Last Time

Project 2 discussion

- Circuits
- Low-level functions

Today

Timing:

- Generating precisely-timed outputs
- Measuring the time that an event occurs

Timing of Events

Suppose that we want produce a pulse on a digital line that was exactly 500 ms in length?

- What would the code look like?

Timing of Events

```
// Assume it is pin 0 of port B

PORTB = PORTB | 1;
delay_ms(500);
PORTB = PORTB & ~1;
```

Timing of Events

```
// Assume it is pin 0 of port B

PORTB = PORTB | 1;
delay_ms(500);
PORTB = PORTB & ~1;
```

This will work, but why is it undesirable?

Timing of Events

This will work, but why is it undesirable?

`delay_ms()` is implemented by using a for() loop

- The microcontroller can’t do anything else while it is looping
- Have to loop a precise number of times (not always easy to do)

Timing of Events: Another Example

Suppose we would want to measure the width of a pulse. How would we implement this?

Timing of Events: Another Example

How would we implement this?

```
// Wait for pin to go high
while(PINB & 0x1 == 0){};

// Now count until it goes low
for(counter = 0; PINB & 0x1; ++counter)
{
  delay_ms(1);
}
// Now: counter is the width of
//   of the pulse in ms
```

Timing of Events: Another Example

Again: the program cannot be doing anything else while it is waiting

Counter/Timers in the Mega8

The mega8 incorporates three counter/timer devices in hardware.

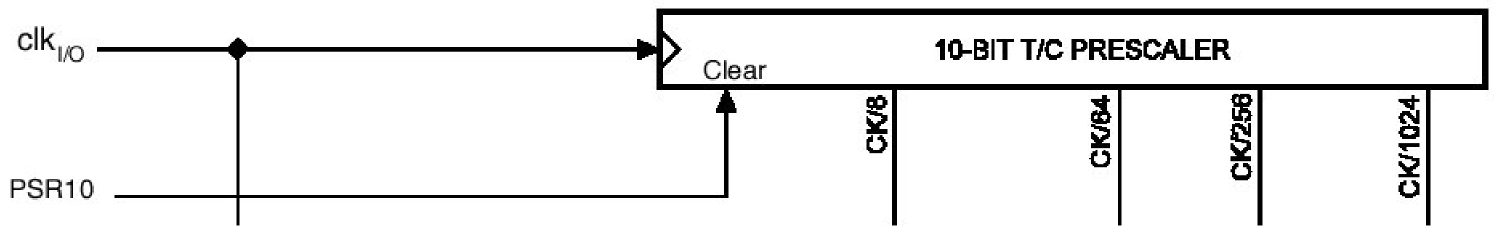
These can:

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

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 (an event) can be generated (more on this next time)

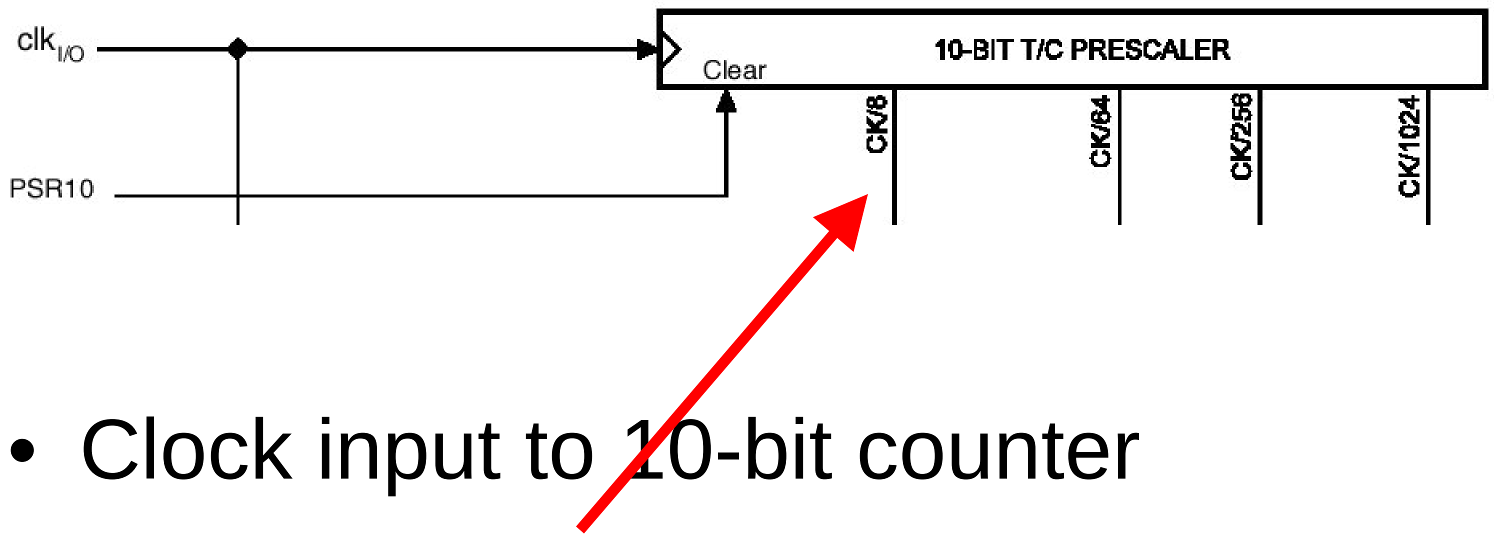
Timer 0 Implementation



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

    (counting from 1)

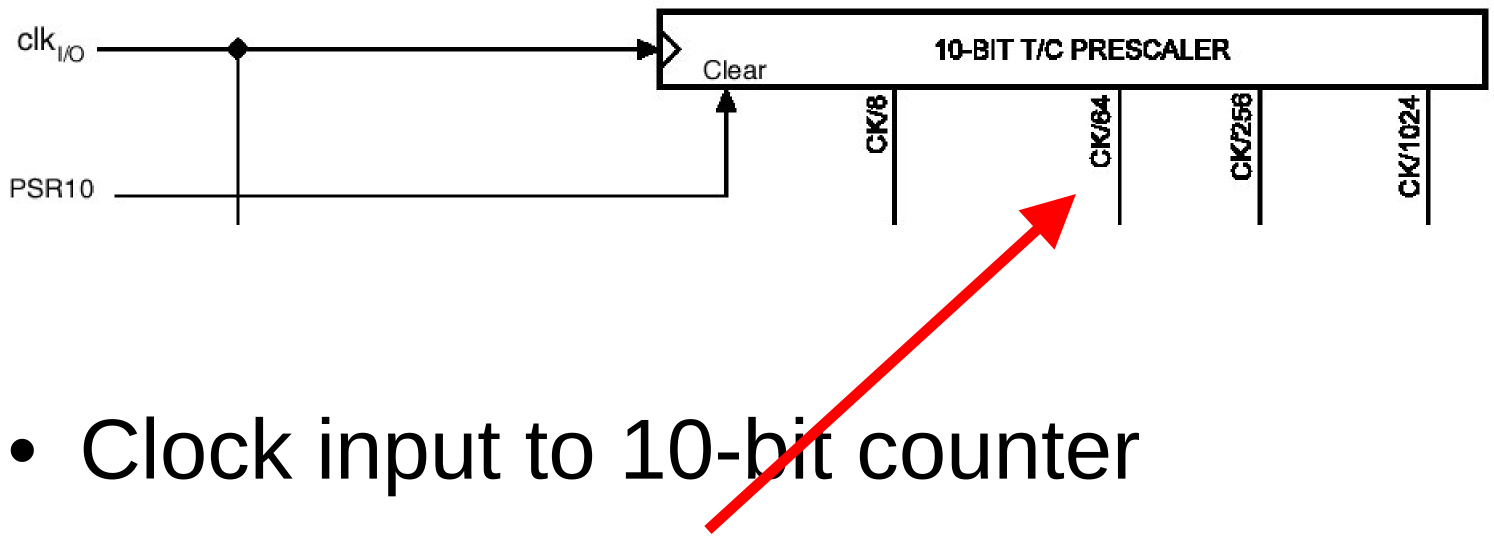
Timer 0 Implementation



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

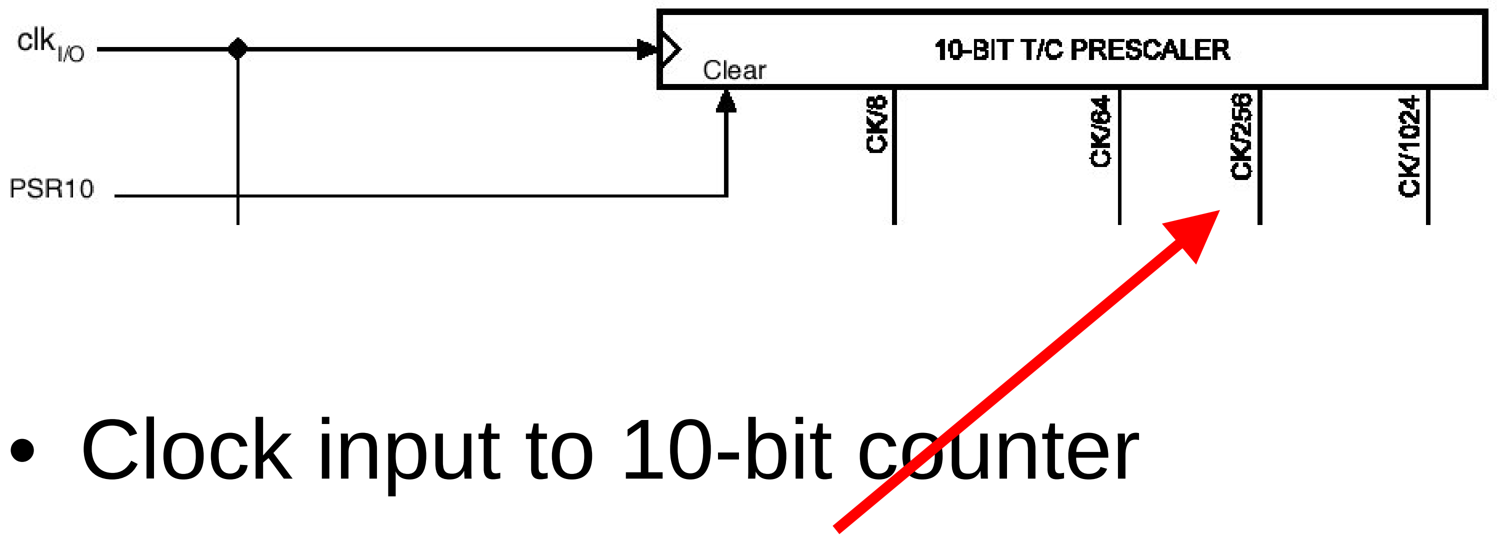
  (counting from 1)

Timer 0 Implementation



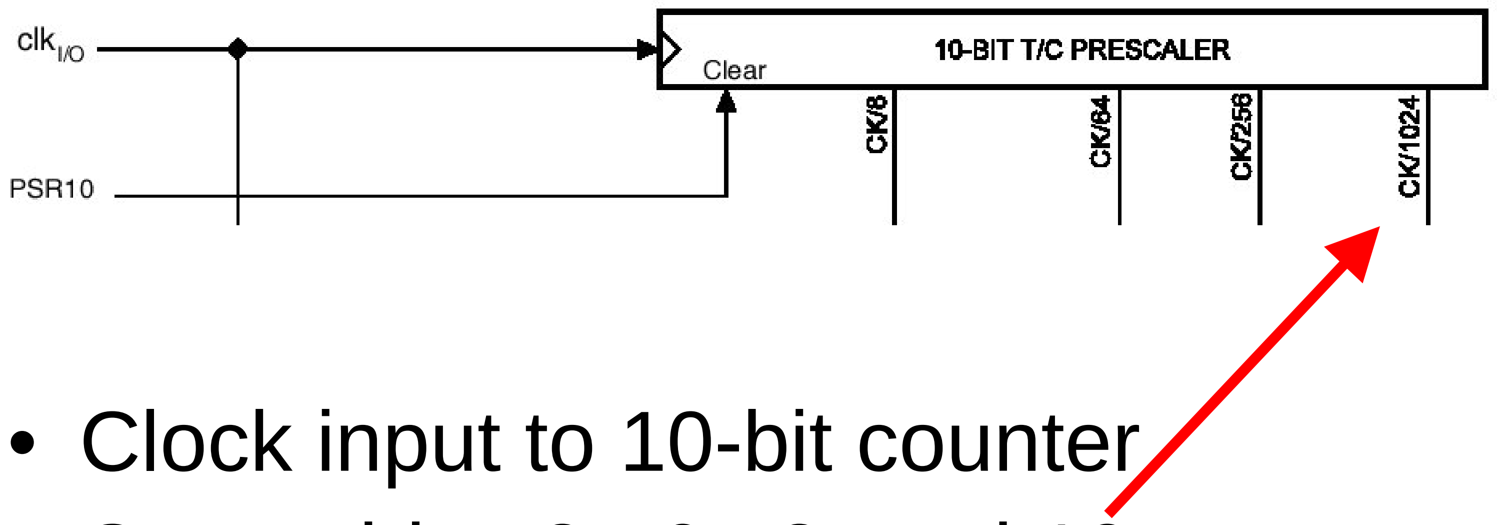
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



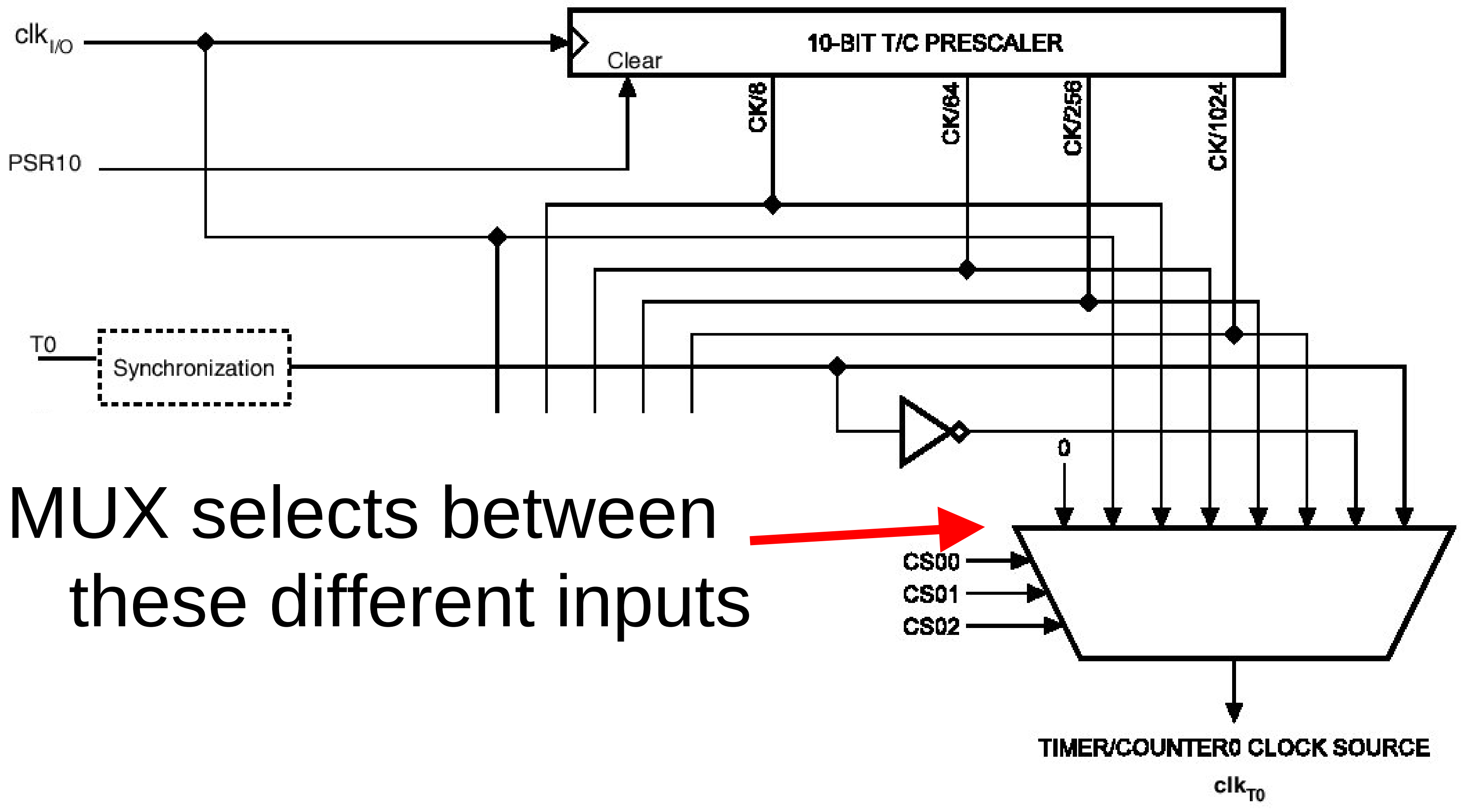
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10

Timer 0 Implementation



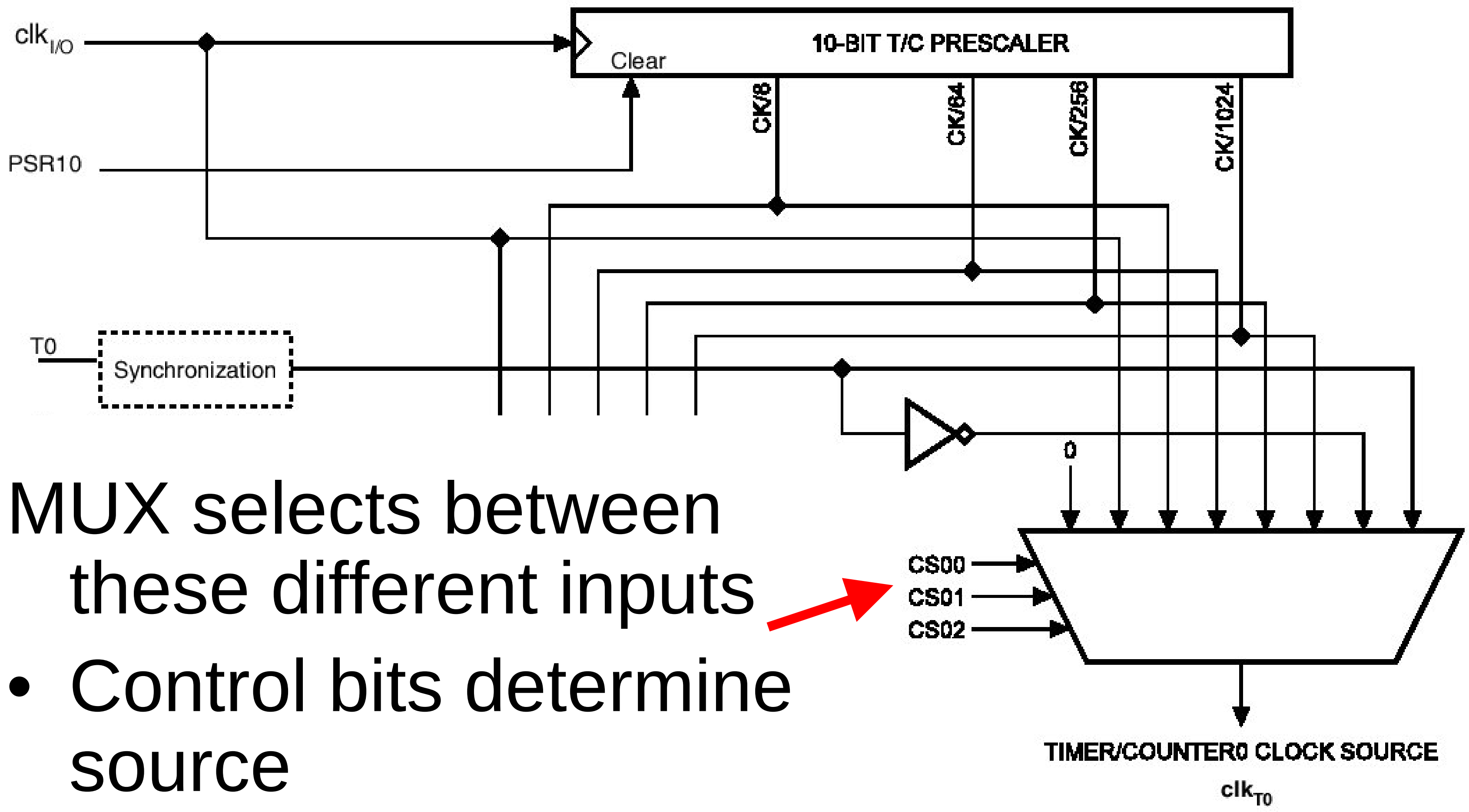
- Clock input to 10-bit counter
- Output bits: 3, 6, 8, and 10
  - These serve to divide the clock by the specified number of counts

Timer 0 Implementation



MUX selects between these different inputs

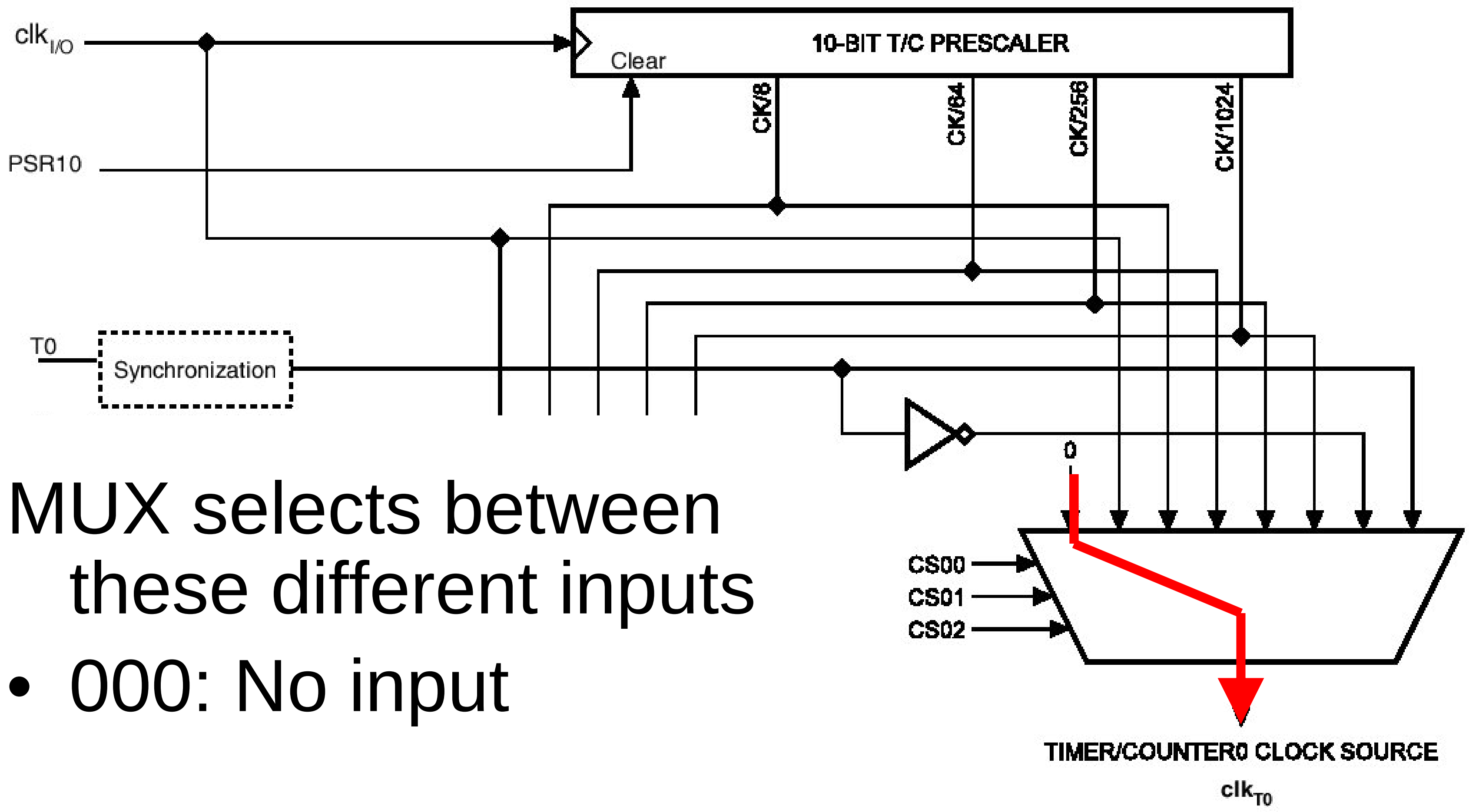
Timer 0 Implementation



MUX selects between these different inputs

- Control bits determine source

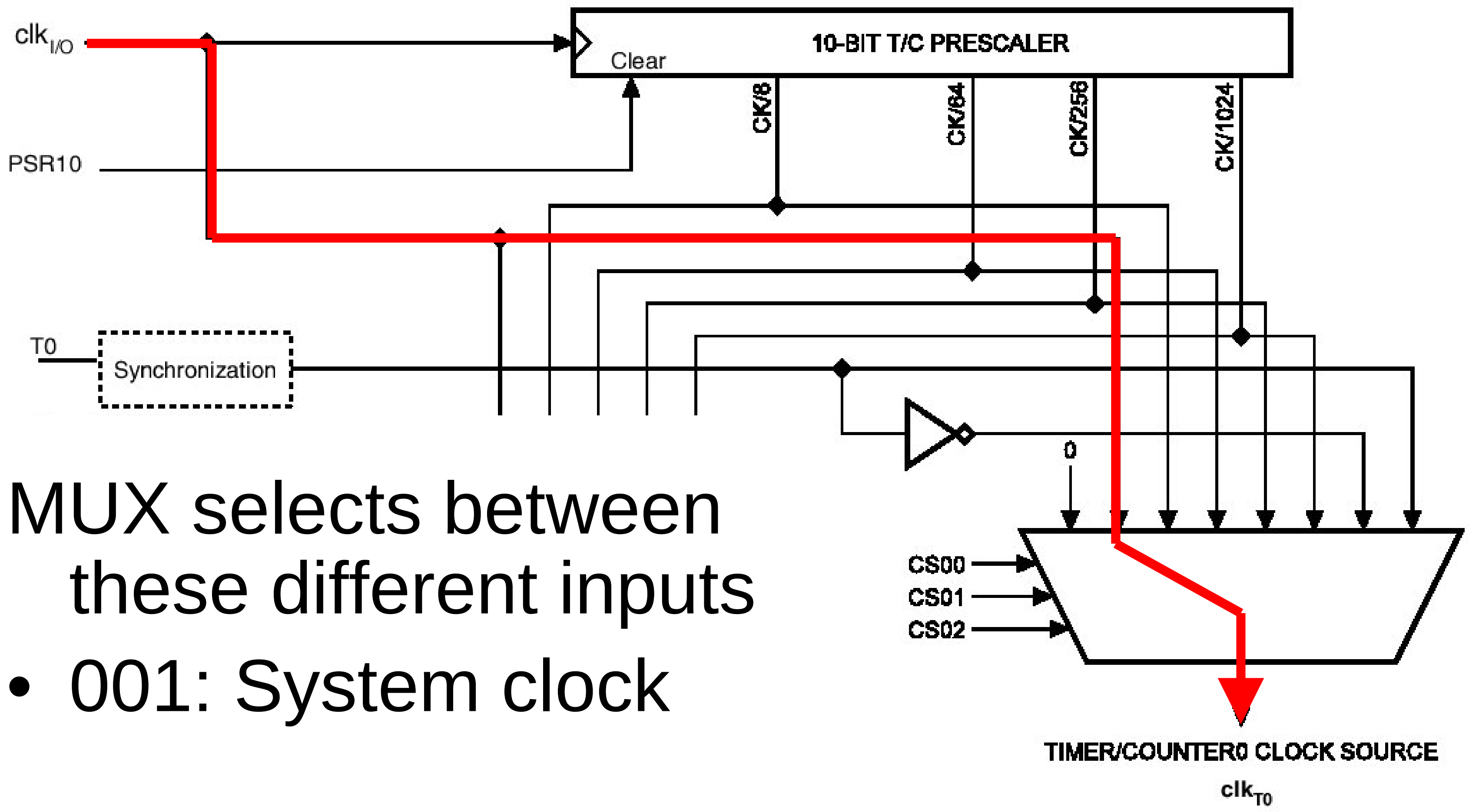
Timer 0 Implementation



MUX selects between these different inputs

- 000: No input

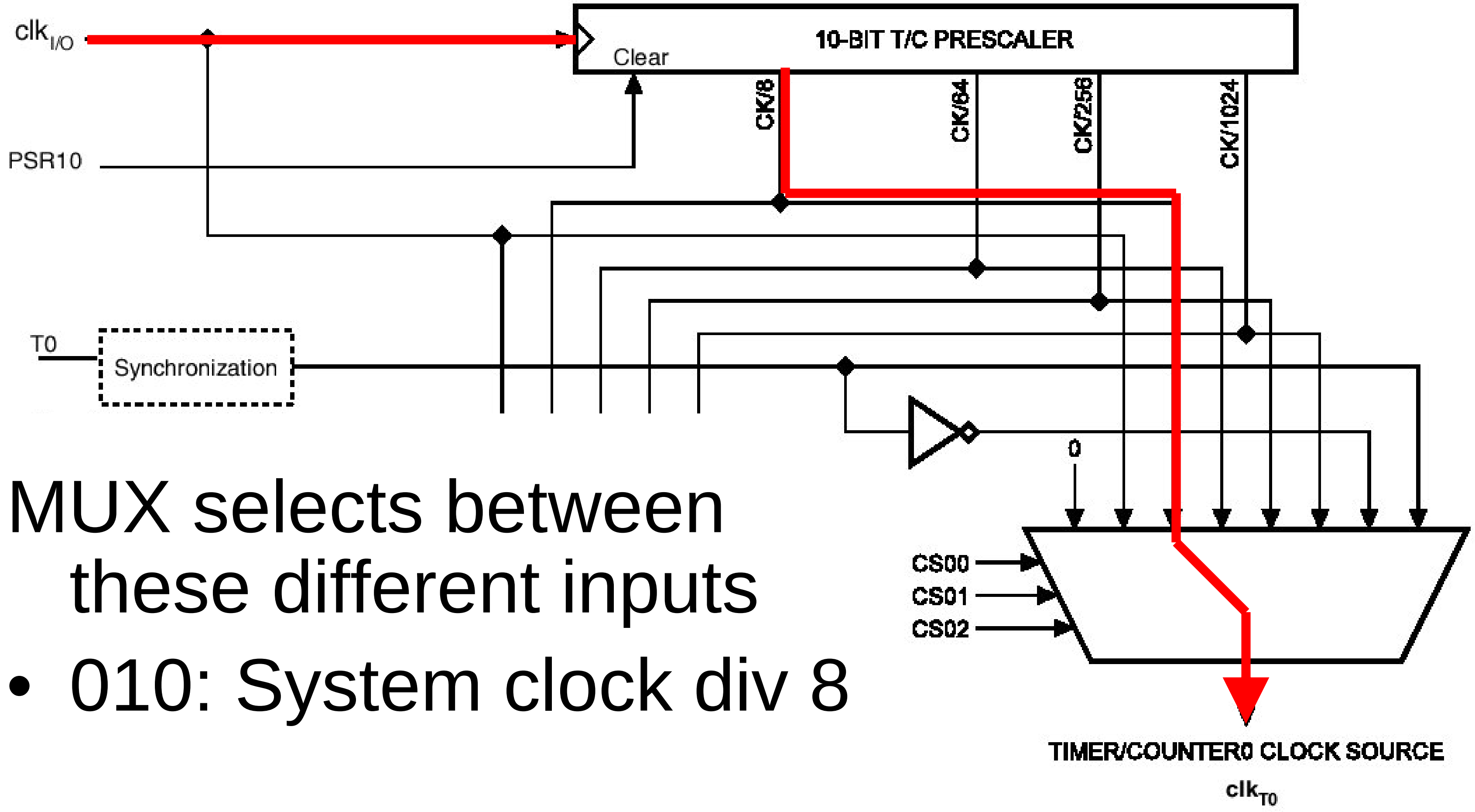
Timer 0 Implementation



MUX selects between these different inputs

- 001: System clock

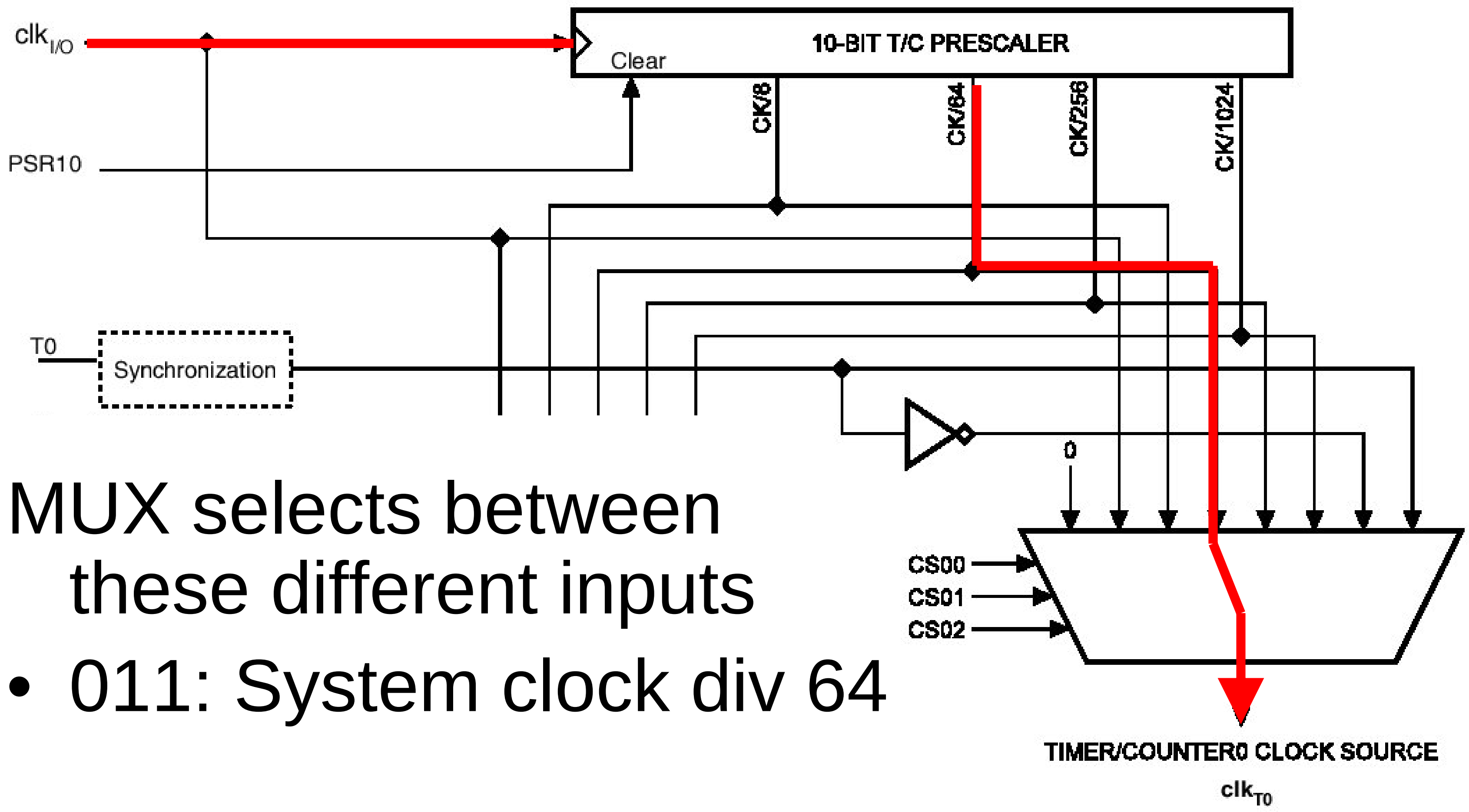
Timer 0 Implementation



MUX selects between these different inputs

- 010: System clock div 8

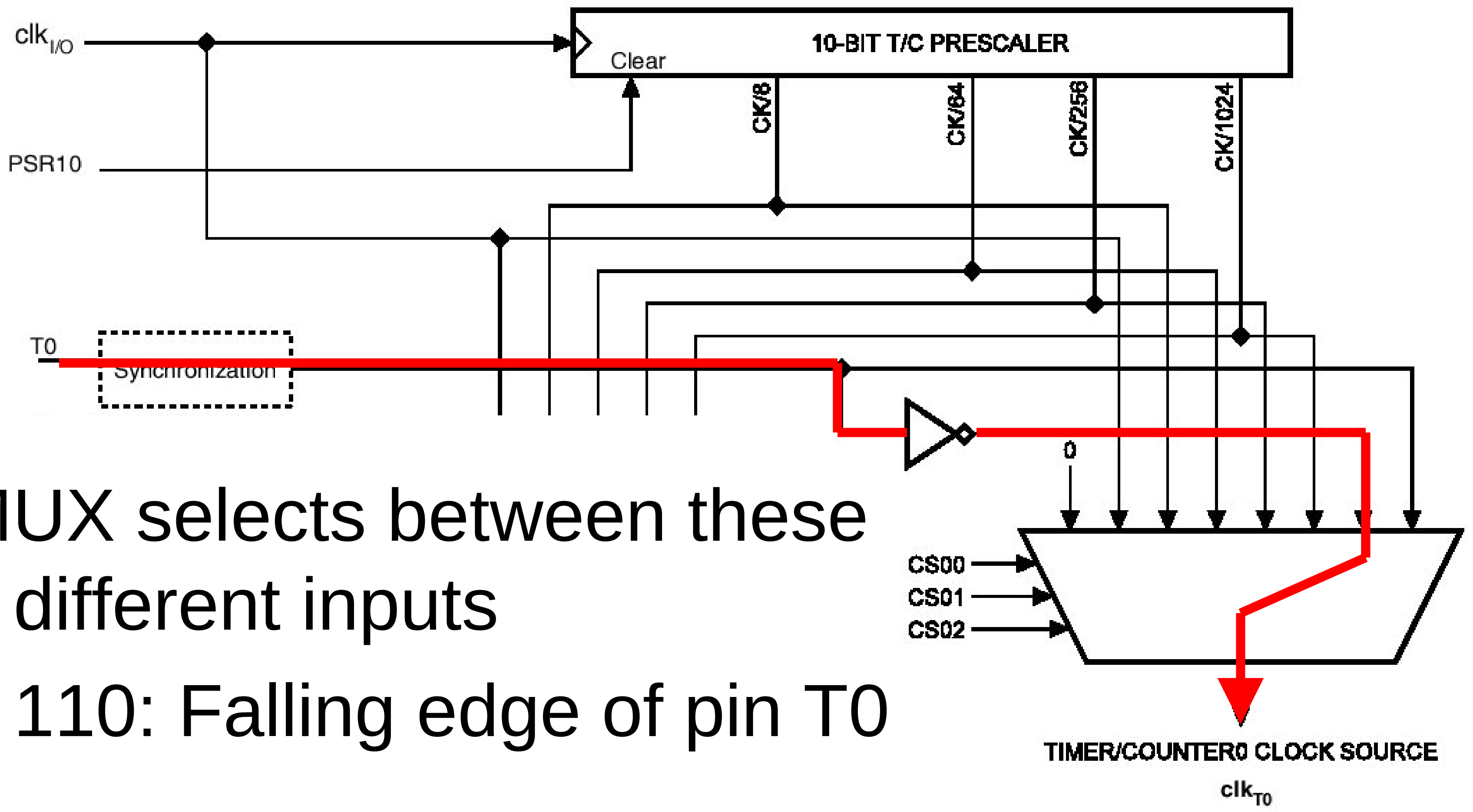
Timer 0 Implementation



MUX selects between these different inputs

- 011: System clock div 64

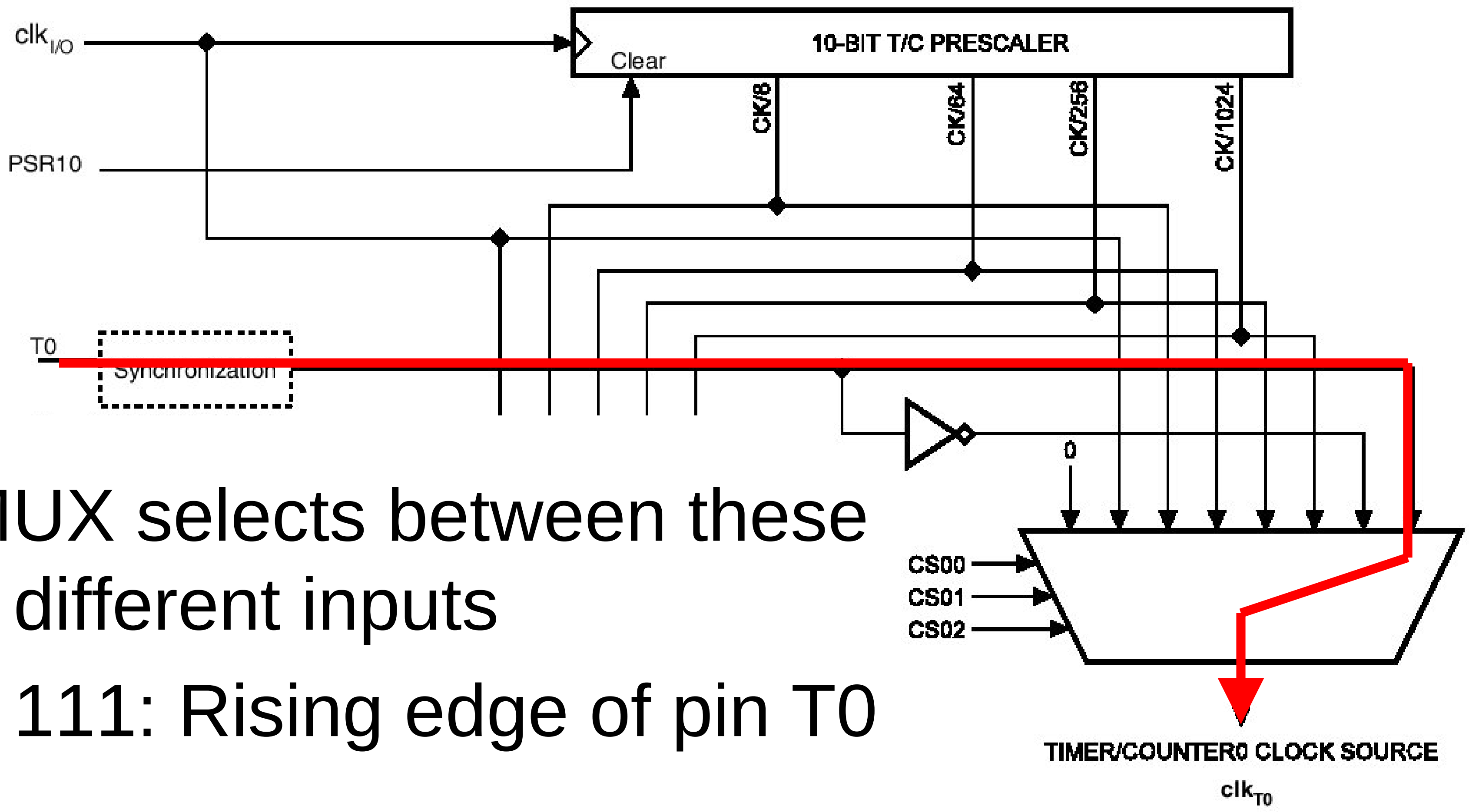
Timer 0 Implementation



MUX selects between these different inputs

- 110: Falling edge of pin T0

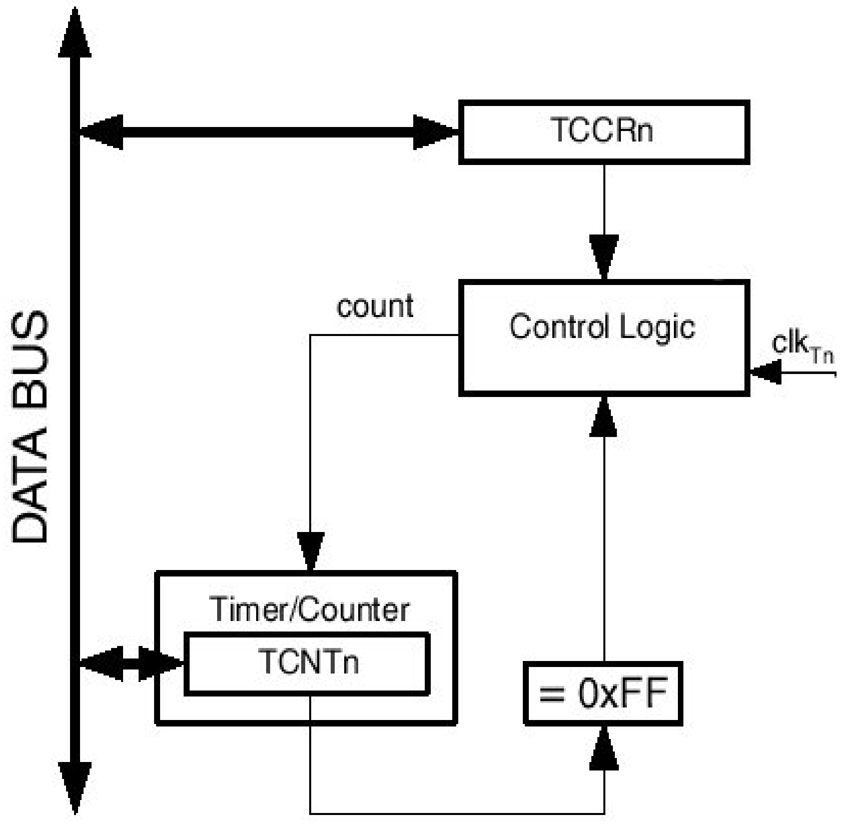
Timer 0 Implementation



MUX selects between these different inputs

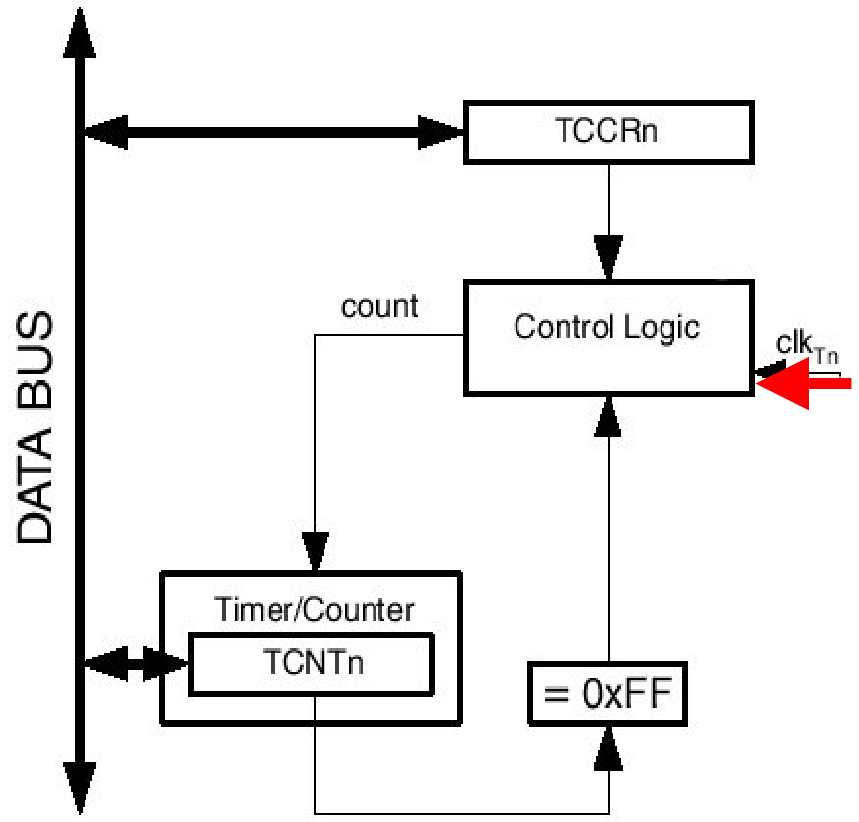
- 111: Rising edge of pin T0

Timer 0



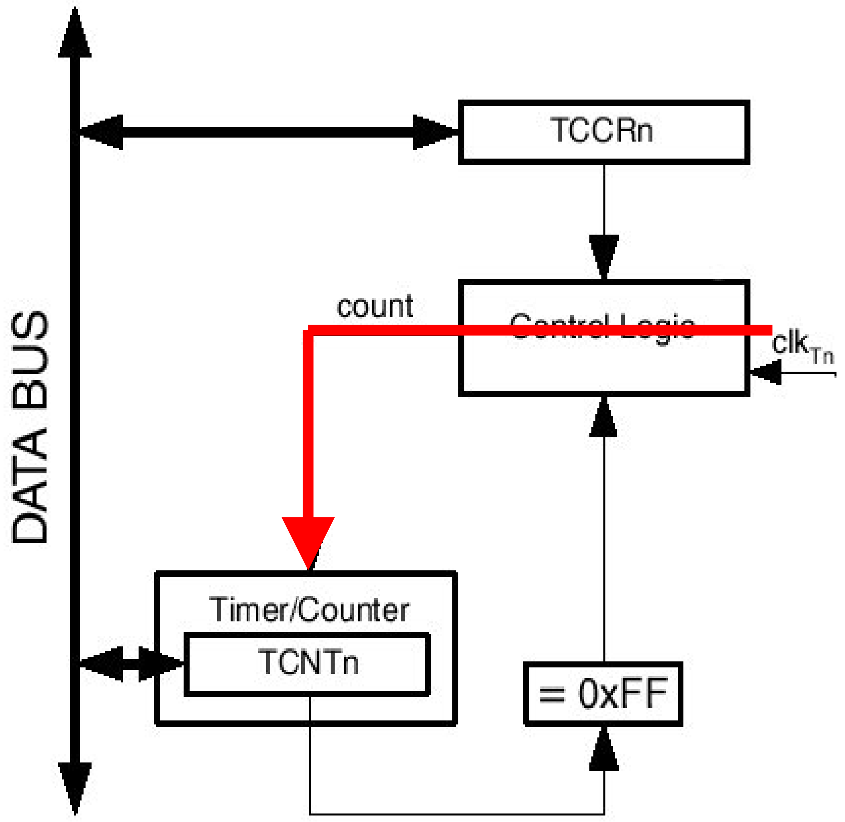
- TCNT0: 8-bit counter (a register)
- TCCR0: control register

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 Code Example

```
timer0_config(TIMER0_PRE_1024);   // Init: 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 of while loop after ~10 ms
```

See Atmel HOWTO for example code (timer_demo2.c)

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

Timer 0 Example

One caution:

- “something else” cannot take very much time

(we have a solution for this – coming soon!)

Next Example

How do we time a delay of 100 usecs?

Next Example

How do we time a delay of 100 usecs?

$$\begin{aligned} clock\_ticks * prescale &= .0001 * clock\_freq \\ &= .0001 * 16000000 \\ &= 1600 \end{aligned}$$

Next Example

How do we time a delay of 100 usecs?

$$
\begin{aligned}
clock\_ticks * prescale &= .0001 * clock\_freq \\
&= .0001 * 16000000 \\
&= 1600 \\
200 \quad * \quad 8 &= 1600 \\
OR & \\
25 \quad * \quad 64 &= 1600
\end{aligned}
$$

Timer 0 Code Example

```
timer0_config(TIMER0_PRE_8);   // Init: Prescale by 1024

timer0_set(0);       // Set the timer to 0

<Do something else for a while>
while(timer0_read() < 200) {
    <Do something while waiting>
};

// Break out of while loop after ~100 us
```

Example 3: Timing the Width of a Pulse

- Input: port B, pin 1
- How long is the pin high?

Example: Timing a Pulse Width

```
timer0_config(TIMER0_PRE_1024);   // Init: Prescale by 1024

// Wait for pin to go high
while(PINB & 0x1 == 0){};
timer0_set(0);       // Set the timer to 0

while((PINB & 0x1) != 0) {
   <Do something while waiting>
};
pulse_width = read_timer0();
```

Example: Timing a Pulse Width

What is the “resolution” of pulse_width?

Example: Timing a Pulse Width

What is the “resolution” of pulse_width?

- Each “tick”of pulse_width is:

$$delay = \frac{1024}{16{,}000{,}000} = 64\ \mu s$$

Example: Timing a Pulse Width

So, with pulse_width ticks:

$$delay = \frac{1024 * pulse\_width}{16{,}000{,}000} = 64 * pulse\_width\ \mu s$$

Example: Timing a Pulse Width

```
timer0_config(TIMER0_PRE_1024);   // Init: Prescale by 1024

// Wait for pin to go high
while(PINB & 0x1 == 0){};
timer0_set(0);       // Set the timer to 0

while((PINB & 0x1) != 0) {
   <Do something while waiting>
};
pulse_width = read_timer0();
```

Note: the longer "something" takes, the larger the error in timing

Other Note

See oulib.h for the list of possible prescalers for timer 0

Two Other Timers

Timer 1:

- 16 bit counter
- Prescalers: 1, 8, 64, 256, 1024

Timer 2:

- 8 bit counter
- Prescalers: 1, 8, 32, 64, 128, 256, 1024

Last Time

Counter/Timers

- Counting events: external events or clock ticks
- Prescalar divides the clock frequency (implemented as yet another counter)

Today

- Input/Output by Polling
- Interrupts
  - Processor is **interrupted** from what it is doing to perform some other task
  - Once done with the task, returns to what it was previously doing

Administrivia

- HW 3 due in class on Tuesday
- Midterm in 1 week
- Project 2

I/O By Polling

One possible approach: the processor continually checks the state of the device:

```
do {
  x = PINB & 0x10;
}while(x == 0);
y = PINC …
```

I/O By Polling

What is wrong with this approach?

I/O By Polling

What is wrong with this approach?

- In embedded systems, we are typically managing many devices at once

I/O By Polling

- We can potentially be waiting for a long time before the state changes
  - We call this **busy waiting**
- The processor is wasting time that could be used to do other tasks

What is one way to solve this?

I/O By Polling: An Alternative

Alternative: do something while we are waiting

```
do {
   x = PINB & 0x10;
  <go do something else>
}while(x == 0);
y = PINC …
```

I/O By Polling: An Alternative

Polling works great … but:

- We have to guarantee that our “something else” does not take too long (otherwise, we may miss the event)
- Depending on the device, “too long” may be very short

I/O by Polling

In practice, we typically reserve this polling approach for situations in which:

- We know the event is coming very soon
- We must respond to the event very quickly

(both are typically measured in nano- to micro- seconds)

An Alternative: Interrupts

- Hardware mechanism that allows some event to temporarily interrupt an ongoing task
- The processor then executes an **interrupt handler** (a small piece of code)
- Execution then continues with the original program

Some Sources of Interrupts (Mega8)

External:

- An input pin changes state
- The UART receives a byte on a serial input

Internal:

- A clock
- Processor reset
- The on-board analog-to-digital converter completes its conversion

Last Time

Interrupts

- Temporarily stopping the main program to handle a time-critical event
- The **interrupt handler** is a small piece of code (we try to make it as short as possible)
- Once the interrupt handler is done, execution continues with the main program as if nothing had happened

Today

- Generating regular interrupts
- Interrupts to produce PWM signals
- Interrupts in practice

Project 2 due on Thursday at the close of the lab

Interrupt Example

Suppose we are executing the "something else" code:

```
LDS R1 (A) ← PC
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

An Example

Suppose we are executing the "something else" code:

LDS R1 (A)

LDS R2 (B) $\leftarrow$ **PC**

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

Suppose we are executing the "something else" code:

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1  ← PC
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

An Example

An interrupt occurs (EXT_INT1):

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1  ← PC
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3        ← remember this location
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

An Example

Execute the interrupt handler

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

PC → LDS R1 (G)
     LDS R5 (L)
     ADD R1, R2
          :
     RETI
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

```
         EXT_INT1:

         LDS R1 (G)
PC ────▶ LDS R5 (L)
         ADD R1, R2
            :
         RETI
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

```
       EXT_INT1:

       LDS R1 (G)
       LDS R5 (L)
PC →   ADD R1, R2
            :
       RETI
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

```
      EXT_INT1:

      LDS R1 (G)
      LDS R5 (L)
      ADD R1, R2
PC →       :
      RETI
```

An Example

Return from interrupt

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

```
       EXT_INT1:

       LDS R1 (G)
       LDS R5 (L)
       ADD R1, R2
          :
PC ──▶ RETI
```

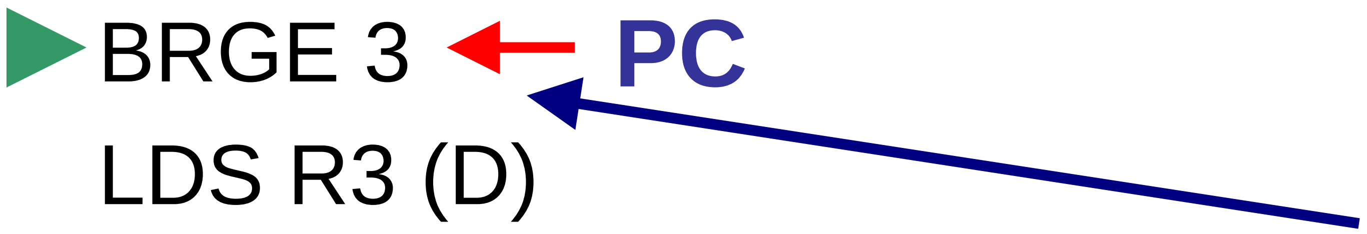
An Example

Return from interrupt

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
      :
RETI
```



An Example

Continue execution with original

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)  ← PC
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
      :
RETI
```

An Example

Continue execution with original

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1  ← PC
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
     :
RETI
```

Interrupt Routines

Generally a very small number of instructions

- We want a quick response so the processor can return to what it was originally doing
- No delays, waits, or floating point operations …

Back to our timer 0 example…

Timer 0 Code Example

```
timer0_config(TIMER0_PRE_1024);   // Init: 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 of while loop at ~10 ms
```

See Atmel HOWTO for example code (timer_demo2.c)

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

Timer 0 Example

One caution:

- “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 that the timer’s counter rolls over from 0xFF to 0x00

Timer 0 Interrupt Example

Suppose:

- 16MHz clock
- Prescaler of 1024

How often is the interrupt generated?

Timer 0 Example II

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

Timer 0
Interrupt Service Routine (ISR)

An ISR is a type of function that is called when the interrupt is generated

```
ISR(TIMER0_OVF_vect) {
    // Toggle the LED attached to bit 0 of port B
    PORTB ^= 1;
};
```

What is the flash frequency?

Timer 0
Interrupt Service Routine (ISR)

```
ISR(TIMER0_OVF_vect) {
    // Toggle the LED attached to bit 0 of port B
    PORTB ^= 1;
};
```

What is the flash frequency?

$$frequency = \frac{16,000,000}{1024 * 256 * 2} = 30.5176\ Hz$$

Example I: ISR Initialization in Main Program

```
// 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 from what is happening in the main program

Next Example: 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) {
    static uint8_t counter;
    ++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 (same as before)

```
// 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 Example II

What is the flash frequency?

Timer 0 Example II

What is the flash frequency?

$$frequency = \frac{16{,}000{,}000}{1024 * 256 * 61 * 2} \approx 0.5\ Hz$$

Interrupts and Timers

Timing can often involve a cascade of multiple counters:

- Prescalar (1 … 1024)
- Timer0 (256)
- Counter within an interrupt routine (any)

Each counter implements a frequency division

Information Encoding

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

- Parallel digital (e.g., for our Project 1)
- Serial digital (Project 2)
- Analog: use voltage to encode a value

Information Encoding

An alternative: pulse-width modulation (PWM)

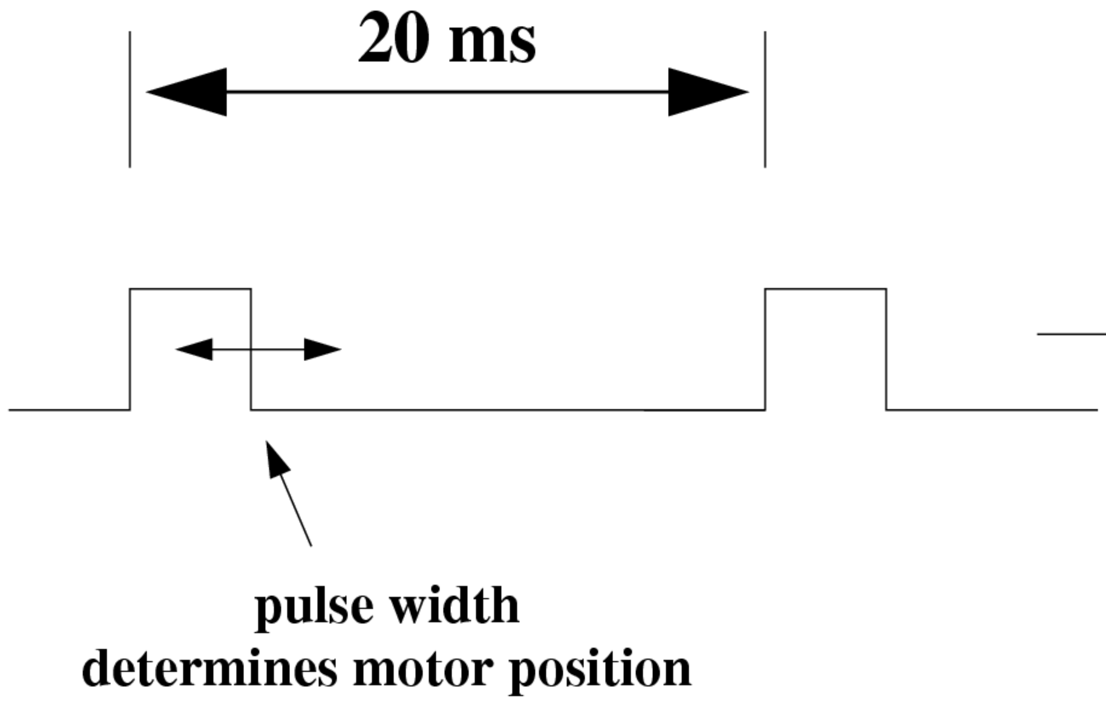
- Information is encoded in the time between the rising and falling edge of a pulse

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
- 1.5 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)
{

}
```

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

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

Interrupt Service Routines

- Should be **very** short
  - No “delays”
  - No busy waiting
  - Function calls from the ISR should be short also
  - Minimize looping
  - No “printf()”
- 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;
}
```

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;
```

to serial interrupt example…