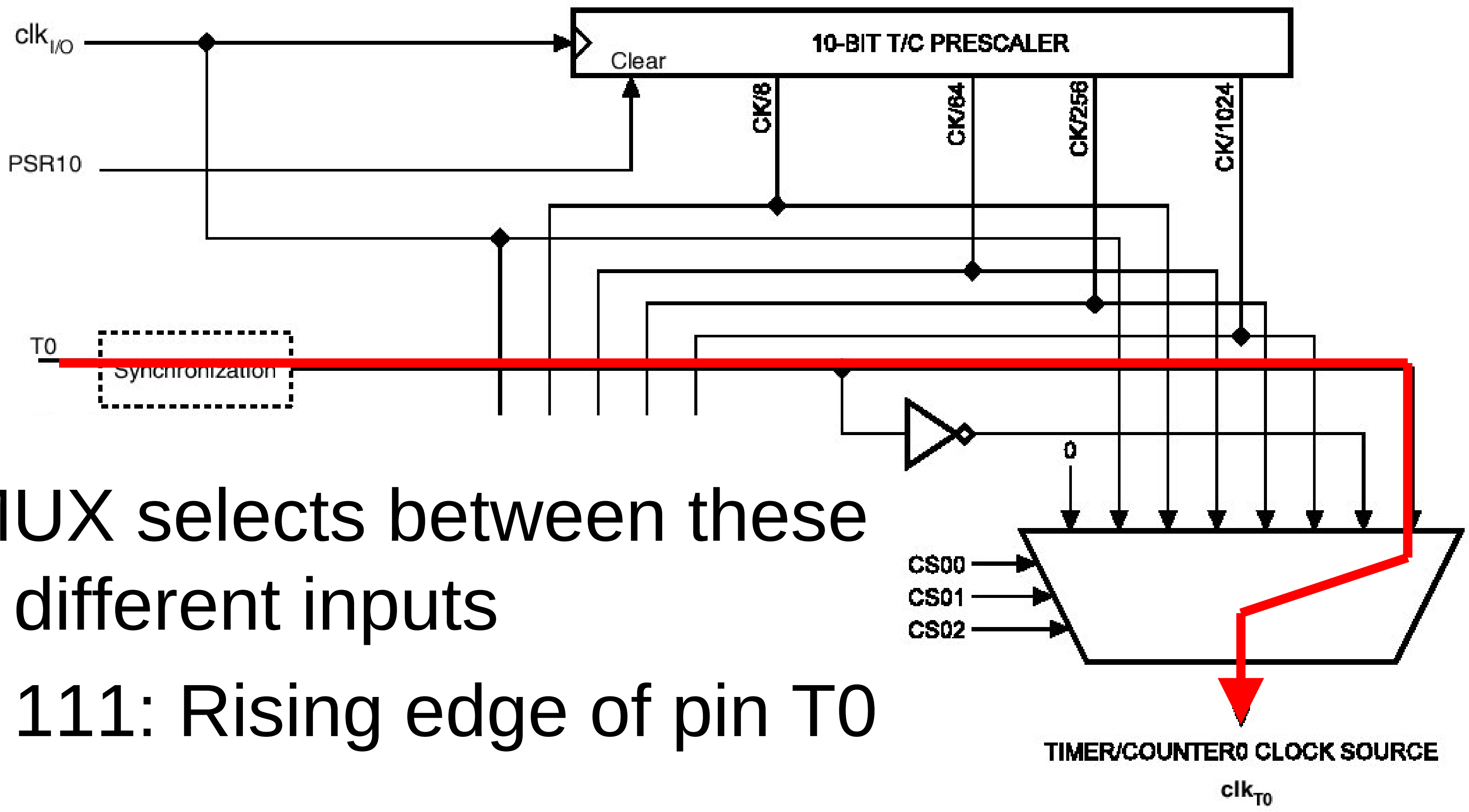
Timer 0 Implementation



MUX selects between these different inputs

- 110: Falling edge of pin T0

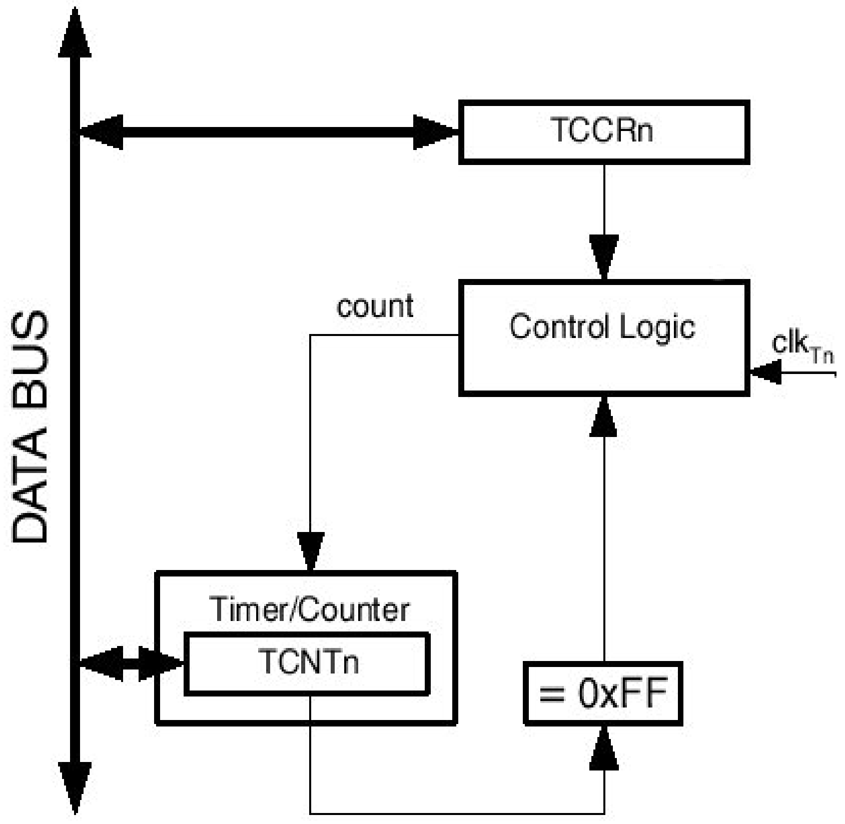
Timer 0 Implementation



MUX selects between these different inputs

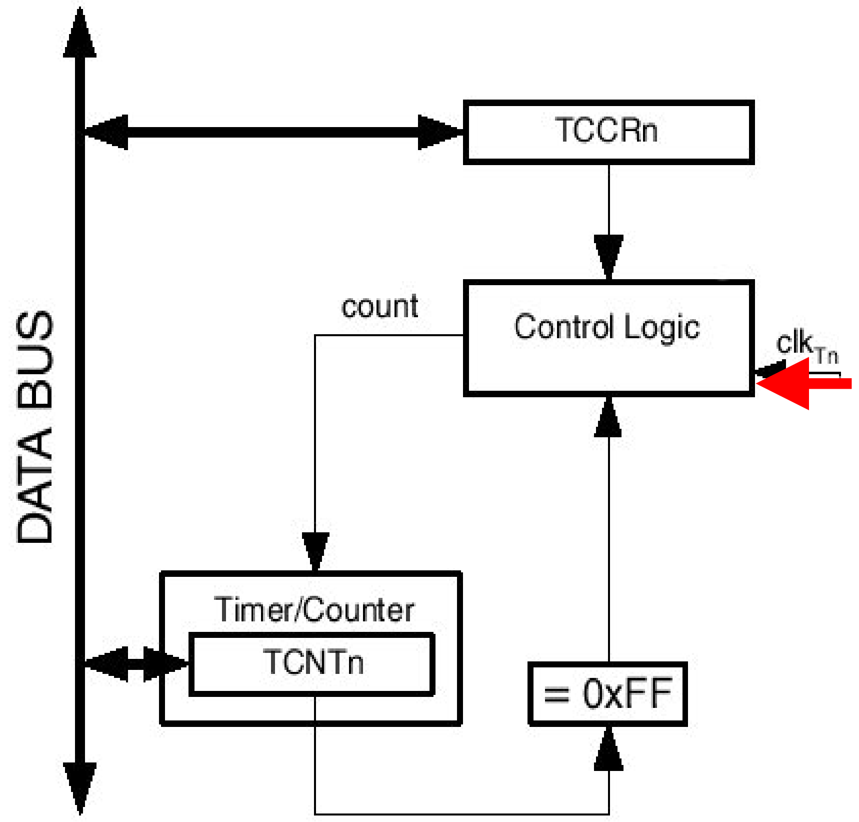
- 111: Rising edge of pin T0

Timer 0



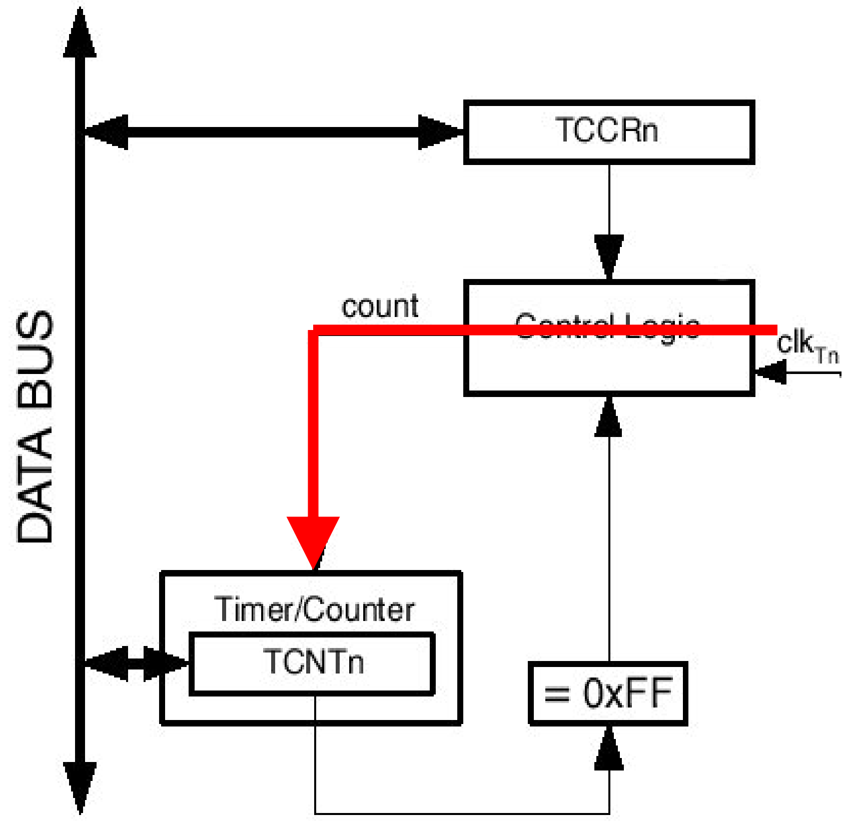
- TCNT0: 8-bit counter (a register)
- TCCR0: control register

Timer 0



- Clock source from previous slide

Timer 0



- Increment counter on every low-to-high transition

Timer 0 Example

Suppose:

- 16MHz clock
- Prescaler of 1024
- We wait for the timer to count from 0 to 156

How long does this take?

Timer 0 Example

$$delay = \frac{1024 * 156}{16{,}000{,}000} = 9948\ \mu s \approx 10\ ms$$

Timer 0 Example

Suppose:

- 16MHz clock
- Prescaler of 1024
- We wait for the timer to count from 0 to 156

How long does this take?

Timer 0 Example

$$delay = \frac{1024 * 156}{16,000,000} = 9948\ \mu s \approx 10\ ms$$

Timer 0 Code Example

```
timer0_config(TIMER0_PRE_1024);   // Prescale by 1024
timer0_set(0);       // Set the timer to 0

// Do something else for a while
while(timer0_read() < 156) {
   // Do something while waiting
};

// Break out at ~10 ms
```

See Atmel HOWTO for example code (timer_demo2.c)

Cascade of Clock Divisors

- Prescalar: 1 to 1024
- Timer 0 counter: up to 256
  - In this case, our software waited for timer 0 to achieve a particular value
- Other timers can choose their divisor arbitrarily (more on this soon)

Timer 0 Example

Advantage over delay_ms():

- Can do other things while waiting
- Timing is much more precise
  - We no longer rely on a specific number of instructions to be executed
  - Interrupts do not interfere with the timing

Timer 0 Example

Disadvantage:

- “something else” cannot take very much time

What is the solution?

Timer 0 Interrupt

What is the solution?

- Use interrupts!

- We can configure the timer to generate an interrupt every time the timer’s counter rolls over from 0xFF to 0x00

Timer 0 Example II

Suppose:

- 16MHz clock
- Prescaler of 1024

How often is the interrupt generated?

Timer 0 Example II

$$interval = \frac{1024 * 256}{16{,}000{,}000} = 16.384\ ms$$

How many counts do we need so that we toggle the state of PB0 every second?

Timer 0 Example II

How many counts do we need so that we toggle the state of PB0 every second?

$$counts = \frac{1000\ ms}{16.384\ ms} = 61.0352$$

We will assume 61 is close enough.

Example II: Interrupt Service Routine (ISR)

```
ISR(TIMER0_OVF_vect) {
    ++counter;
    if(counter == 61) {
          // Toggle output state every 61st interrupt:
          //  This means: on for ~1 second and then off for ~1 sec
          PORTB ^= 1;
          counter = 0;
    };
};
```

See Atmel HOWTO for example code (timer_demo.c)

Example II: Initialization

```
// Initialize counter
counter = 0;

// Interrupt occurs every (1024*256)/16000000 = .016384 seconds
timer0_config(TIMER0_PRE_1024);

// Enable the timer interrupt
timer0_enable();

// Enable global interrupts
sei();

 while(1) {
   // Do something else
 };
```

Timer 0 with Interrupts

This solution is particularly nice:

- “something else” does not have to worry about timing at all
  - PB0 state is altered asynchronously

- Note that we **can** have a shared data problem (but not in this example)

Cascade of Clock Divisors

- Prescalar: 1 to 1024
- Timer 0 counter: 256
  - Other timers can choose their divisor arbitrarily
- Software: arbitrary

Two Other Timers

Timer 1:

- 16 bit counter

Timer 2:

- 8 bit counter

Interrupt Service Routines

- Should be **very** short
  - No “delays”
  - No busy waiting
  - Function calls from the ISR should be short also
  - Minimize looping
- Communication with the main program using global variables

Interrupts, Shared Data and Compiler Optimizations

- Compilers (including ours) will often optimize code in order to minimize execution time
- These optimizations often pose no problems, but can be problematic in the face of interrupts and shared data

Shared Data and Compiler Optimizations

For example:

```
A = A + 1;
C = B * A
```

Will result in ‘A’ being fetched from memory once (into a general-purpose register) – even though ‘A’ is used twice

Shared Data and Compiler Optimizations

Now consider:

```
while(1) {
  PORTB = A;
}
```

What does the compiler do with this?

Shared Data and Compiler Optimizations

The compiler will assume that ‘A’ never changes.

This will result in code that looks something like this:

```
R1 = A;  // Fetch value of A into register 1
while(1) {
   PORTB = R1;
}
```

The compiler only fetches A from memory once!

Shared Data and Compiler Optimizations

This optimization is generally fine – but consider the following interrupt routine:

```
ISR(TIMER0_OVF_vect){
  A = PIND;
}
```

- The global variable ‘A’ is being changed!
- The compiler has no way to anticipate this

Shared Data and Compiler Optimizations

The fix: the programmer must tell the compiler that it is not allowed to assume that a memory location is not changing

- This is accomplished when we declare the global variable:

```
volatile uint8_t A;
```

Information Encoding

Many different options for encoding information for transmission to/from other devices:

- Parallel digital (e.g., for our Project 1)
- Serial digital (e.g., USB, RS232)
- Analog: use voltage to encode a value

Information Encoding

An alternative: pulse-width modulation (PWM)

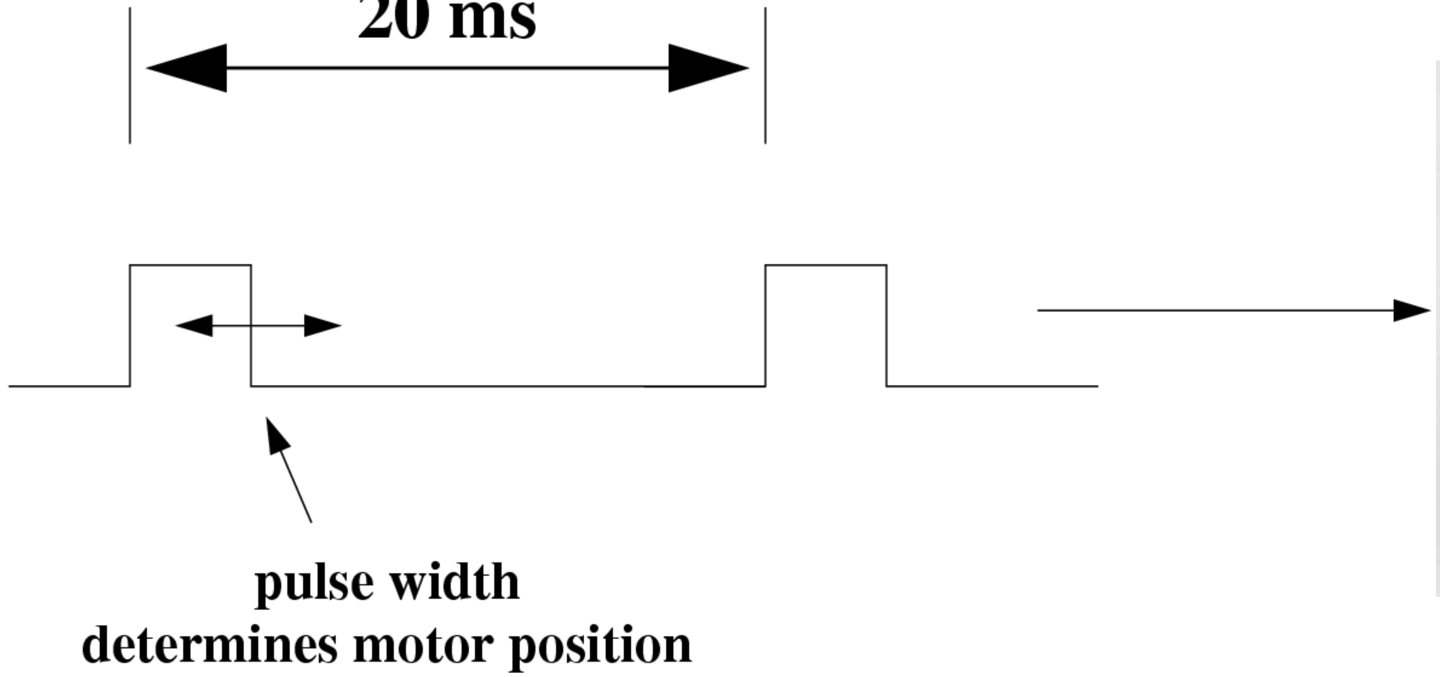
- Information is encoded in the time between the rising and falling edge of a pulse

PWM Example:

RC Servo Motors

- 3 pins: power (red), ground (black), and command signal (white)
- Signal pin expects a PWM signal

PWM Example



Internal circuit translates pulse width into a goal position:

- 0.5 ms: 0 degrees
- 2.5 ms: 180 degrees

RC Servo Motors

- Internal potentiometer measures the current orientation of the shaft
- Uses a **Position Servo Controller**: the difference between current and commanded shaft position determines shaft velocity.
- Mechanical stops limit the range of motion
  - These stops can be removed for unlimited rotation

PWM Example II: Controlling LED Brightness

What is the relationship of current flow through an LED and the rate of photon emission?

Controlling LED Brightness

What is the relationship of current flow through an LED and the rate of photon emission?

- They are linearly related (essentially)

Controlling LED Brightness

Suppose we pulse an LED for a given period of time with a digital signal: what is the relationship between pulse width and number of photons emitted?

Controlling LED Brightness

Suppose we pulse an LED for a given period of time with a digital signal: what is the relationship between pulse width and number of photons emitted?

- Again: they are linearly related (essentially)

- If the period is short enough, then the human eye will not be able to detect the flashes

Controlling LED Brightness

We need:

- To produce a periodic behavior, and
- A way to specify the pulse width (or the duty cycle)

How do we implement this in code?

Controlling LED Brightness

How do we implement this in code?

One way:

- Interrupt routine increments an 8-bit counter
- When the counter is 0, turn the LED on
- When the counter reaches some "duration", turn the LED off

```
volatile uint8_t counter = 0;
volatile uint8_t duration = 0;

ISR(TIMER0_OVF_vect)
{

}
```

Back to Our Interrupt Implementation …

```
volatile uint8_t counter, duration;

ISR(TIMER0_OVF_vect) {
  ++counter;
  if(counter == 0)
     PORTB |= 1;
  if(counter >= duration)
     PORTB &= ~1;
}
```

Initialization Details

- Set up timer
- Enable interrupts
- Set duration in some way
  - In this case, we will slowly increase it

What does this implementation look like?

Initialization

```
int main(void) {
   DDRB = 0xFF;
   PORTB = 0;

   // Initialize counter
   counter = 0;
   duration = 0;

   // Interrupt configuration
   timer0_config(TIMER0_NOPRE);  // No prescaler
   // Enable the timer interrupt
   timer0_enable();
   // Enable global interrupts
   sei();
                 :
```

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

- The timer0 counter (8 bits) expires every 256 clock cycles

$$t = \frac{256}{16000000} = 16\ \mu s$$

(assuming a 16MHz clock)

PWM Implementation

What is the period of the pulse?

PWM Implementation

What is the period of the pulse?

- The 8-bit counter (of the interrupt) expires every 256 interrupts

$$t = \frac{256 * 256}{16000000} = 4.096\, ms$$

Doing “Something Else”

```
          :
  unsigned int i;
  while(1) {
    for(i = 0; i < 256; ++i)
        duration = i;
        delay_ms(50);
    };
  };
}
```

Timer 1

- 16 bit counter
  - All the same functionality as we see with timer 0
- One **input capture** unit
  - On an external event, save the state of the counter
- Two **output compare** units
  - Generate an event when the counter reaches a certain state

Timer 1

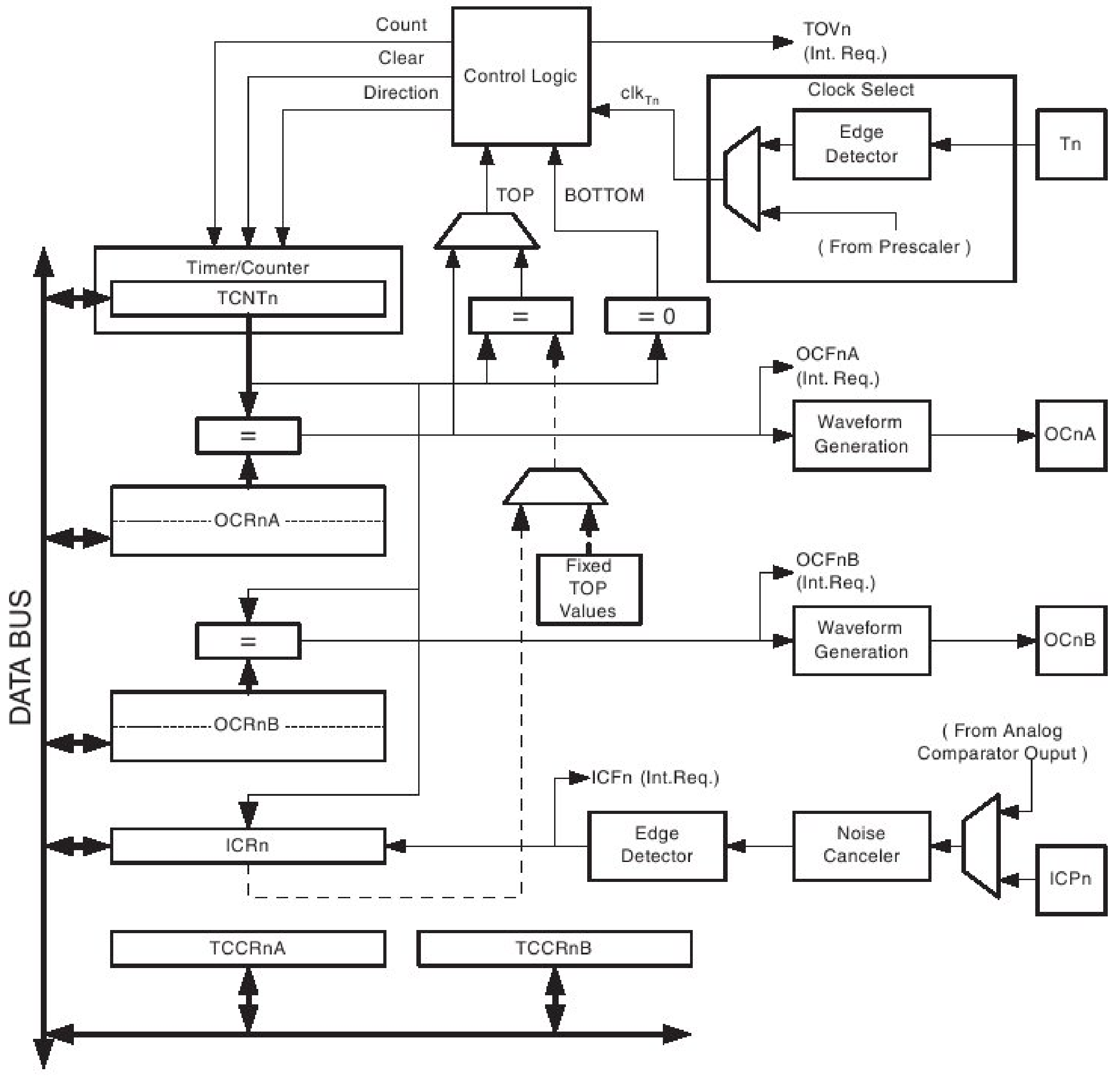
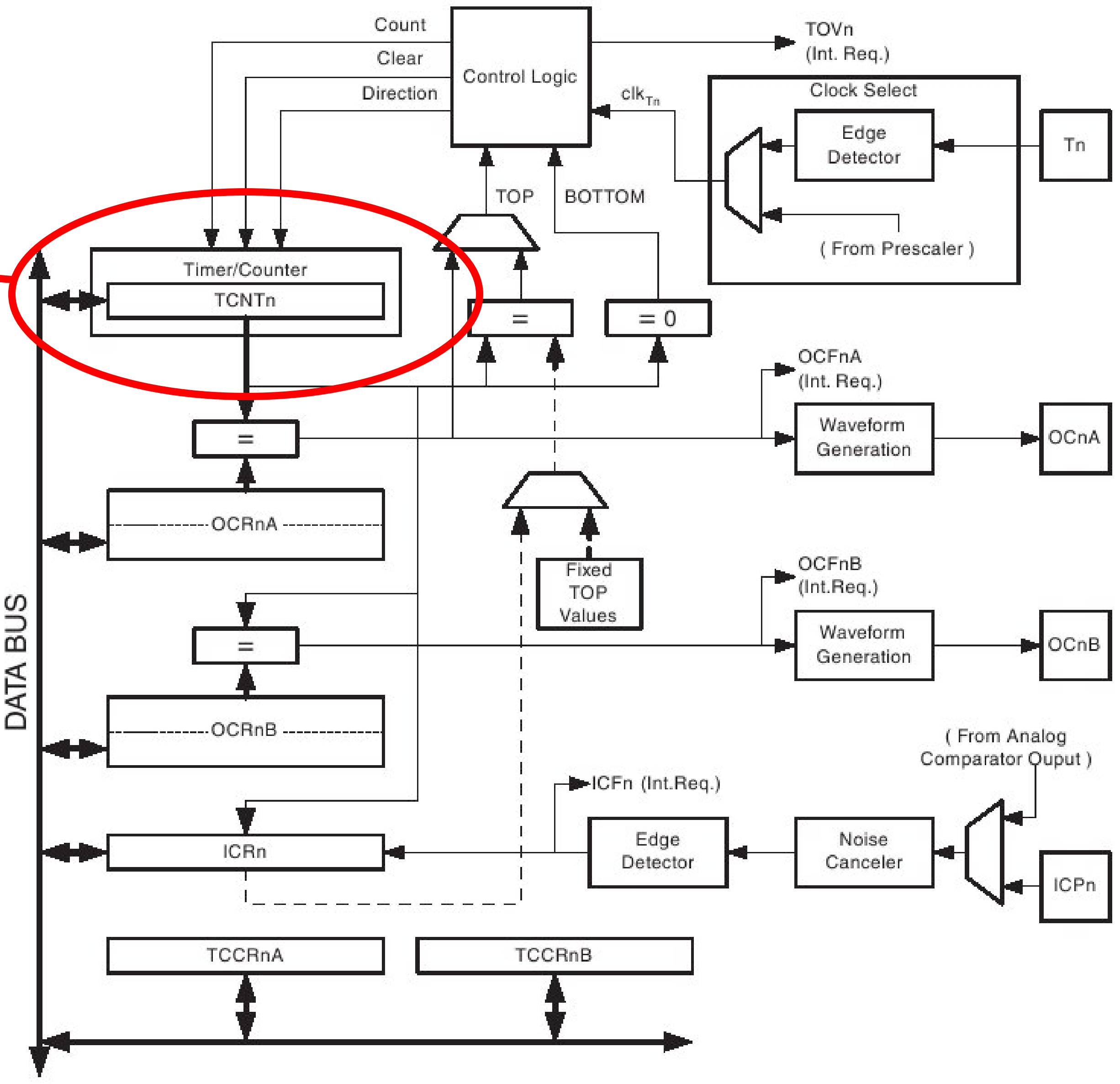


Figure from: Atmel mega 8 specification

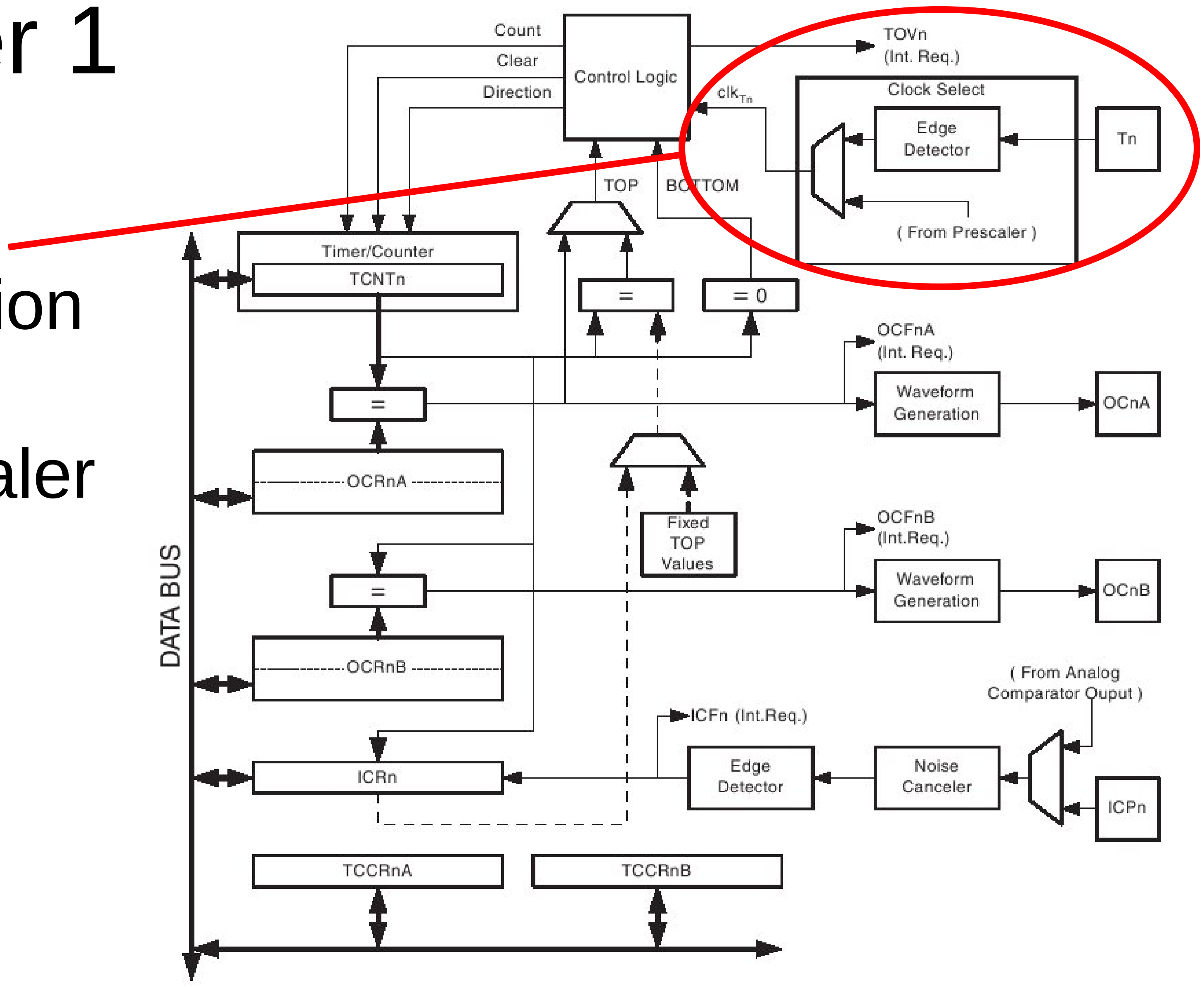
Timer 1

Counter



Timer 1

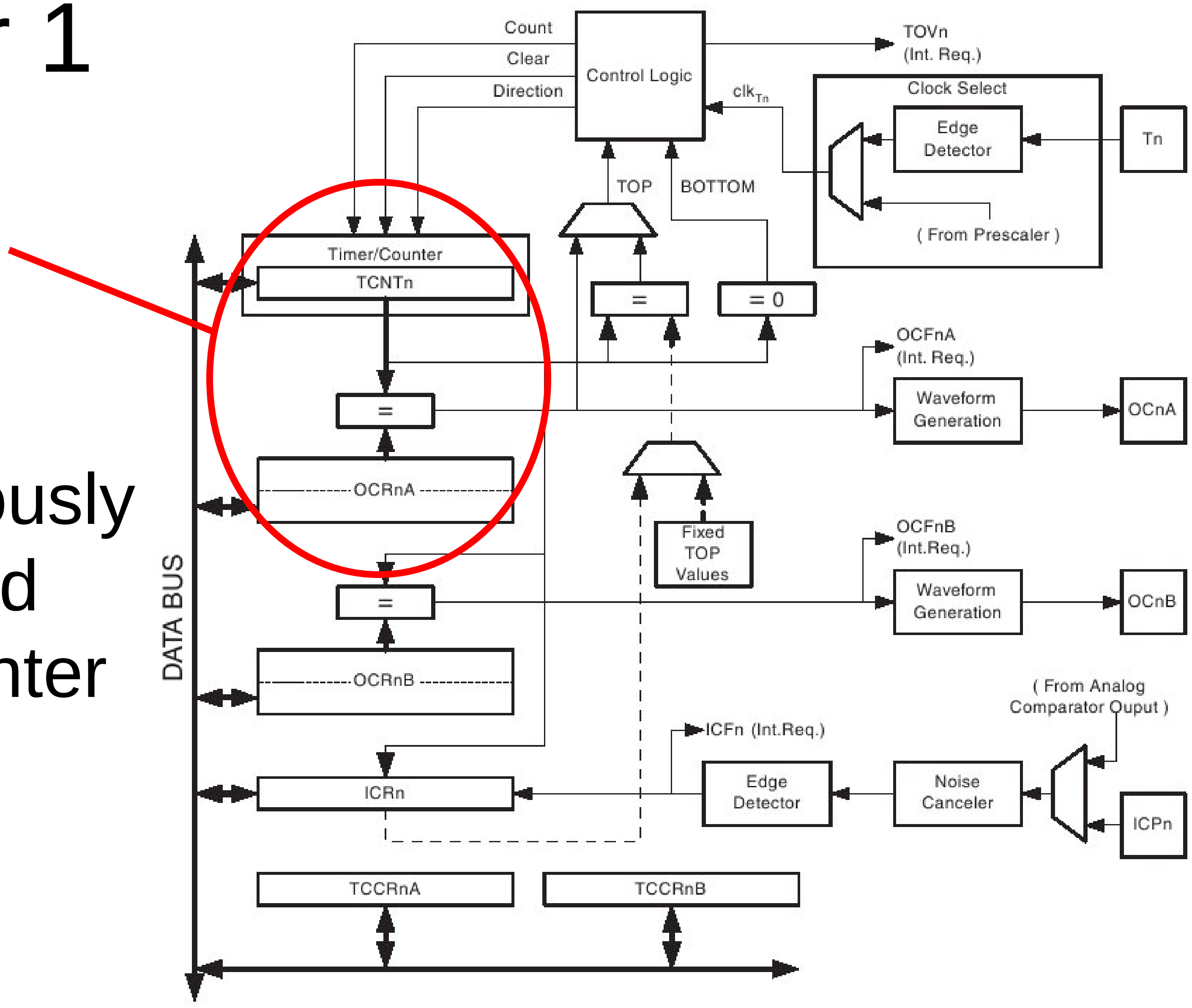
Source selection and prescaler



Timer 1

Output compare register

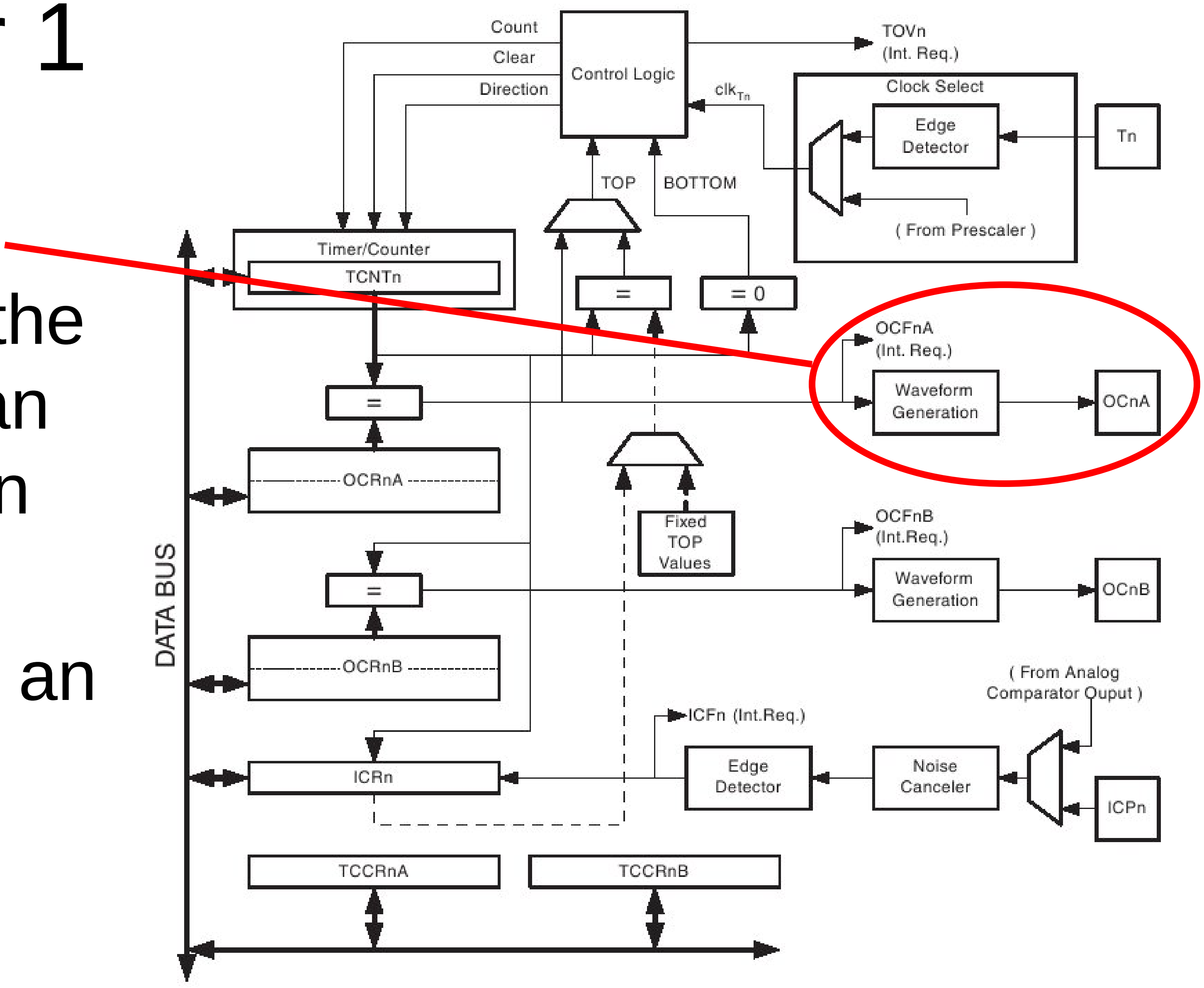
- Continuously compared with counter



Timer 1

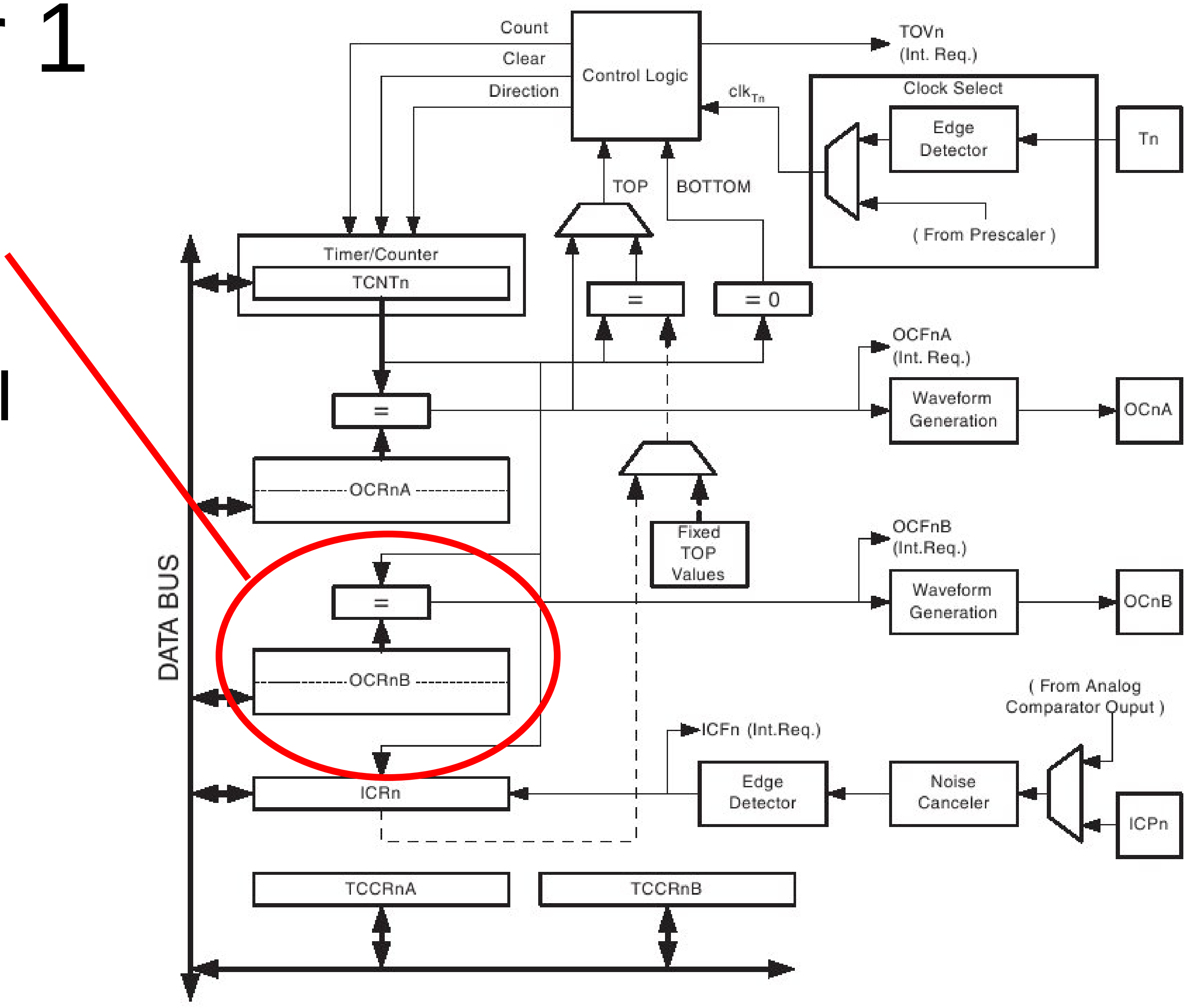
On match:

- Change the state of an output pin
- And/or generate an interrupt



Timer 1

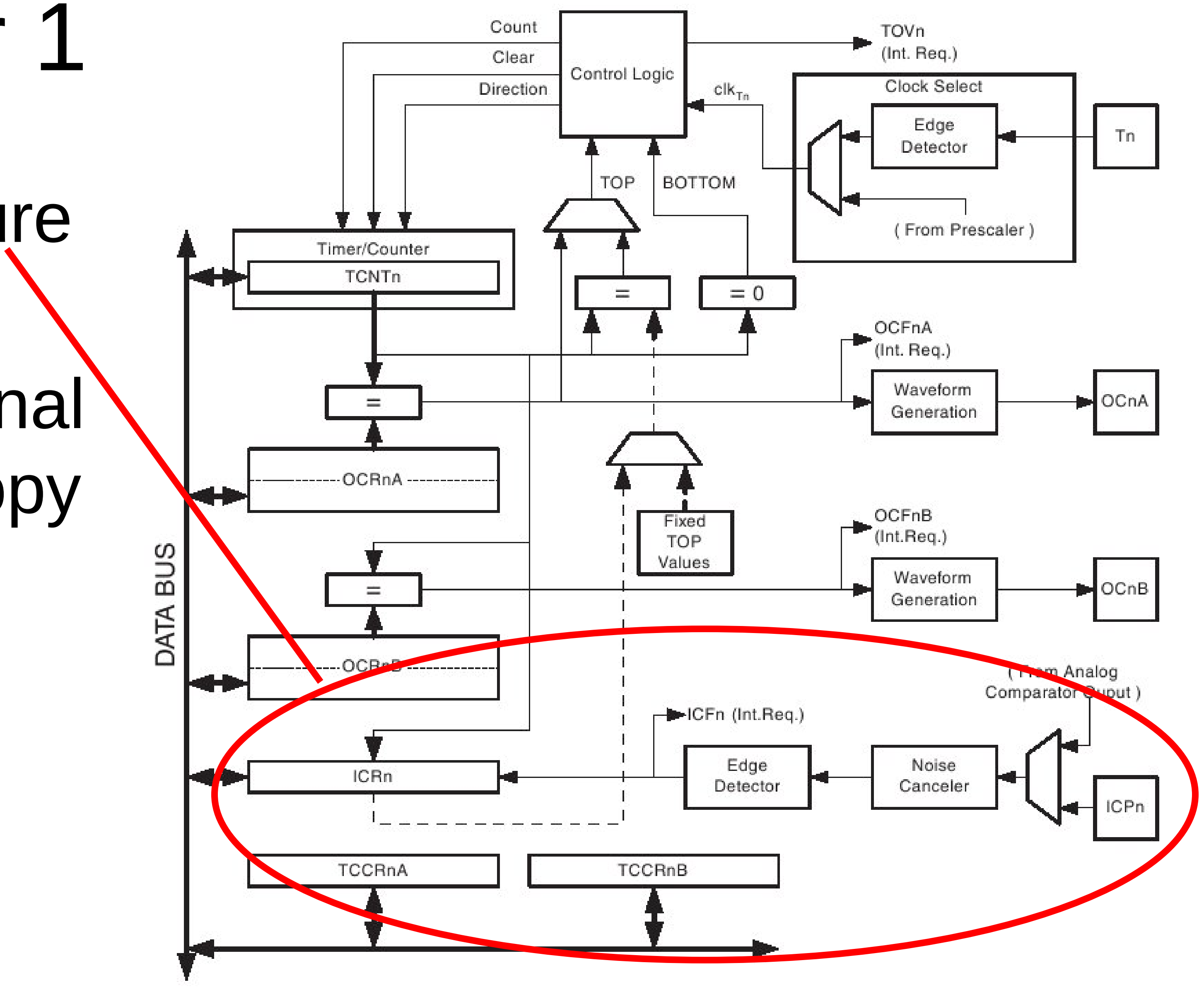
Output compare register II



Timer 1

Input capture register:

- On external event, copy state of counter



Timer 1: Register Access and Timing

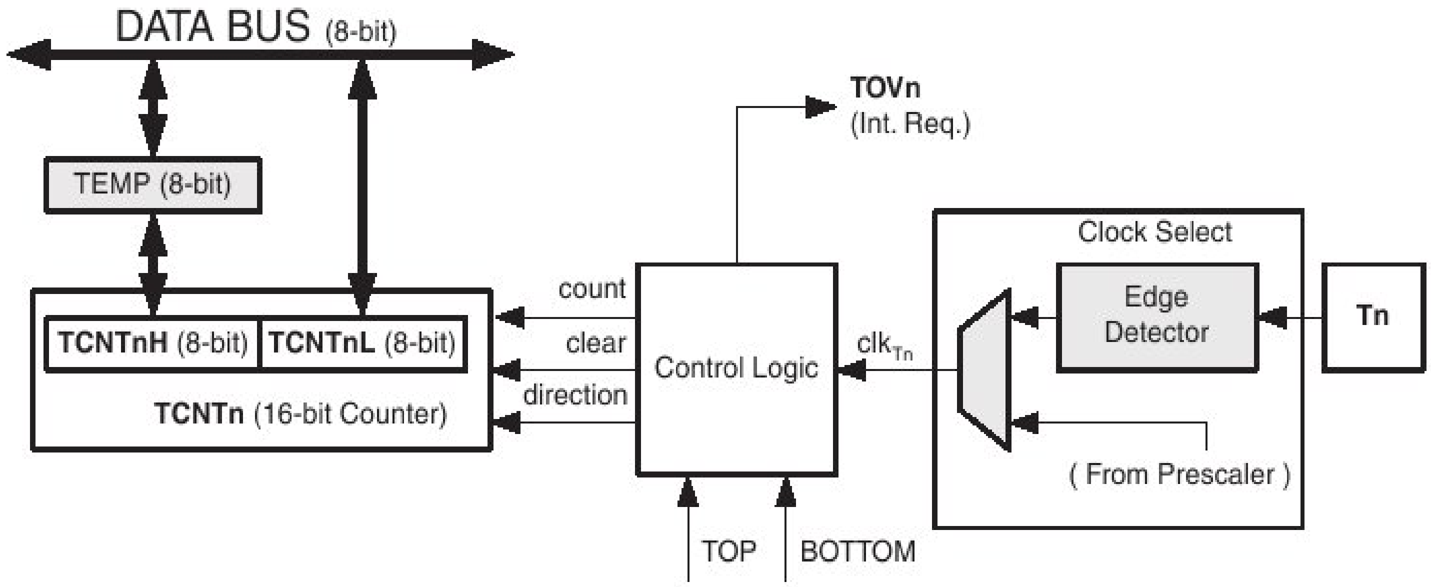
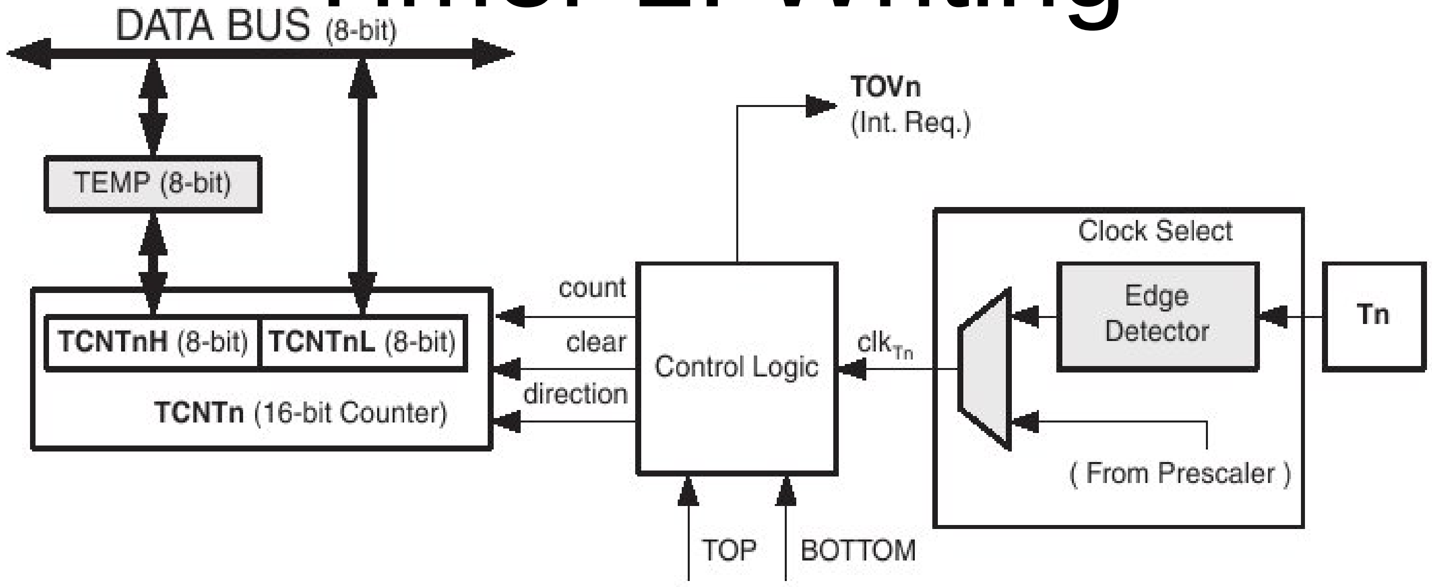


Figure from: Atmel mega 8 specification

Problem: 8 bit data bus, but 16 bit registers

- How to access the registers so as to avoid the shared data problem?

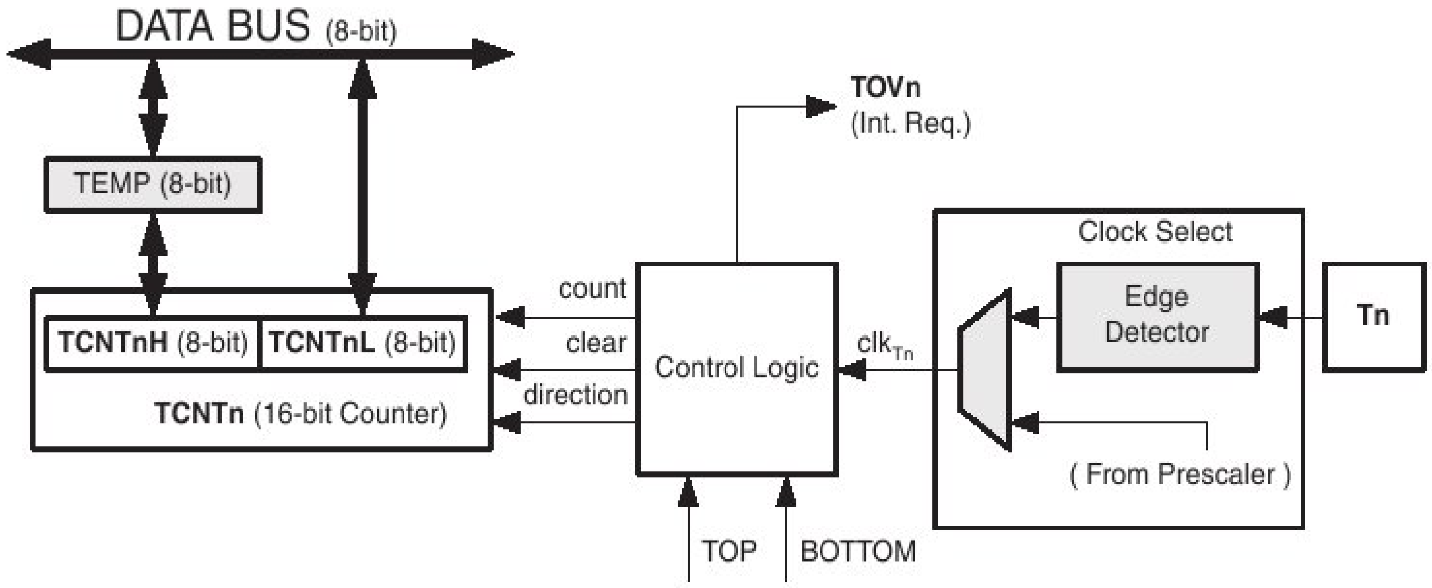
Timer 1: Writing



- Write to the high byte first (TCNTnH)
  - This stores the 8-bit value in a temporary register
- Write to low byte (TCNTnL)
  - What is on the data bus is written to the low byte
  - The temporary register is written to the high byte

(so both are changed simultaneously)

Timer 1: Reading



- Read from the low byte first (TCNTnL)
  - TCNTnH will also be written to the temporary register
- Read from high byte (TCNTnH)
  - This will actually pull the value from the temporary register

Timer 1 Access: The Good News

- OUlib provides functions to do this for you:

```
unsigned int timer1_read(void);
void timer1_set(unsigned int);
```

- The caveat:
  - OUlib is “thread safe”
  - Interrupts are disabled between access of the high and low registers (see implementations)

Input Capture Unit

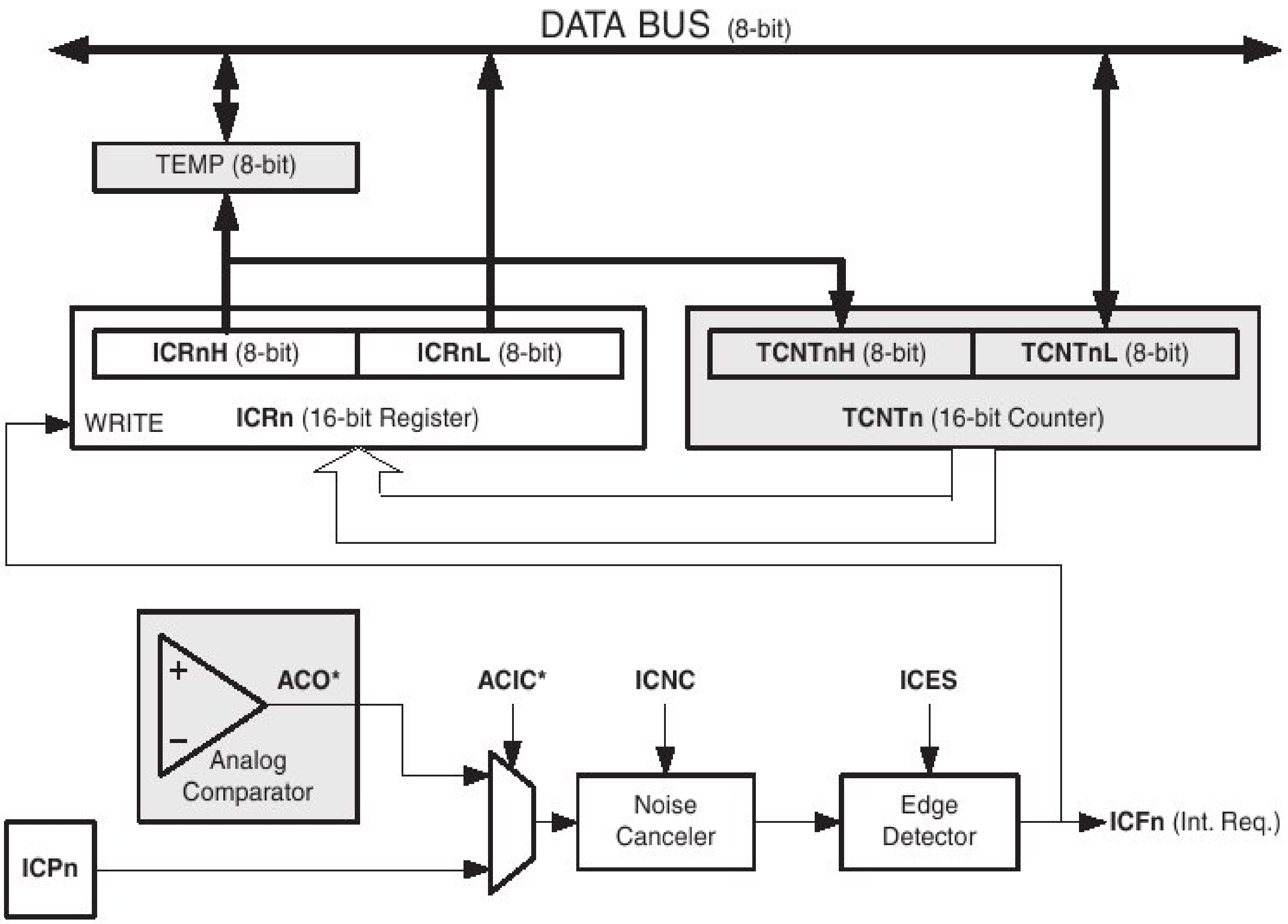


Figure from: Atmel mega 8 specification

Input Capture Unit

Captured value

- Access just as you would TCNTn[HL]

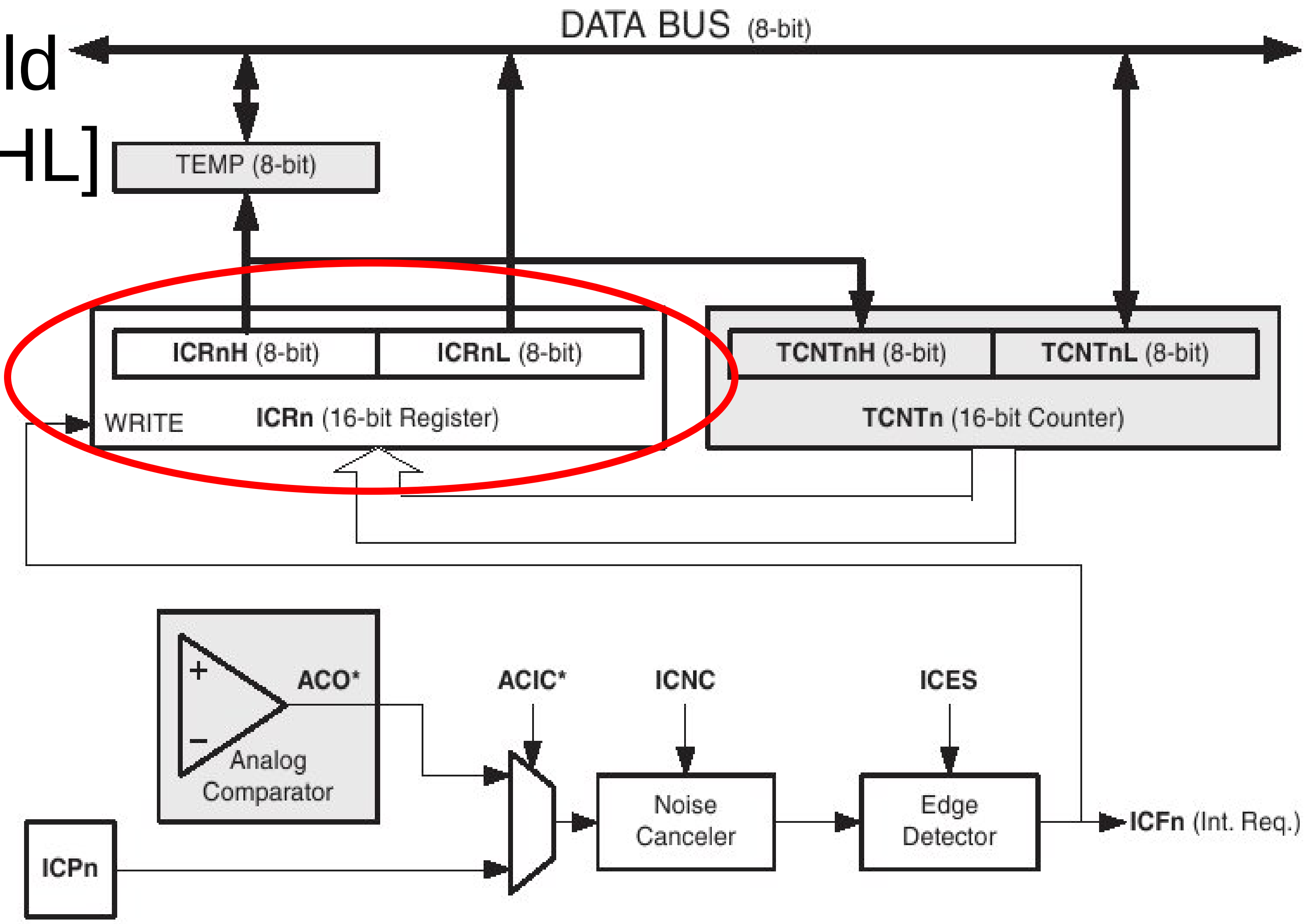


Figure from: Atmel mega 8 specification

Input Capture Unit

Copy on event

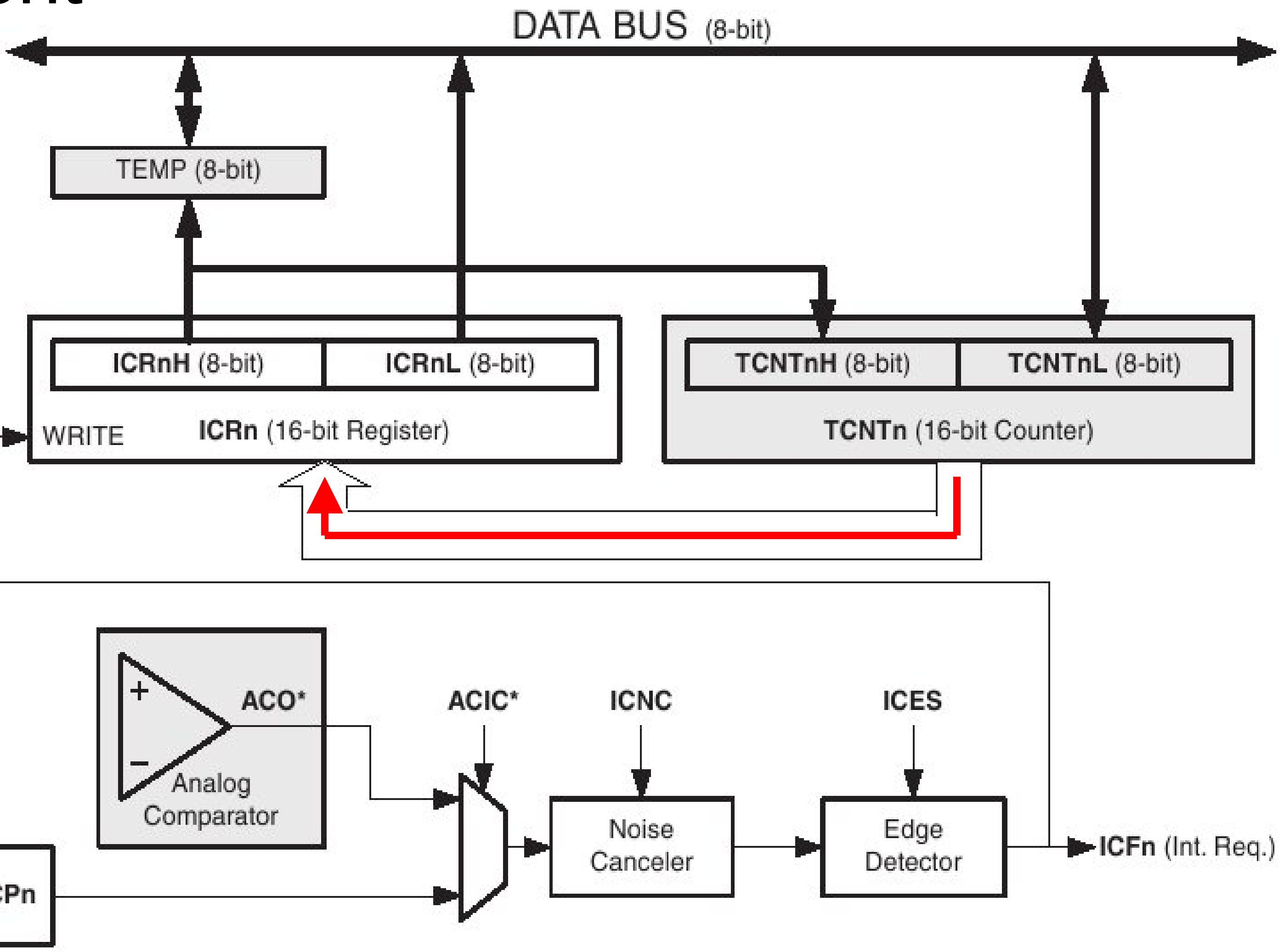


Figure from: Atmel mega 8 specification

Input Capture Unit

Event detector

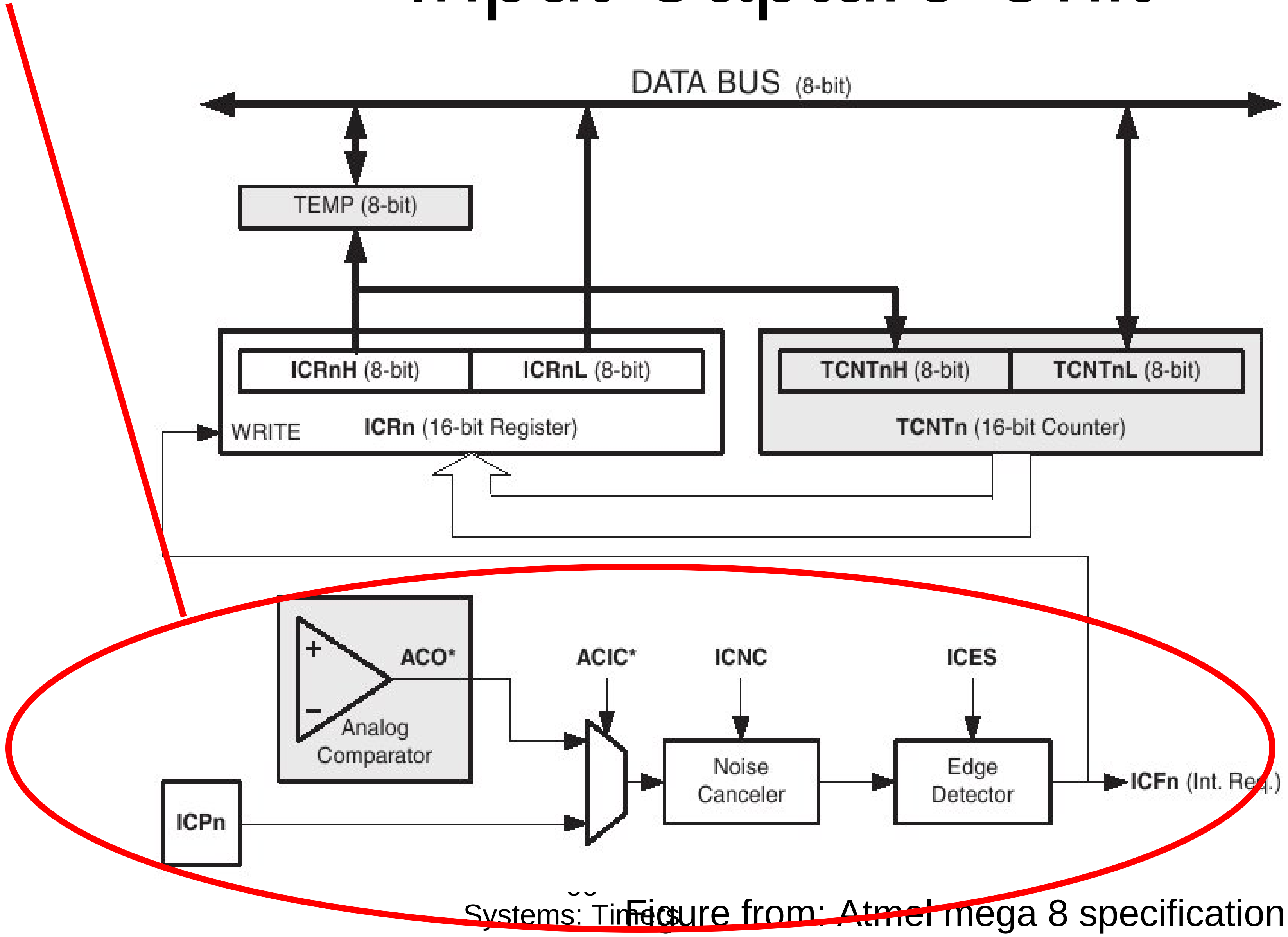


Figure from: Atmel mega 8 specification

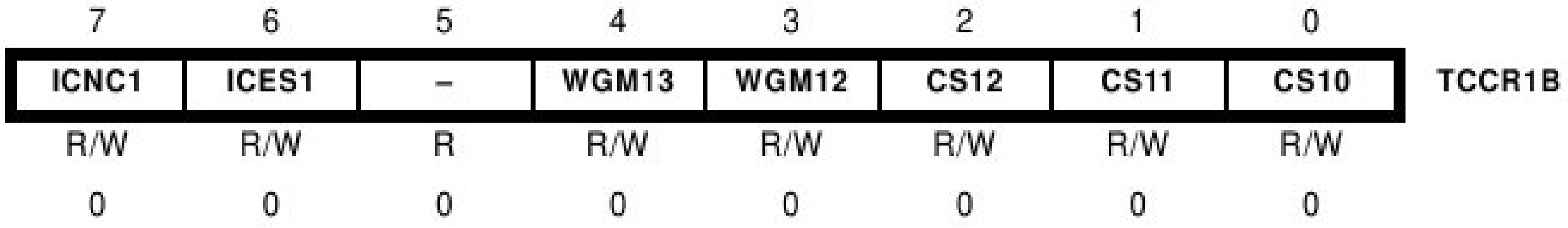
Input Capture Unit

No OUlib support right now…

Critical registers:

- ICRn[LH]: captured value
- TCCR1B: configuration
- ACSR: event source selection
- TIMSK: interrupt enable bit

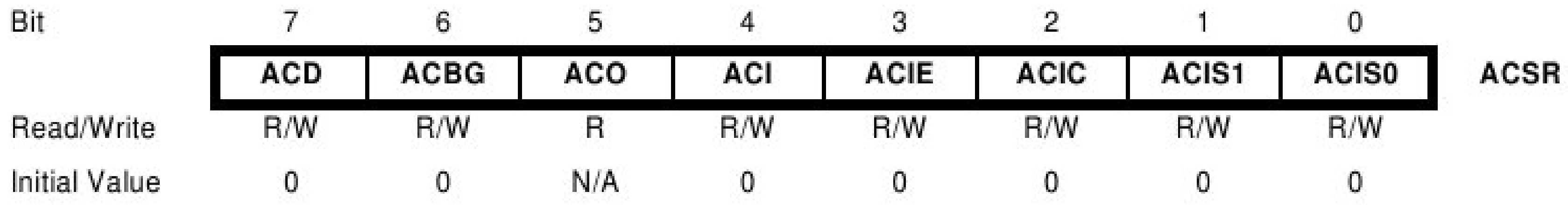
Input Capture Unit: TCCR1B



7	6	5	4	3	2	1	0	
ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

- ICNC1: Input compare noise canceller
  - Value = 1 -> canceling is turned on
  - Takes multiple samples of the pin state before detecting an event (this induces a small delay but gives a cleaner signal)
- ICES1: Input compare edge select
  - Value = 1 -> rising edge
  - Value = 0 -> falling edge

Input Capture Unit: ACSR

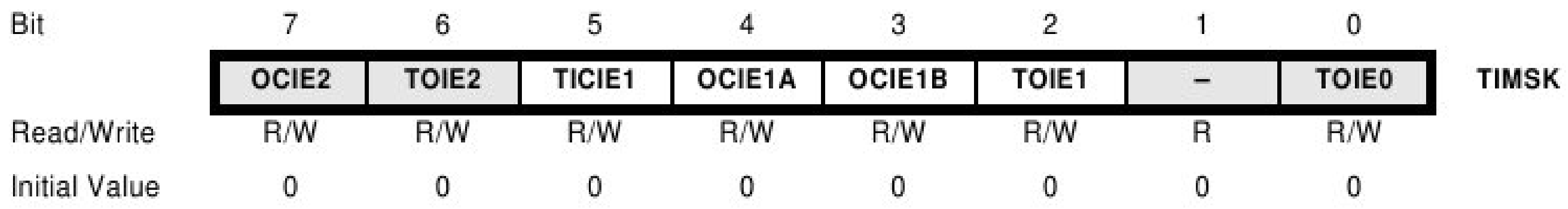


Bit	7	6	5	4	3	2	1	0	
	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	0	0	0	0	0	

ACIC: External event source

- Value = 1 -> Analog comparator
- Value = 0 -> ICPn pin

Input Capture Unit: TIMSK

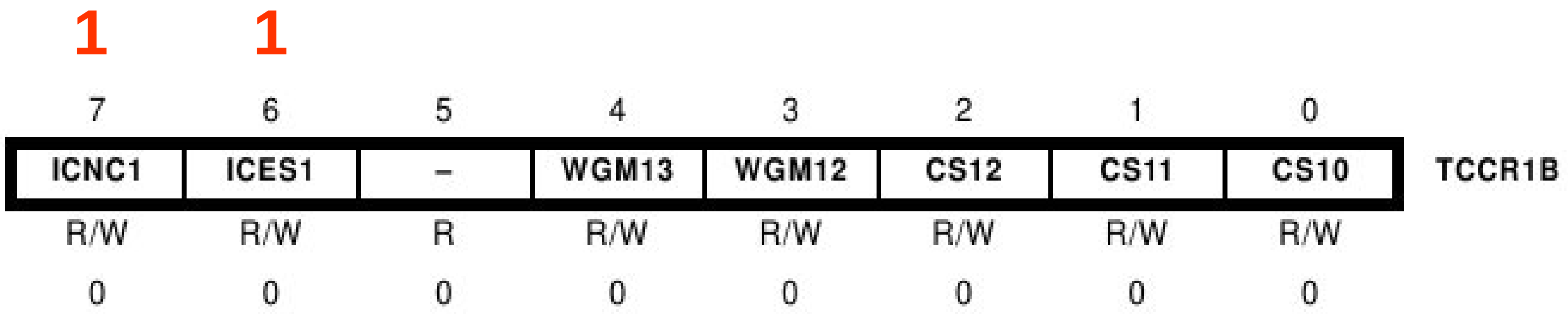


Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- TICIE1: Input capture interrupt enable
  - Value = 1 -> enabled

Some Example Code

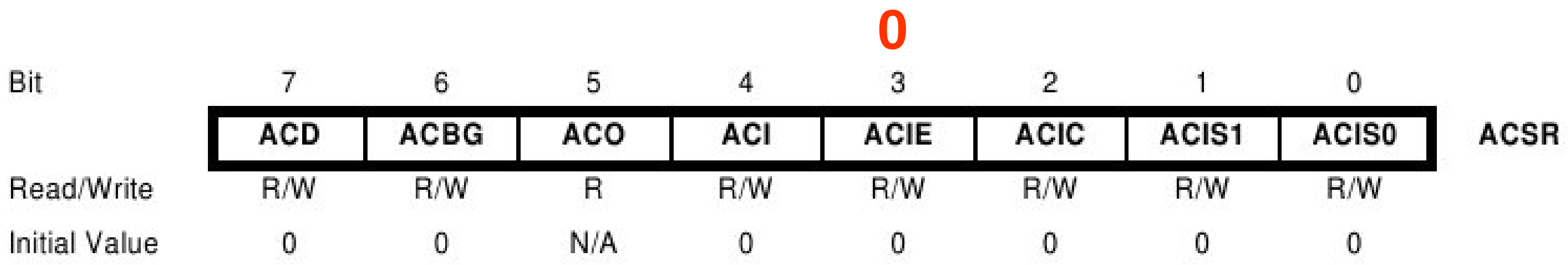
```
// Turn on noise canceling; detect
  rising edge
TCCR1B |= _BV(ICNC1) | _BV(ICES1);
```



1	1							
7	6	5	4	3	2	1	0	
ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

Some Example Code

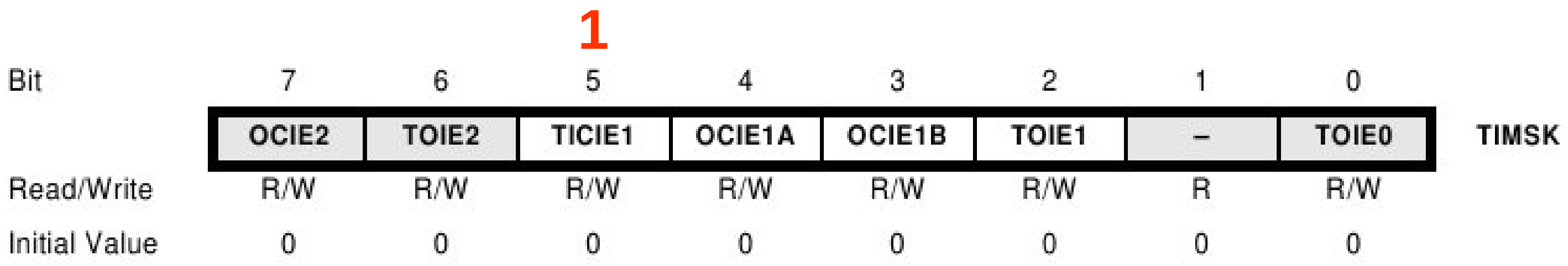
```
// Turn on noise canceling; detect
  rising edge
TCCR1B |= _BV(ICNC1) | _BV(ICES1);
// Use pin as input (not analog comp)
ACSR &= ~_BV(ACIE);
```



					0				
Bit	7	6	5	4	3	2	1	0	
	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	0	0	0	0	0	

Some Example Code

```
// Turn on noise canceling; detect
  rising edge
TCCR1B |= _BV(ICNC1) | _BV(ICES1);
// Use pin as input (not analog comp)
ACSR &= ~_BV(ACIE);
// Enable interrupt
TIMSK |= _BV(TICIE1);
```



			1						
Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Some Example Code

```
// Turn on noise canceling; detect
  rising edge
TCCR1B |= _BV(ICNC1) | _BV(ICES1);
// Use pin as input (not analog comp)
ACSR &= ~_BV(ACIE);
// Enable interrupt
TIMSK |= _BV(TICIE1);
// Enable global interrupts
sei();
```

Interrupt Service Routine

```
ISR(TIMER1_CAPT_vec)
{
   // Do something …
}
```

- Read ICRn[LH] as soon as possible (it could be overwritten by the next event)
- You can change the configuration of the input capture unit (e.g. to alternate between falling and rising edges)

Output Compare Mode

General idea:

- Counter moves through some sequence of values
- At some specified counter value(s), the processor produces an event
  - Generate an interrupt
  - Change the state of the output pin

Many Different Output Compare Modes

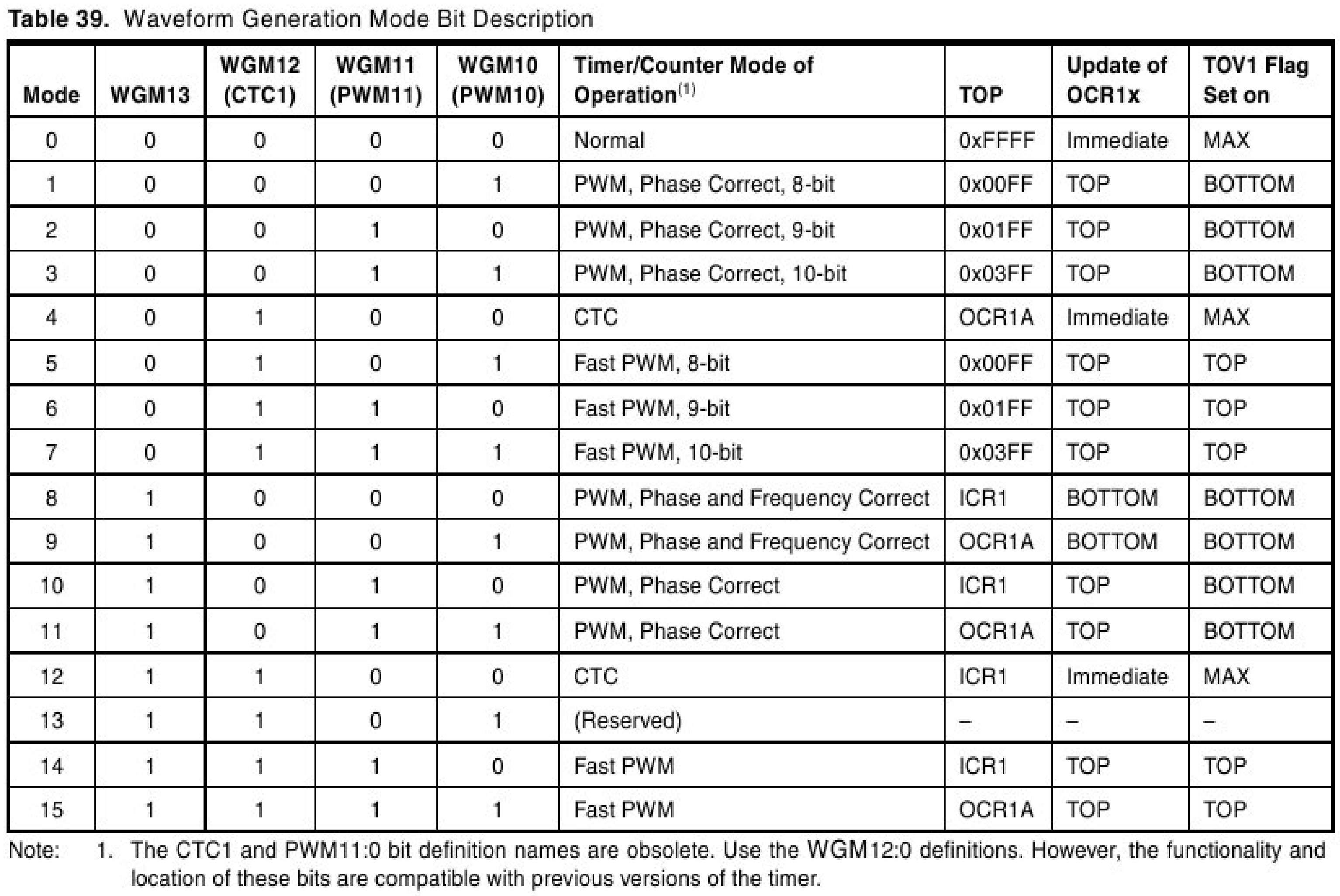


Table 39. Waveform Generation Mode Bit Description

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation(1)	TOP	Update of OCR1x	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

We Will Focus on Fast PWM

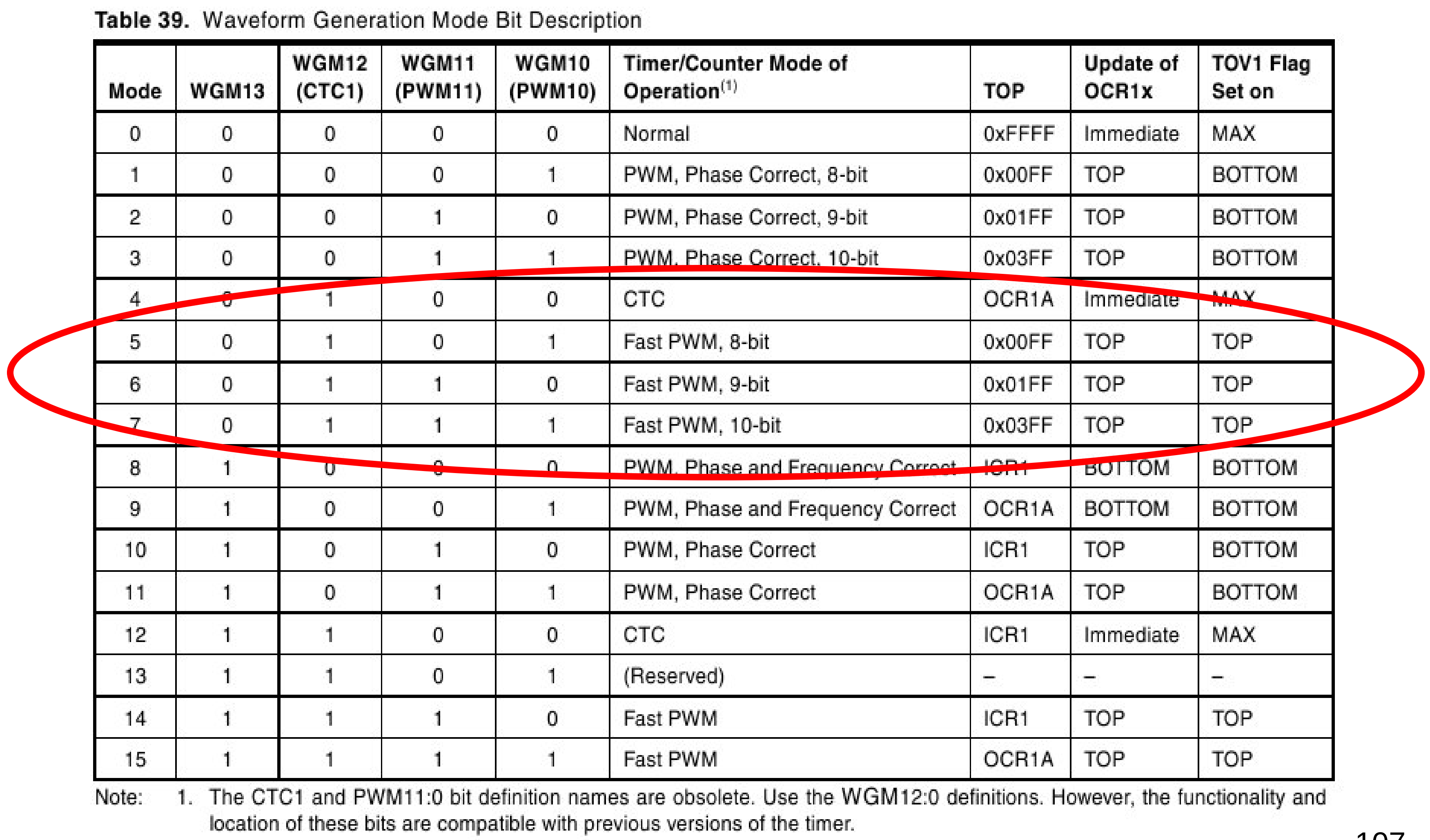


Table 39. Waveform Generation Mode Bit Description

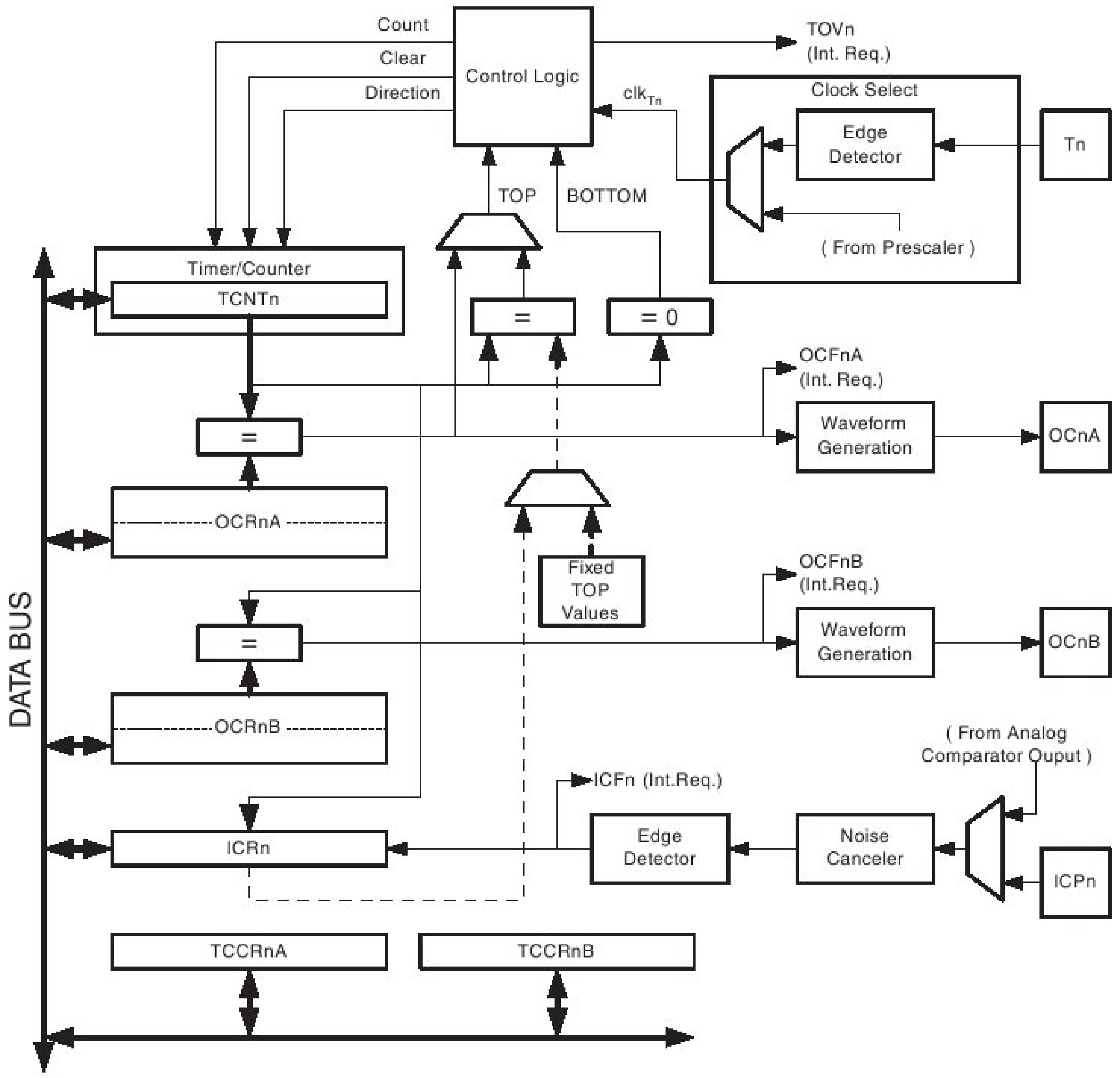
Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation[1]	TOP	Update of OCR1x	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

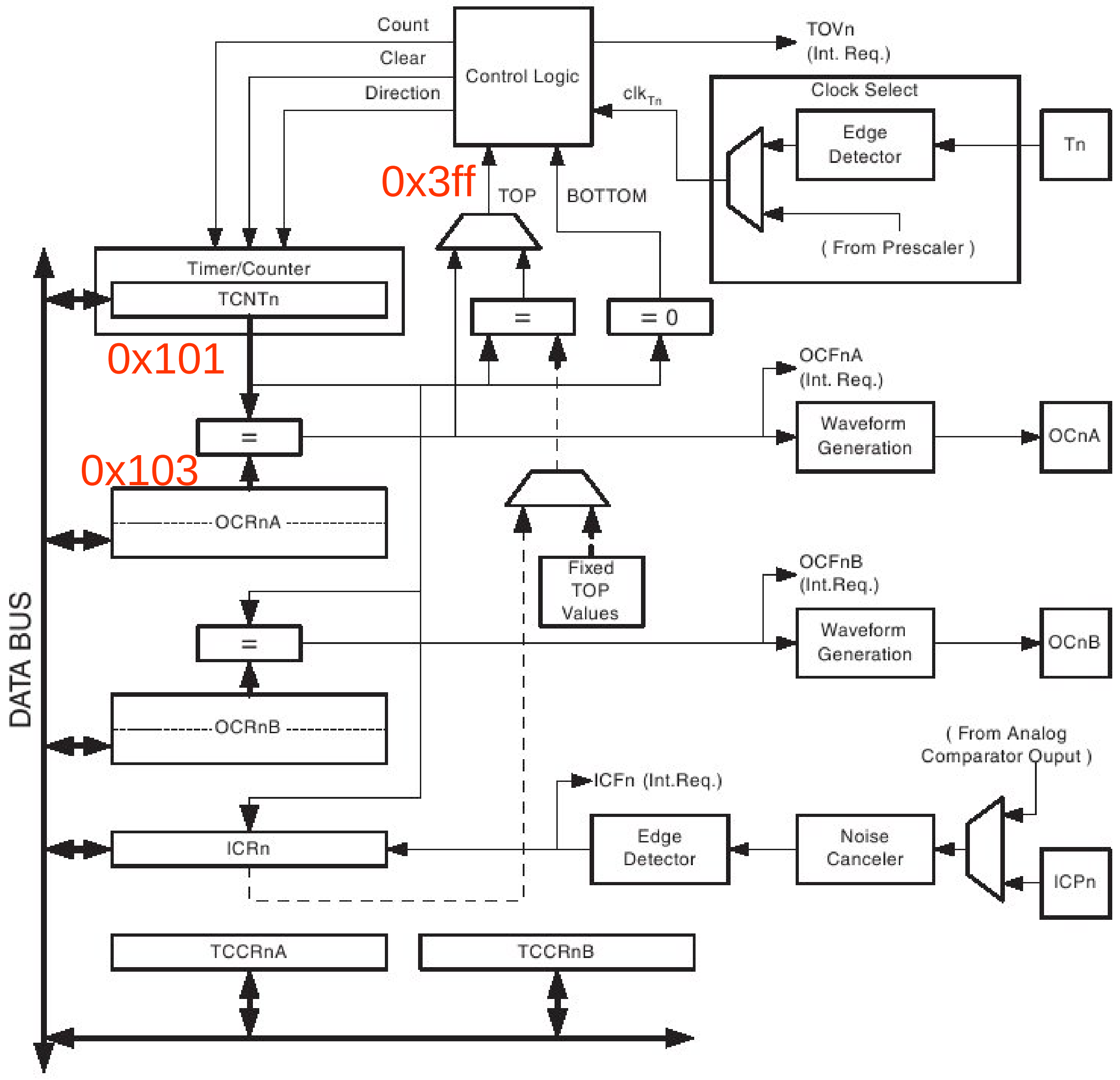
Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

Output Compare Mode: Fast PWM

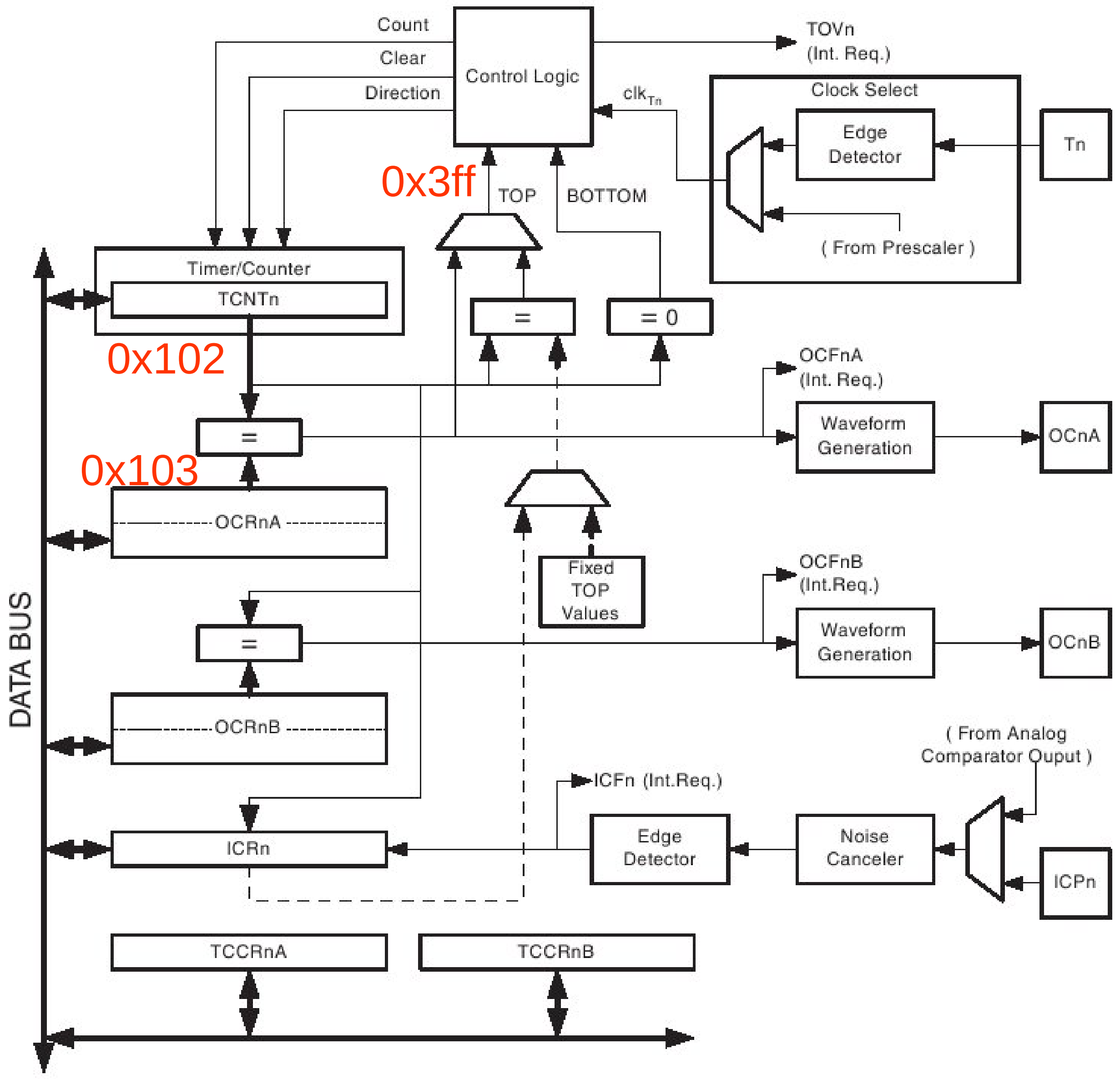
Generating a pulse width modulated signal:

- Counter increments from BOTTOM (0) to TOP (configurable). Once TOP is reached:
  - Set the state of an output pin (e.g., set to 1)
  - Roll over to BOTTOM
- When the counter reaches a specific intermediate value:
  - Change the state of the output pin (e.g. to 0)

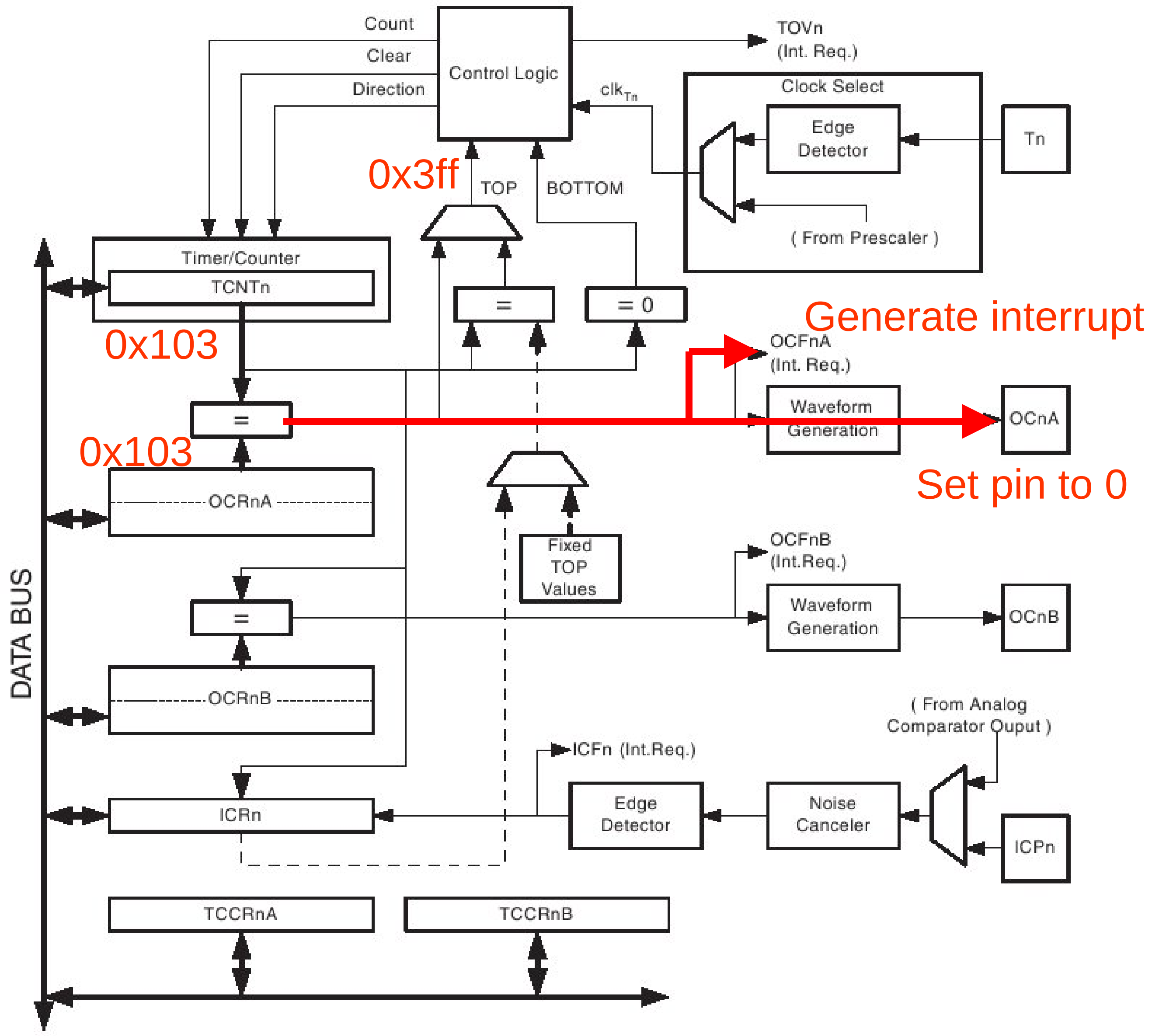

Count
Clear
Direction
Control Logic
TOVn
(Int. Req.)
clk$_{Tn}$
Clock Select
Edge Detector
Tn
(From Prescaler)
TOP
BOTTOM
DATA BUS
Timer/Counter
TCNTn
=
= 0
OCFnA
(Int. Req.)
Waveform Generation
OCnA
OCRnA
Fixed TOP Values
OCFnB
(Int.Req.)
Waveform Generation
OCnB
OCRnB
(From Analog Comparator Ouput)
ICFn (Int.Req.)
Edge Detector
Noise Canceler
ICPn
ICRn
TCCRnA
TCCRnB

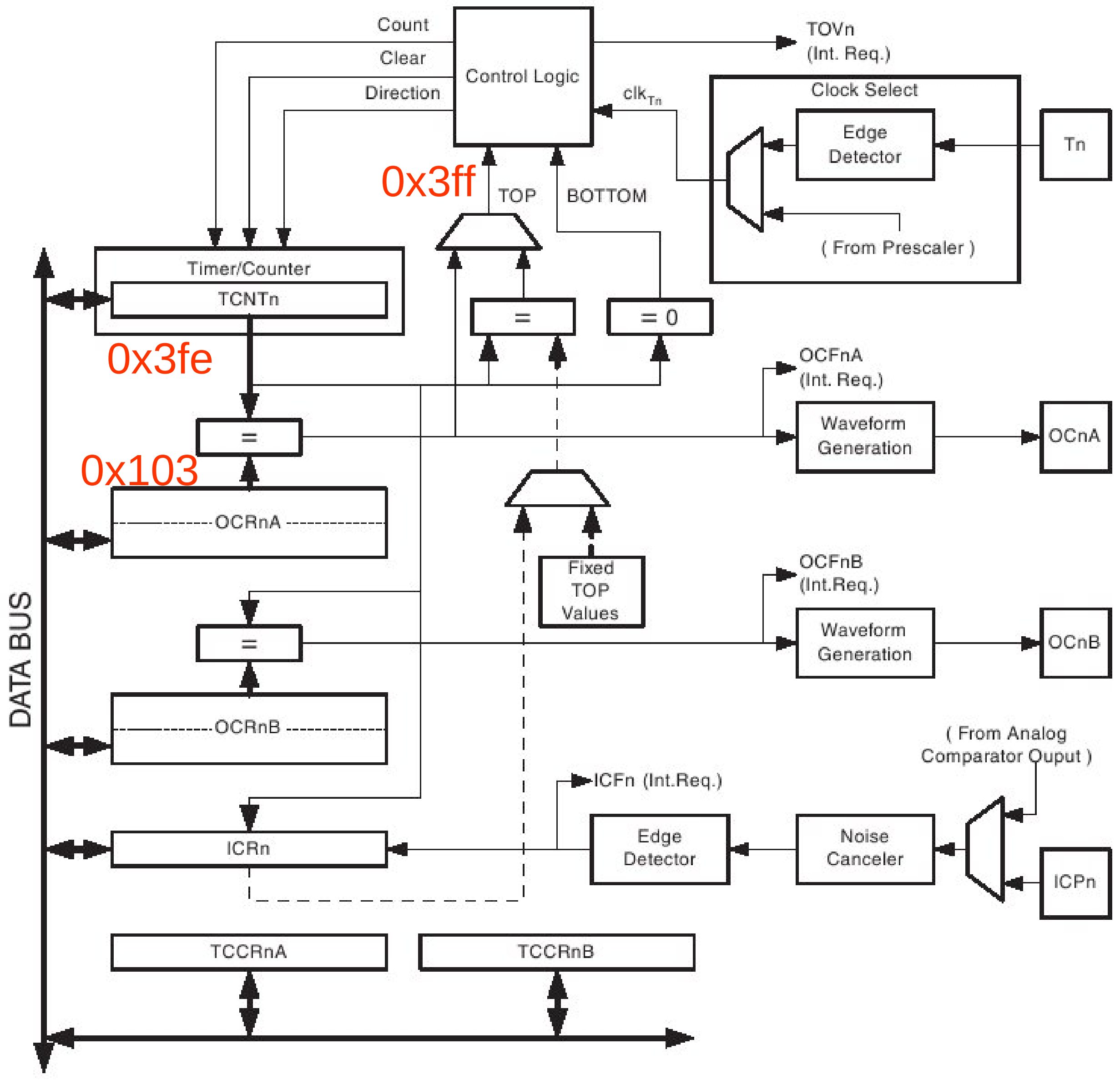


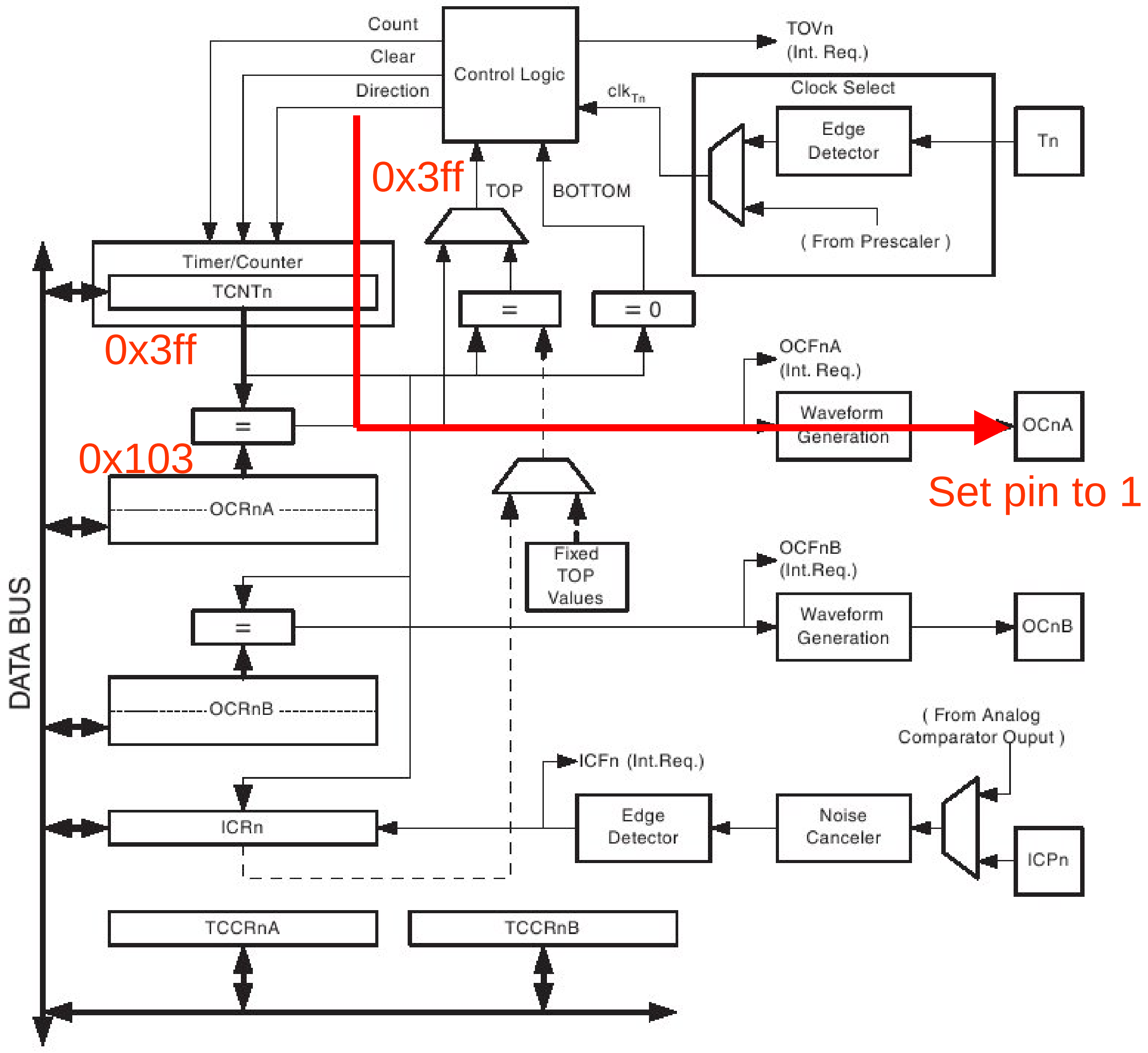
Count
Clear
Direction
Control Logic
TOVn (Int. Req.)
clk$_{Tn}$
Clock Select
Edge Detector
Tn
(From Prescaler)
0x3ff
TOP
BOTTOM
Timer/Counter
TCNTn
=
= 0
0x101
OCFnA (Int. Req.)
Waveform Generation
OCnA
=
0x103
OCRnA
Fixed TOP Values
OCFnB (Int.Req.)
Waveform Generation
OCnB
DATA BUS
=
OCRnB
(From Analog Comparator Ouput)
ICFn (Int.Req.)
Edge Detector
Noise Canceler
ICPn
ICRn
TCCRnA
TCCRnB



Count
Clear
Direction
Control Logic
TOVn (Int. Req.)
clk$_{Tn}$
Clock Select
Edge Detector
Tn
(From Prescaler)
0x3ff
TOP
BOTTOM
Timer/Counter
TCNTn
=
= 0
0x102
OCFnA (Int. Req.)
Waveform Generation
OCnA
0x103
OCRnA
Fixed TOP Values
OCFnB (Int.Req.)
Waveform Generation
OCnB
DATA BUS
OCRnB
(From Analog Comparator Ouput)
ICFn (Int.Req.)
Edge Detector
Noise Canceler
ICPn
ICRn
TCCRnA
TCCRnB


Count
Clear
Direction
Control Logic
TOVn
(Int. Req.)
clkTn
Clock Select
Edge Detector
Tn
(From Prescaler)
0x3ff
TOP
BOTTOM
Timer/Counter
TCNTn
=
= 0
Generate interrupt
OCFnA
(Int. Req.)
0x103
=
Waveform Generation
OCnA
0x103
OCRnA
Set pin to 0
Fixed TOP Values
OCFnB
(Int.Req.)
=
Waveform Generation
OCnB
DATA BUS
OCRnB
(From Analog Comparator Ouput)
ICFn (Int.Req.)
Edge Detector
Noise Canceler
ICRn
ICPn
TCCRnA
TCCRnB


Count
Clear
Direction
Control Logic
TOVn (Int. Req.)
clkTn
Clock Select
Edge Detector
Tn
(From Prescaler)
0x3ff
TOP
BOTTOM
Timer/Counter
TCNTn
=
= 0
0x3fe
OCFnA (Int. Req.)
Waveform Generation
OCnA
=
0x103
OCRnA
Fixed TOP Values
OCFnB (Int.Req.)
Waveform Generation
OCnB
=
OCRnB
(From Analog Comparator Ouput)
ICFn (Int.Req.)
Edge Detector
Noise Canceler
ICPn
ICRn
TCCRnA
TCCRnB
DATA BUS



Count
Clear
Direction
Control Logic
TOVn (Int. Req.)
clk$_{Tn}$
Clock Select
Edge Detector
Tn
(From Prescaler)
TOP
BOTTOM
0x3ff
Timer/Counter
TCNTn
0x3ff
=
= 0
OCFnA (Int. Req.)
Waveform Generation
OCnA
=
0x103
OCRnA
Set pin to 1
Fixed TOP Values
OCFnB (Int.Req.)
=
Waveform Generation
OCnB
OCRnB
DATA BUS
(From Analog Comparator Ouput)
ICFn (Int.Req.)
Edge Detector
Noise Canceler
ICPn
ICRn
TCCRnA
TCCRnB

PWM and Interrupt Frequency

$$pwm\ \ freq = \frac{clock\ \ freq}{prescalar * (1 + TOP)}$$

Example:

$$pwm\ \ freq = \frac{16{,}000{,}000}{1024 * (1 + 0x3ff)}$$

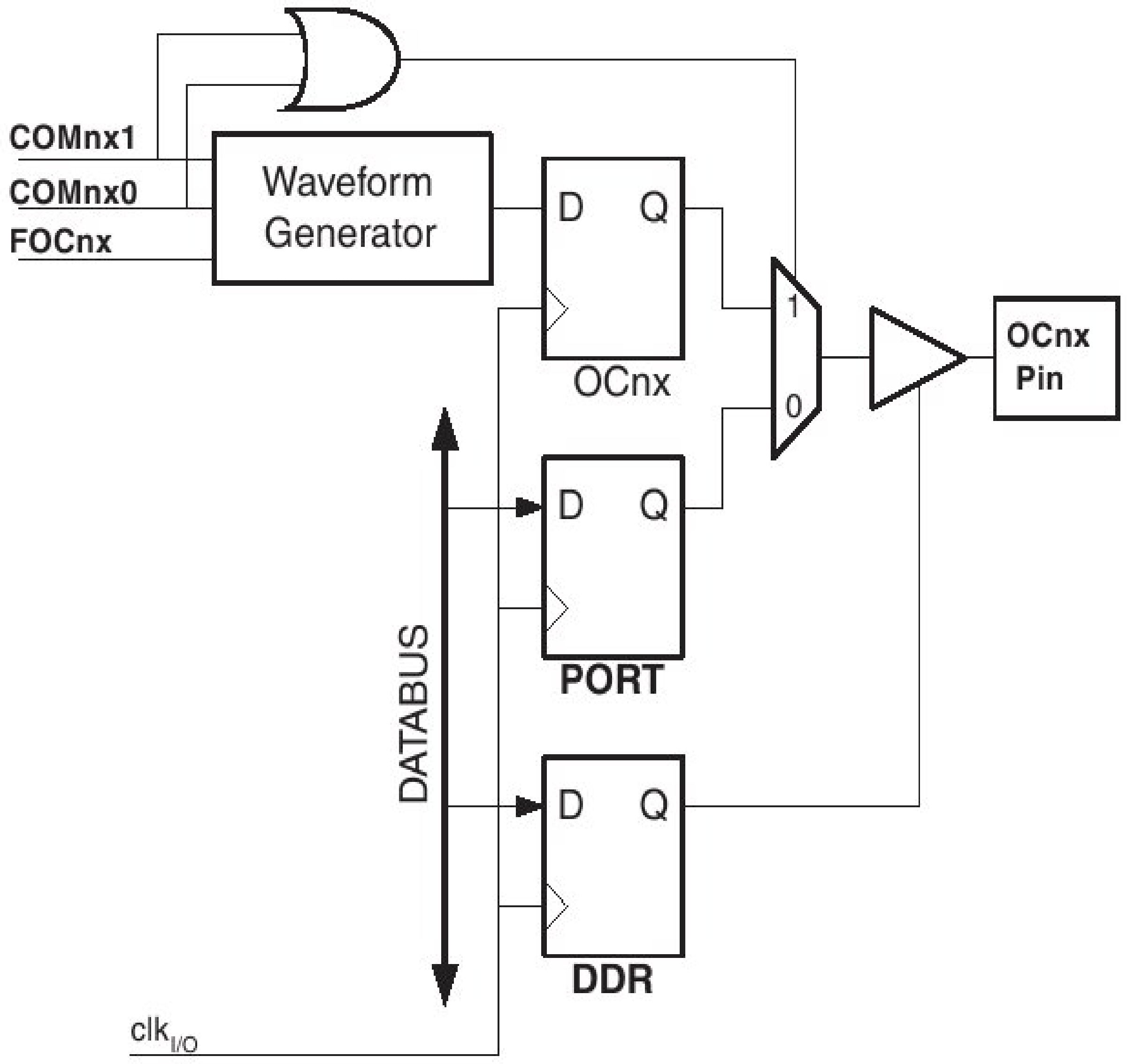
$$= 15.2588\,Hz$$

This gives us 10 bits of pulse width resolution

Pin Driver Circuit

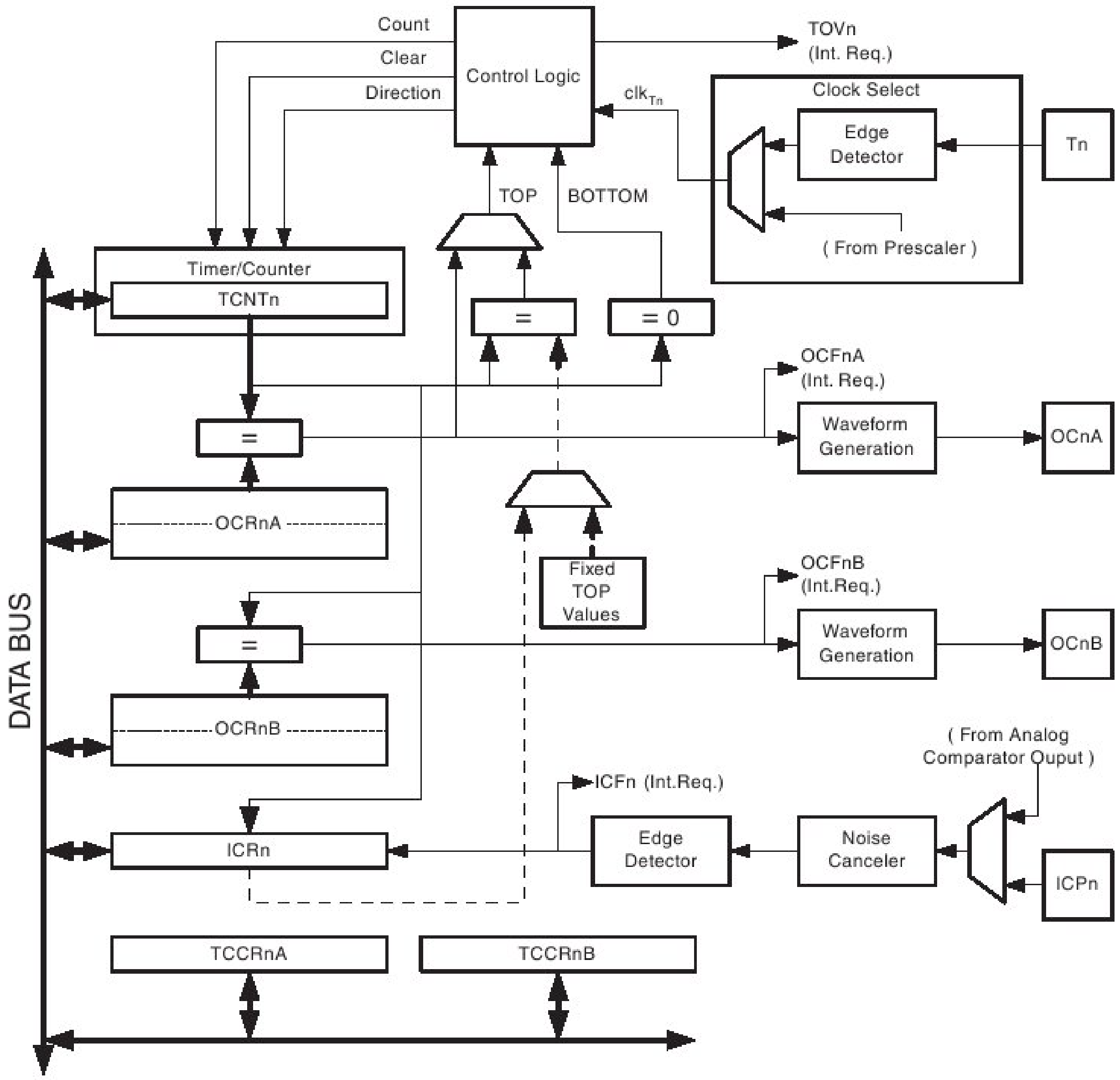
- Use of this waveform generator overrides PORTx

Figure 36. Compare Match Output Unit, Schematic



OCRnA is double-buffered

- The real OCRnA as shown is updated when the counter rolls over
- Eliminates problems with updates in the middle of your pulse



Configuration

- Prescalar
- Waveform Generation Mode (in our case, Fast PWM, 10 bit)
- Polarity of the output bit (Output Mode)
- Interrupt enable (if desired)
- Initial pulse width

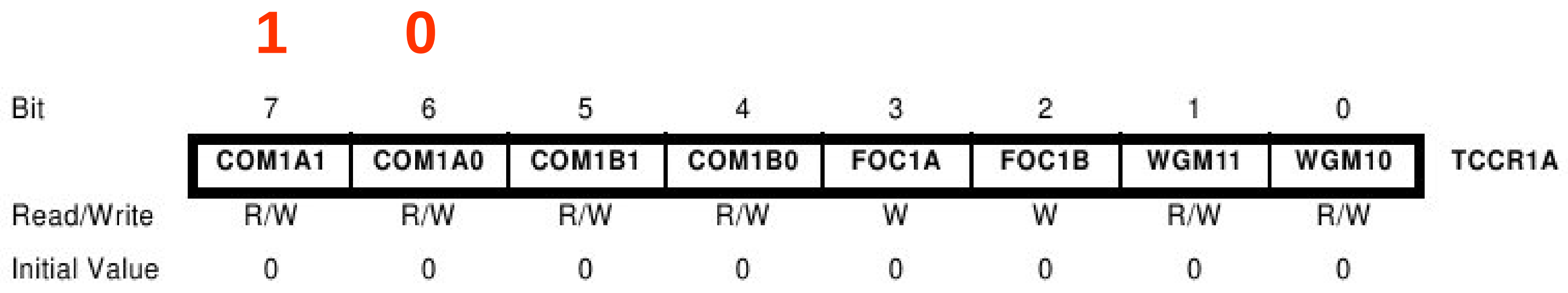
Configuration

```
// Configure PWM for output compare pin A
// Prescaler
timer1_config(TIMER1_PRE_1024);
```

Prescaler configuration is the same as with timer0

Configuration

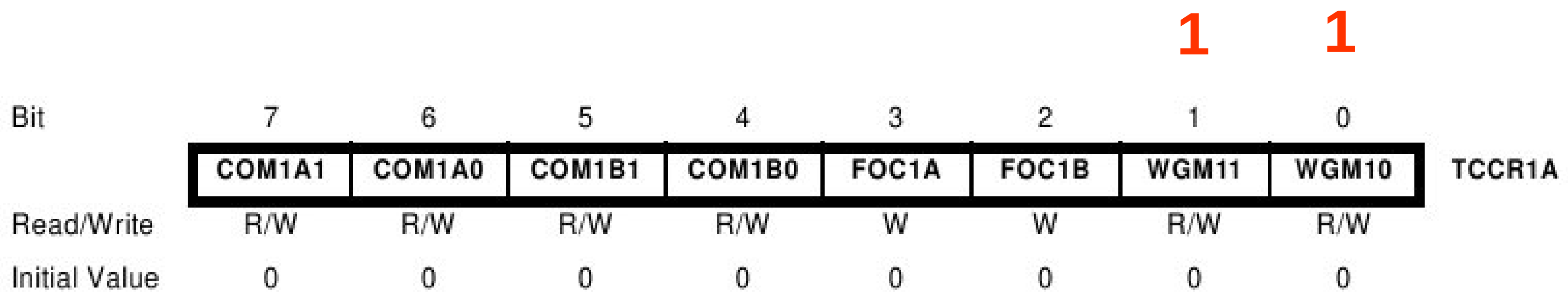
```
// Configure PWM for output compare pin A
// Prescaler
timer1_config(TIMER1_PRE_1024);

// Output Mode for channel A: output is low after compare match
// COM1A[10] = 10
TCCR1A = TCCR1A & ~_BV(COM1A0) | _BV(COM1A1);
```



	1	0							
Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Configuration

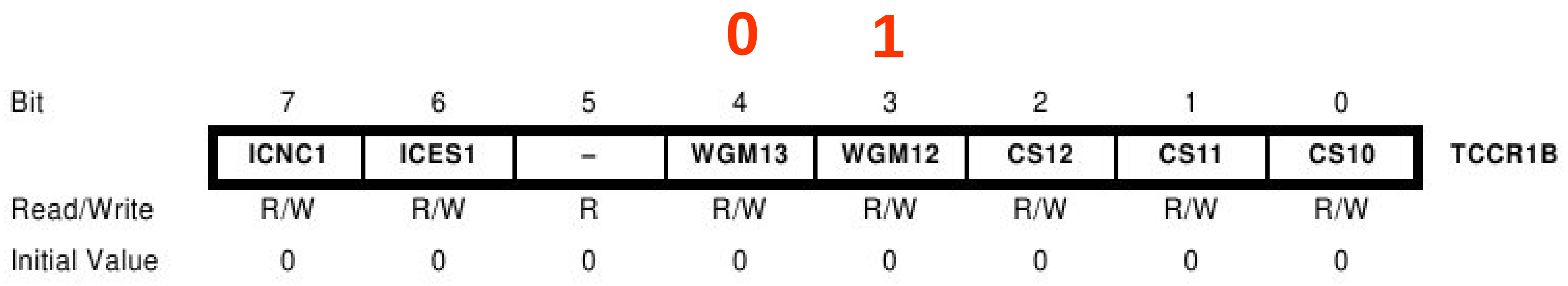
```
// Configure PWM for output compare pin A
// Prescaler
timer1_config(TIMER1_PRE_1024);

// Output Mode for channel A: output is low after compare match
// COM1A[10] = 10
TCCR1A = TCCR1A & ~_BV(COM1A0) | _BV(COM1A1);

// WGM1[3210] = 01 11.  Fast PWM, 10-bit
TCCR1A = TCCR1A | _BV(WGM11) | _BV(WGM10);
```



							1	1	
Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Configuration

```
// Configure PWM for output compare pin A
// Prescaler
timer1_config(TIMER1_PRE_1024);

// Output Mode for channel A: output is low after compare match
// COM1A[10] = 10
TCCR1A = TCCR1A & ~_BV(COM1A0) | _BV(COM1A1);

// WGM1[3210] = 01 11.  Fast PWM, 10-bit
TCCR1A = TCCR1A | _BV(WGM11) | _BV(WGM10);

TCCR1B = TCCR1B & ~_BV(WGM13) | _BV(WGM12);
```

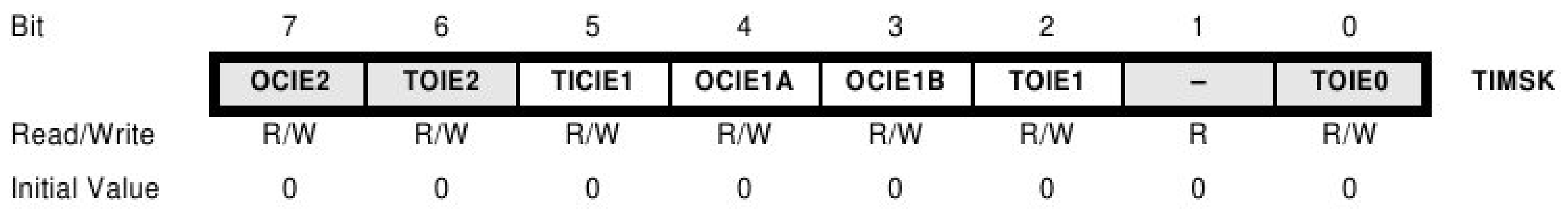


				0	1				
Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Configuration

```
// Configure PWM for output compare pin A
// Prescaler
timer1_config(TIMER1_PRE_1024);

// Output Mode for channel A: output is low after compare match
// COM1A[10] = 10
TCCR1A = TCCR1A & ~_BV(COM1A0) | _BV(COM1A1);

// WGM1[3210] = 01 11.  Fast PWM, 10-bit
TCCR1A = TCCR1A | _BV(WGM11) | _BV(WGM10);

TCCR1B = TCCR1A & ~(_BV(WGM13)) | _BV(WGM12);

// Enable interrupt
TIMSK |= _BV(OCIE1A);
```

1



Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Configuration

```
// Configure PWM for output compare pin A
// Prescaler
timer1_config(TIMER1_PRE_1024);

// Output Mode for channel A: output is low after compare match
// COM1A[10] = 10
TCCR1A = TCCR1A & ~(_BV(COM1A1) | _BV(COM1A0));

// WGM1[3210] = 01 11.  Fast PWM, 10-bit
TCCR1A = TCCR1A | _BV(WGM11) | _BV(WGM10);

TCCR1B = TCCR1A & ~(_BV(WGM13)) | _BV(WGM12);

// Enable interrupt
TIMSK |= _BV(OCIE1A);

// Enable global interrupts
sei();
```

Use of PWM Generator

Change the pulse width at any time

- This change will take effect at the beginning of the next pulse
- Must deal with the synchronous update of the high and low byte of OCR1A

Continuously Varying Pulse Width

```
while(1);
 {
   // Loop over entire range
   for(val=0; val<0x400; ++val) {
     // Write high byte first (goes to temporary register)
     OCR1AH = (uint8_t) (val >> 8);

     // Write low byte second (causes both to be written
    // simultaneously)
     OCR1AL = (uint8_t) (val & 0xff);

     // Sleep
     delay_ms(1);
   };
 };
```

Temporary Register

- Registers such as `OCR1AH` are all mapped to the same temporary register
- You must ensure that between the writes to `OCR1AH` and `OCR1AL` that no other code is executed that manipulate the temporary register
- This can come up if your ISR is also modifying these registers

Timer 2

- 8-bit counter
- Output-compare
- Waveform generator
  - So: can also generate PWM signals