ECE3411 – Fall 2016

Lecture 3a.

ISRs, Timer0
Task Based Programming

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

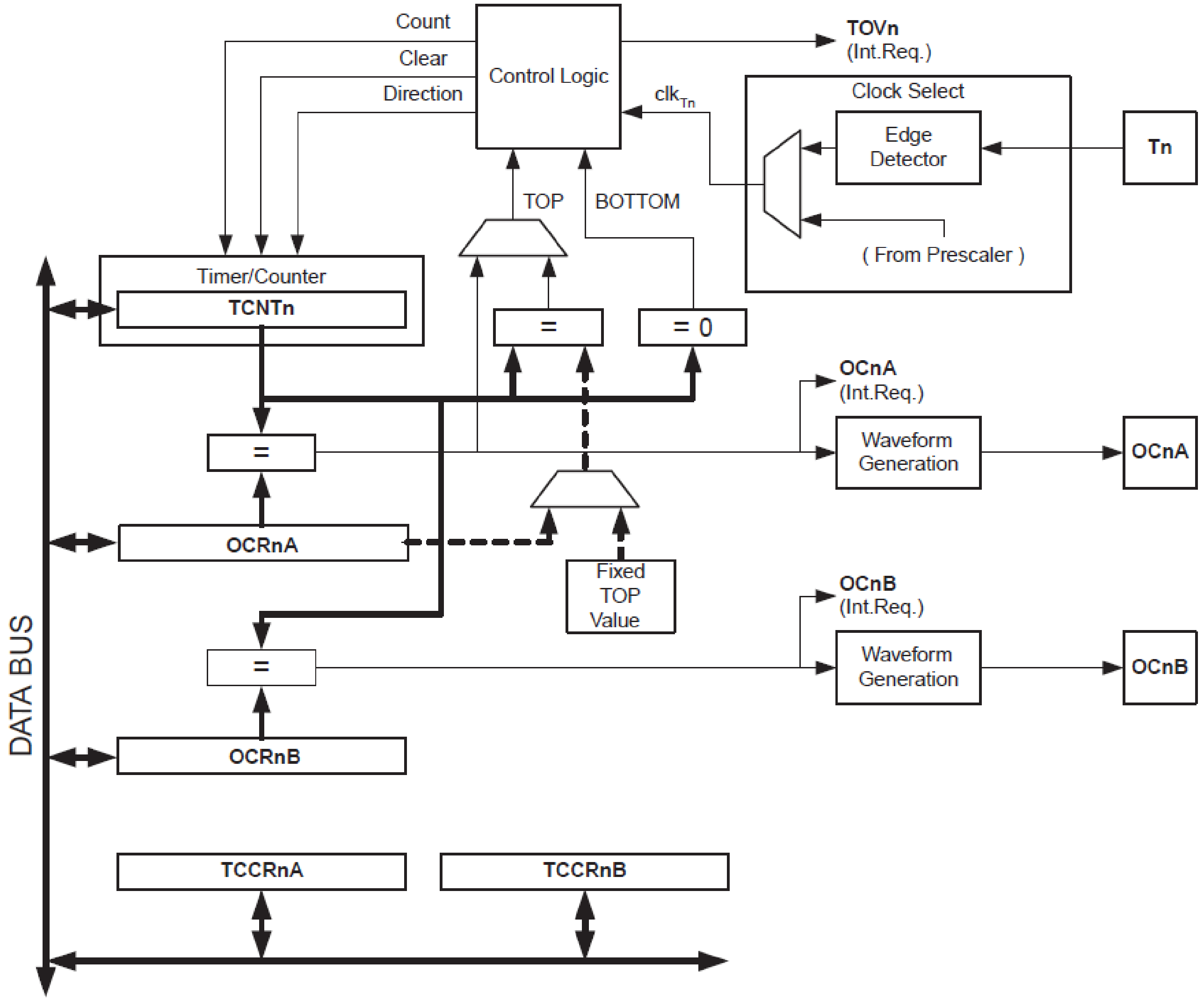
Copied from Lecture 3a, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UCONN

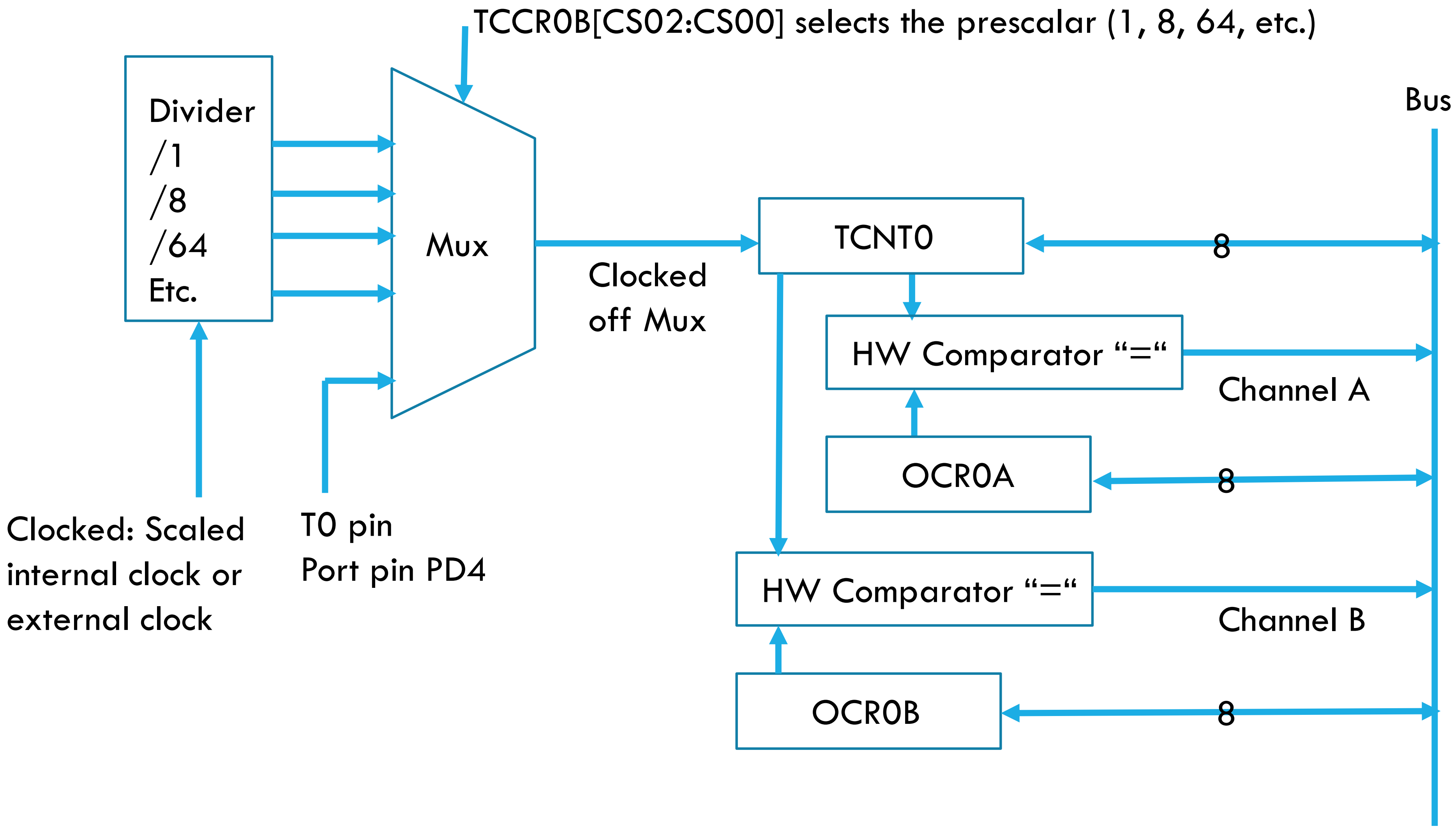
Based on the Atmega328P datasheet and material
from Bruce Land's video lectures at Cornel

Timer 0

Figure 14-1. 8-bit Timer/Counter Block Diagram



Timer 0



TCCR0B[CS02:CS00] selects the prescalar (1, 8, 64, etc.)
Divider
/1
/8
/64
Etc.
Mux
Clocked off Mux
Bus
TCNT0
8
HW Comparator "="
Channel A
OCR0A
8
HW Comparator "="
Channel B
OCR0B
8
Clocked: Scaled internal clock or external clock
T0 pin
Port pin PD4

Channels A and B

- TCNT0 and OCR0A are compared in HW, on equality:
  - Can clear TCNT0
  - Set interrupt flag (forces a HW event leading to possibly have the interrupt unit make the PC jump to the corresponding ISR)
  - Toggle an I/O line (Channel A), etc.

- TCNT0 and OCR0B are compared in HW, on equality as above
  - Except clearing TCNT0 is not an option

- Channels A and B can be used for PWM (discussed in a couple of weeks)

TCCR0A, TCCR0B

14.9.1 TCCR0A – Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.9.2 TCCR0B – Timer/Counter Control Register B

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

WGM00, WGM01, WGM02 → Waveform generation mode

CS00, CS01, CS02 → Controls the rate of the Mux

TCCR0A, TCCR0B

Table 14-8. Waveform Generation Mode Bit Description

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on[1][2]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

Table 14-9. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Waveform Generation Mode sets autoclear on matching OCR0A if TCCR0A |= (1<<WGM01);

- TCNT0 increments to OCR0A, is reset back to 0, and starts incrementing again
- TCNT0 follows a sawtooth

Every increment of TCNT0 is clocked using F_CPU/prescaler

- E.g., for F_CPU = 1MHz, then after TCCR0B = 2; each TCNT0 increment takes 8/(1MHz) = 8 micro seconds
- For OCR0A = 124, TCNT0 transitions from 0→1, 1→2, …, 123→124, 124→0, each transition taking 8 micro second giving one full period of 125*8 micro seconds, i.e., 1ms

Enabling an ISR every period can be used to create a precise 1ms clock!

Building a SW 1ms clock from HW Timer 0

14.9.6 TIMSK0 – Timer/Counter Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- TOIE0: timer 0 overflow interrupt enable
- OCIE0A: timer 0 output compare interrupt enable A
  - Set TIMSK0 = 2;
  - Program ISR(TIMER0_COMPA_vect) { SWTaskTimer++;}
  - Initialize global variable volatile int SWTaskTimer=0;
- Now SWTaskTimer is a reliable clock which increments every 1ms !
  - Suppose your task is to toggle a LED every 1/2 seconds (a 1Hz signal), then you can add in your main while loop the instruction if (SWTaskTimer == 500) { LEDToggle(); SWTaskTimer == 0;}
  - This avoids using the blocking delay functionality and allows other tasks to execute while waiting for the next moment at which the MCU should toggle the LED again

Putting It Together: Task Based Programming

```
....
int TaskTime = 500;
volatile int SWTaskTimer=TaskTime;

ISR(TIMER0_COMPA_vect)
{
   if (SWTaskTimer>0) {SWTaskTimer--;}
}

// 1ms ISR for Timer 0 assuming F_CPU = 1MHz
void InitTimer0(void)
{
   TCCR0A |= (1<<WGM01);
   OCR0A = 124;
   TIMSK0 =2;
   TCCR0B = 2; //Timer starts
}
....
```

```
int main(void)
{
   ...
   InitTimer0();
   ...
   sei(); // Enable global interrupt

   while(1)
   {
      if (SWTaskTimer == 0)
      {
         Task();
         SWTaskTimer == TaskTime;
      }
   }

   return 0;
}
```

Using Prescalars

- E.g., can we use prescaler = 1 for a 1ms clock?
- Each TCNT0 increment takes 1/(1MHz) = 1 micro seconds
- 1ms = 1000 TCNT0 increments → OCR0A must be equal to 1000-1=999
- Does not fit an 8-bit register/character!

- E.g., can we use prescalar 64 instead?
- Each TCNT0 increment takes 64/(1MHz) = 64 micro seconds
- 1ms = 1ms / 64 us = 1000/64 = 15.625 TCNT0 increments
- OCR0A is an integer: it must be either 14 or 15, giving a 15*64 um = 0.96ms period or a16*64 um = 1.024ms period
- SW clock is off by 2.4% (OCR0A=15 yields the least noise)

Performance Overhead Caused by ISR

- Current setting TCNT0 increments every 8um (prescalar set to 8) and ISR is triggered every 125 increments/ticks (our 1ms clock implementation)
- ISR takes 120 cycles = 120/1MHz = 120um = 120/8 ticks = 15 ticks → within one full period of 125 ticks, 15 are used up for the ISR, 15/125 = 12% of the time (lots of overhead)
- Can we do better?
  - Do we need a 1ms SW counter or does our application allows something larger? E.g., if TaskTime = 500 ms then we can use a 0.5s SW counter! How do you now initialize Timer0 and what performance overhead does this cost?
  - Use higher clock speed: Can we scale the internal clock up to 8MHz? Or do we use an external clock of say 16MHz? What do we have?
- Can we do worse? E.g., suppose we initialize Timer0 so that each period takes only 96um; for 8um TCNT0 ticks, set OCR0A = 15. Since 96<120, the ISR is always busy and incrementing at 120um (not at 96um):
  - There is no real forward progress on the main code: a forced 1 instruction every 120um as if the MCU is running at 4 cycles/ 120 micro second = 1/30 MHz!
  - The software clock is completely off

Removing Blocking delay_ms()

- Task Based Programming shows how to remove delay_m() from the main while loop
- What about a procedure/task that uses delay_ms()?
- Suppose you create code which writes a 16 character string on each line: this takes 32 LCD_GoTO commands and 32 LCDDataWrites, each taking 4ms due to delay_ms(1) delays → Takes 250ms
- During these 250ms nothing else happens, in particular, if you have a software routine that adapts a PWM signal using the hardware timers, then this routine is interrupted for 250ms.
- This means that the PWM signal remains unchanged for this period. If the LCD string writes are programmed to happen every 1s you will hear clicks/glitches every 1s.
- Even if you write just 1 character every say 40ms, this will introduce a new frequency of 25Hz (1000/40) to the spectrum of your PWM signal, which is in your hearing range.

Removing Blocking delay_ms()

```
void TaskAB(inputAB)
{
    CodeA;
   delay_ms(WaitTime);
    CodeB;
}

int main(void)
{ …
   while(1)
   {
       if (CondAB)
       {
          TaskAB(InpAB);
          ResetCondAB;
       }
   }
   …
}
```

```
void TaskA(InpAB)
{
    CodeA;
    InputB = CaptureCurrentStateCodeA;
}

ISR(TIMER0_COMPA_vect)
{
   if (TimerABWaiting>0 && WaitingFor==B)
   {  TimerABWaiting--;  }
}

void TaskB(InpB)
{
    RecoverStateEndOfCodeA(InpB);
    CodeB;
}
```

```
int main(void)
{ …
   while(1)
   {
      if (CondAB && WaitingFor==A)
      {
         TaskA(InpAB);
         WaitingFor = B;
         TimerABWaiting == WaitTime;
      }
      if (WaitingFor==B && TimerABWaiting==0)
      {
        TaskB(InpB);
        WaitingFor = A;
        ResetCondAB;
      }
   }
   …
}
```

Serves as “Busy Signal” and “FSM state”

Without WaitingFor Multiple threads may start to interfere

Multiple Threads

- CodeA executes on InpAB and at the end captures it state in InpB
- While waiting for starting execution of CodeB (and resume from state InpB), which takes WaitTime ms, the main while loop starts to execute CodeA again …
- Ouch: a new end state of CodeA is captured in InpB and overwrites the old one!
- The first call to “TaskAB” will never finish to completion and is essentially discarded.
- We need to remember a priority queue of states InpB for each call to “TaskAB” in the main while loop → needs a pointer structure
  - Ouch, what happens if the task consists of multiple code portions separated by delay_ms() commands
  - What if the delay_ms() command is in a while loop or for loop …
  - What if a task calls another task that has a delay_ms() operation …
  - We need a smart queue which remembers all the states (like InpB) of all the procedures the main while loop is waiting for; in addition it needs to remember what needs to execute in-order (according to a priorty queue) and what can be executed in parallel ..
  - Need an operating system (OS), a tiny one as we have limited storage in the MCU