ECE3411 – Fall 2017
Lecture 2a.

Interrupts & Interrupt Service Routines (ISRs)

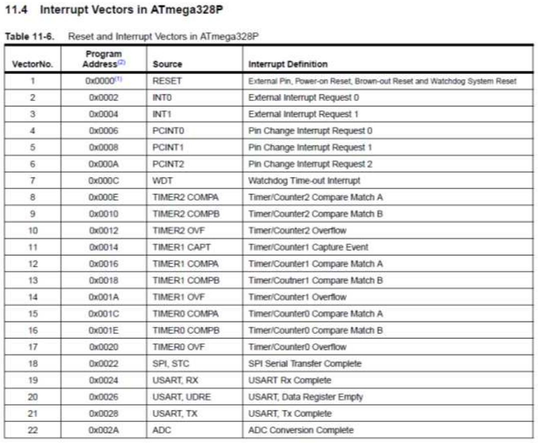
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Copied from Lecture 2c, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

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

Interrupts

Lower range of program storage in flash:



11.4 Interrupt Vectors in ATmega328P

Table 11-6. Reset and Interrupt Vectors in ATmega328P

VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	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/Coutner1 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

If you want to set the mask bit of an interrupt, i.e., you enable a certain interrupt, then you ***must*** write a corresponding ISR (interrupt service routine).

The table contains the address of the ISR that you write (upon the HW event that will cause the interrupt, the program counter will jump to the address indicated by the table to execute the programmed ISR).

Program memory has 2^16 registers
→ an address has 16 bits, e.g., 0xabcd
→ 0xabcd is stored in two 8-bit registers
→ Interrupt vector table associates interrupt vectors to addresses 0x0000, 0x0002, 0x0004 etc. (by increments of 2)

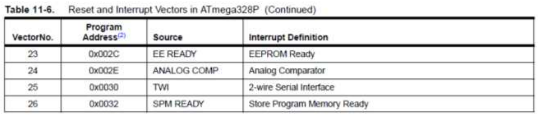
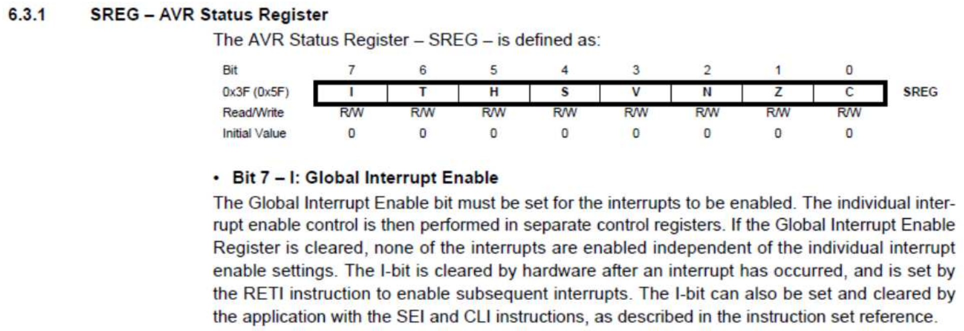


Table 11-6. Reset and Interrupt Vectors in ATmega328P (Continued)

VectorNo.	Program Address[2]	Source	Interrupt Definition
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Program Layout

- Initialization procedure:
  - Set up tables,
  - Initialize timers,
  - Do bookkeeping before you can put on interrupts
  - Turn on the master interrupt bit: This is the I-bit in register SREG, the C-macro sei() does this for you



6.3.1 SREG – AVR Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**

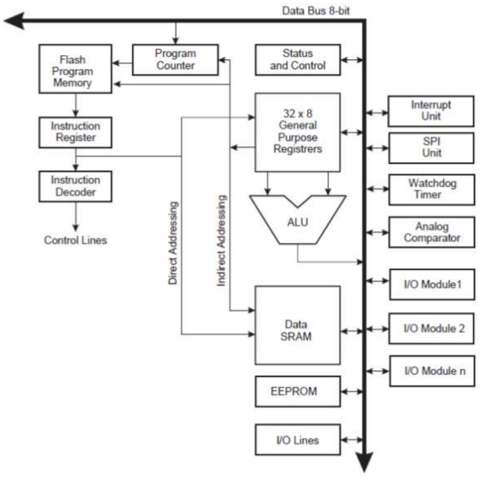
The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

Program Layout

- Main() executes slow background code forever → you never exit main in a MCU
  - Interrupt driven tasks are asynchronously called from main, for example:
    - a HW timer may cause a HW event every 1000 cycles, upon which in the corresponding ISR a SW counter is incremented;
    - upon reaching an a-priori defined maximum value, the background code calls a corresponding procedure which executes some task, and upon returning the SW counter is reset to 0
- ISRs have no parameters, no return value, they save CPU state (and C does this for you); they are called by HW events:
  - E.g., the bit value for position RXC0 in UCSR0A goes to high when receiving a character is completed
  - If the mask bit in UCSR0B for position RXCIE0 is set [meaning that an interrupt is enabled for the flag (UCSR0A & (1<<RXC0))], then the MCU will jump to the address of the ISR as indicated by the interrupt vector table for source USART, RX.
  - It takes about 75 cycles to go in and out of an ISR; another 32 cycles to safe state of the MCU (32 registers); another 7/8 cycles overhead.

AVR Architecture

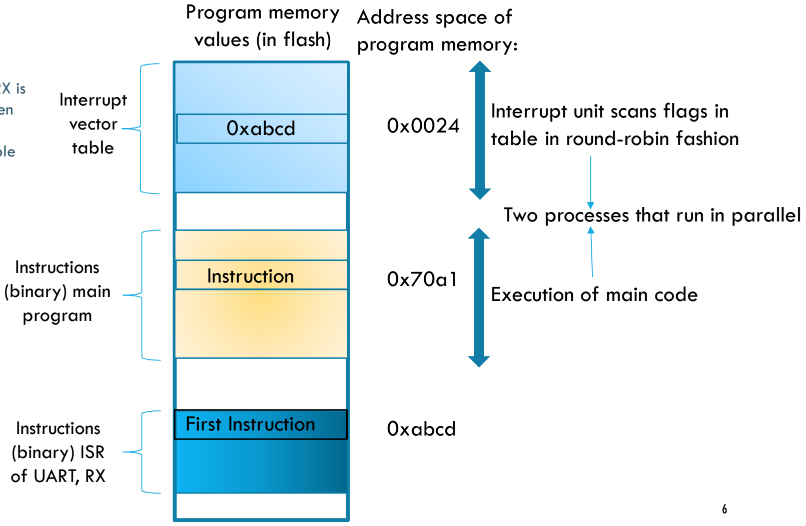
Figure 6-1. Block Diagram of the AVR Architecture



Execution of an ISR

1. Character receive complete
→ UART RX HW event which makes flag UCSR0A & (1<<RXC0) non-zero
→ If UCSR0B |= (1<<RXCIE0), i.e., ISR UART RX is enabled, then the interrupt unit sees flag when checking for the UART RX HW event
→ Interrupt unit looks at the Interrupt vector table at position 0x0024 for UART RX, and reads address 0xabcd

2. Program counter (PC) points at 0x70a1, corresponding instruction is executed to completion
→ 0x70a1 is pushed on to the PC stack
→ PC becomes 0xabcd
→ ISR UART RX is executed
→ Upon return from interrupt, the PC stack pops the value 0x70a1
→ PC gets the next address and main program continues its execution



Program memory values (in flash)

Address space of program memory:

Interrupt vector table

0xabcd

0x0024

Interrupt unit scans flags in table in round-robin fashion

Two processes that run in parallel

Instructions (binary) main program

Instruction

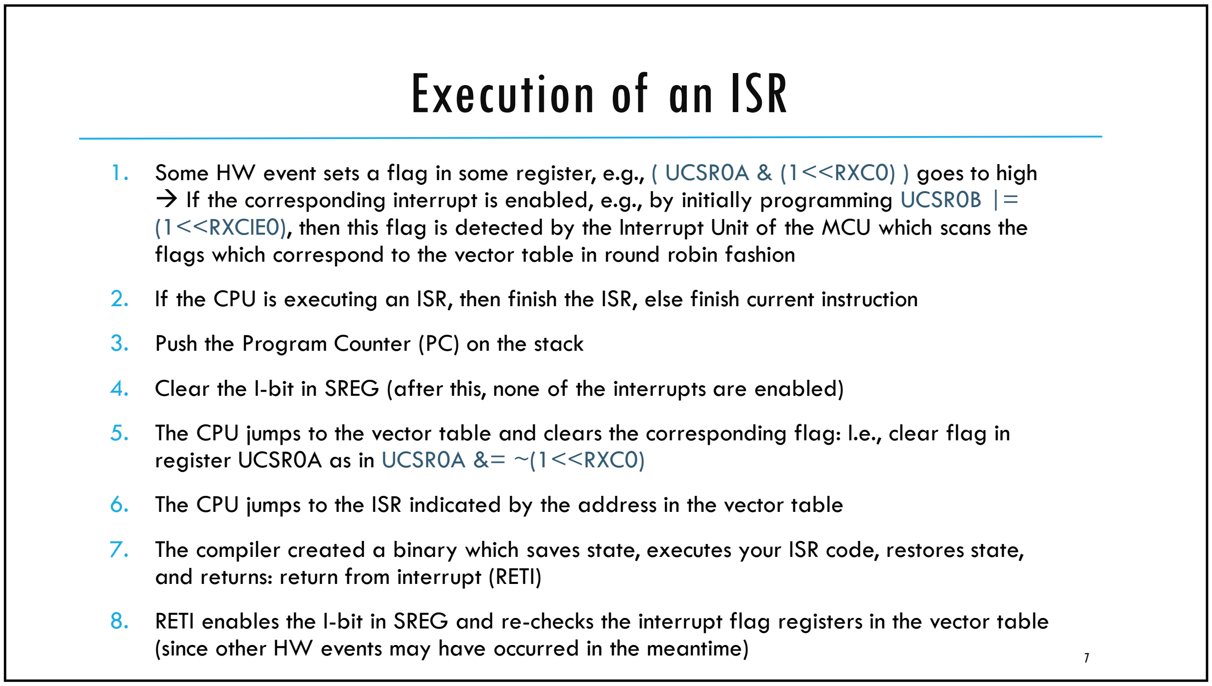
0x70a1

Execution of main code

Instructions (binary) ISR of UART, RX

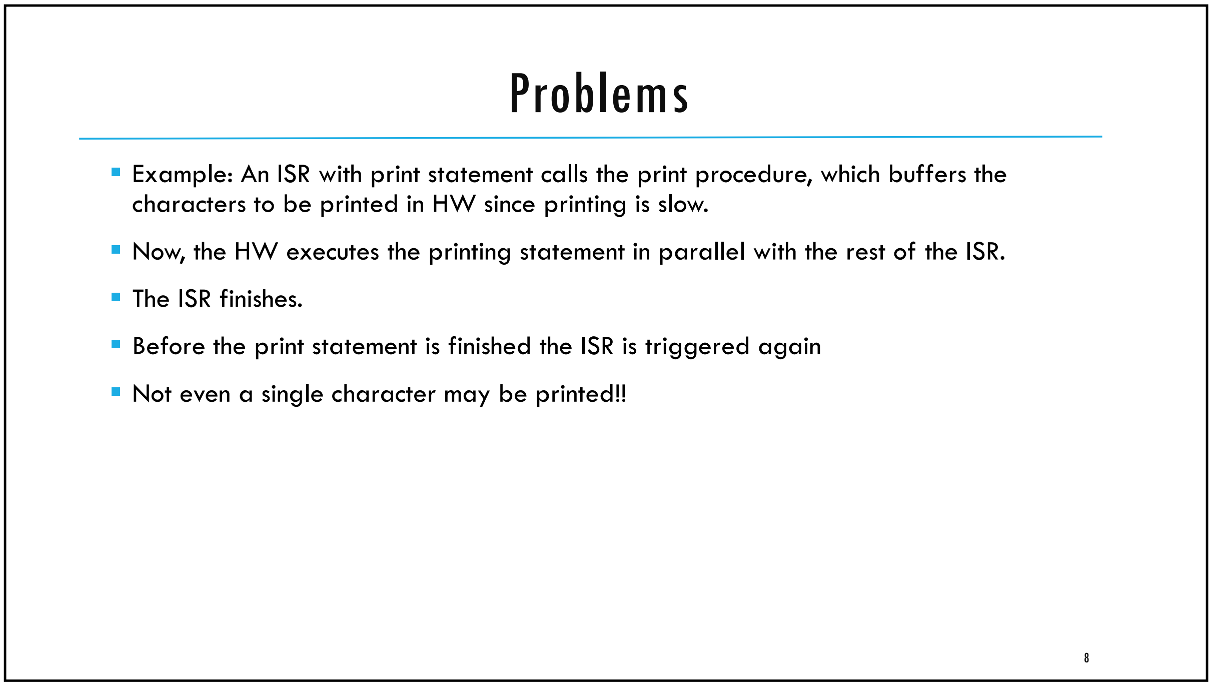
First Instruction

0xabcd



Execution of an ISR

1. Some HW event sets a flag in some register, e.g., (UCSR0A & (1<<RXC0)) goes to high → If the corresponding interrupt is enabled, e.g., by initially programming UCSR0B |= (1<<RXCIE0), then this flag is detected by the Interrupt Unit of the MCU which scans the flags which correspond to the vector table in round robin fashion
2. If the CPU is executing an ISR, then finish the ISR, else finish current instruction
3. Push the Program Counter (PC) on the stack
4. Clear the I-bit in SREG (after this, none of the interrupts are enabled)
5. The CPU jumps to the vector table and clears the corresponding flag: I.e., clear flag in register UCSR0A as in UCSR0A &= ~(1<<RXC0)
6. The CPU jumps to the ISR indicated by the address in the vector table
7. The compiler created a binary which saves state, executes your ISR code, restores state, and returns: return from interrupt (RETI)
8. RETI enables the I-bit in SREG and re-checks the interrupt flag registers in the vector table (since other HW events may have occurred in the meantime)



Problems

- Example: An ISR with print statement calls the print procedure, which buffers the characters to be printed in HW since printing is slow.
- Now, the HW executes the printing statement in parallel with the rest of the ISR.
- The ISR finishes.
- Before the print statement is finished the ISR is triggered again
- Not even a single character may be printed!!

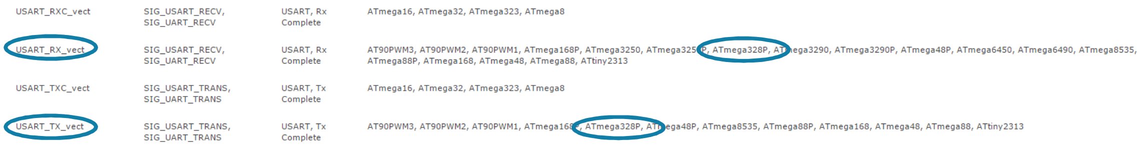
Problems

- In your ISR you may enable the master interrupt bit → this creates a nested interrupt → not recommended

- Memory of one event deep: e.g.,
  - MCU handles a first flag of (UCSR0A & (1<<RXC0))
  - After clearing this flag, the same HW event happens again which will again set the interrupt flag vector for (UCSR0A & (1<<RXC0)) (which will be handled after the current interrupt)
  - But more interrupts for (UCSR0A & (1<<RXC0)) are forgotten while handling the current interrupt (first flag)!!
  - You need to write ***efficient*** ISR code to avoid missing HW events, which may cause your application to be unreliable.

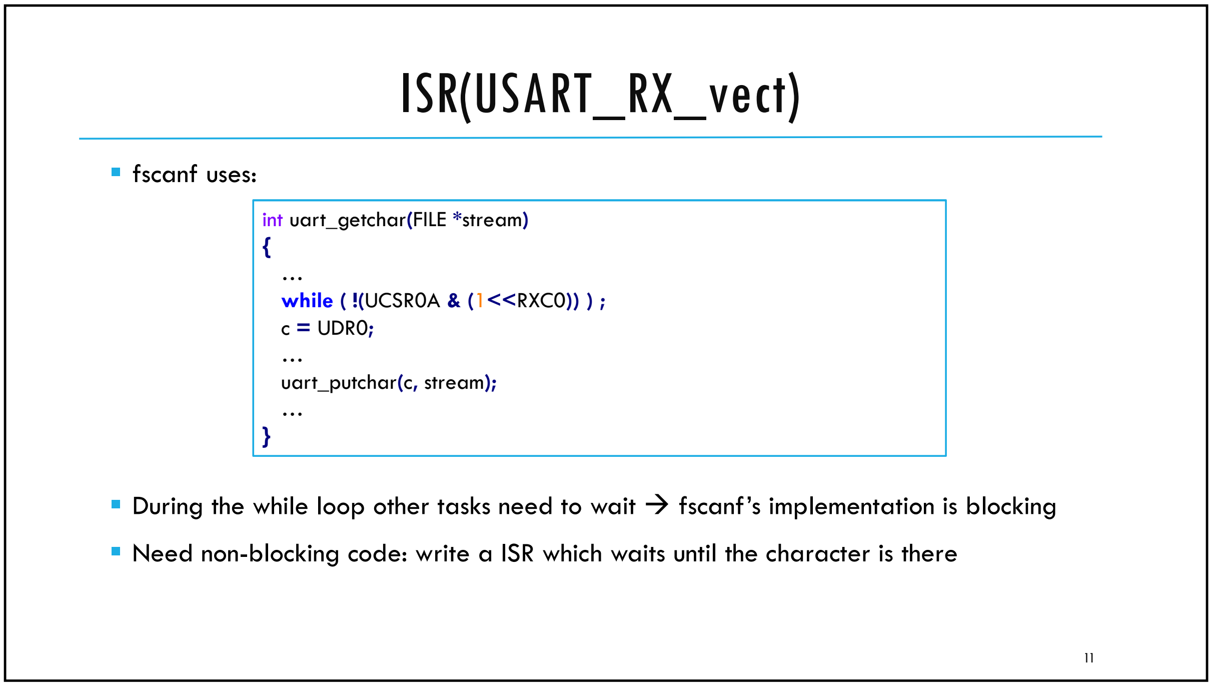
Interrupt Service Routine (ISR)

- http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__interrupts.html
- #include <avr/interrupt.h>



USART_RXC_vect	SIG_USART_RECV, SIG_UART_RECV	USART, Rx Complete	ATmega16, ATmega32, ATmega323, ATmega8
USART_RX_vect	SIG_USART_RECV, SIG_UART_RECV	USART, Rx Complete	AT90PWM3, AT90PWM2, AT90PWM1, ATmega168P, ATmega3250, ATmega325P, ATmega328P, ATmega3290, ATmega3290P, ATmega48P, ATmega6450, ATmega6490, ATmega8535, ATmega88P, ATmega168, ATmega48, ATmega88, ATtiny2313
USART_TXC_vect	SIG_USART_TRANS, SIG_UART_TRANS	USART, Tx Complete	ATmega16, ATmega32, ATmega323, ATmega8
USART_TX_vect	SIG_USART_TRANS, SIG_UART_TRANS	USART, Tx Complete	AT90PWM3, AT90PWM2, AT90PWM1, ATmega168P, ATmega328P, ATmega48P, ATmega8535, ATmega88P, ATmega168, ATmega48, ATmega88, ATtiny2313

- We need to program ISR(USART_RX_vect)



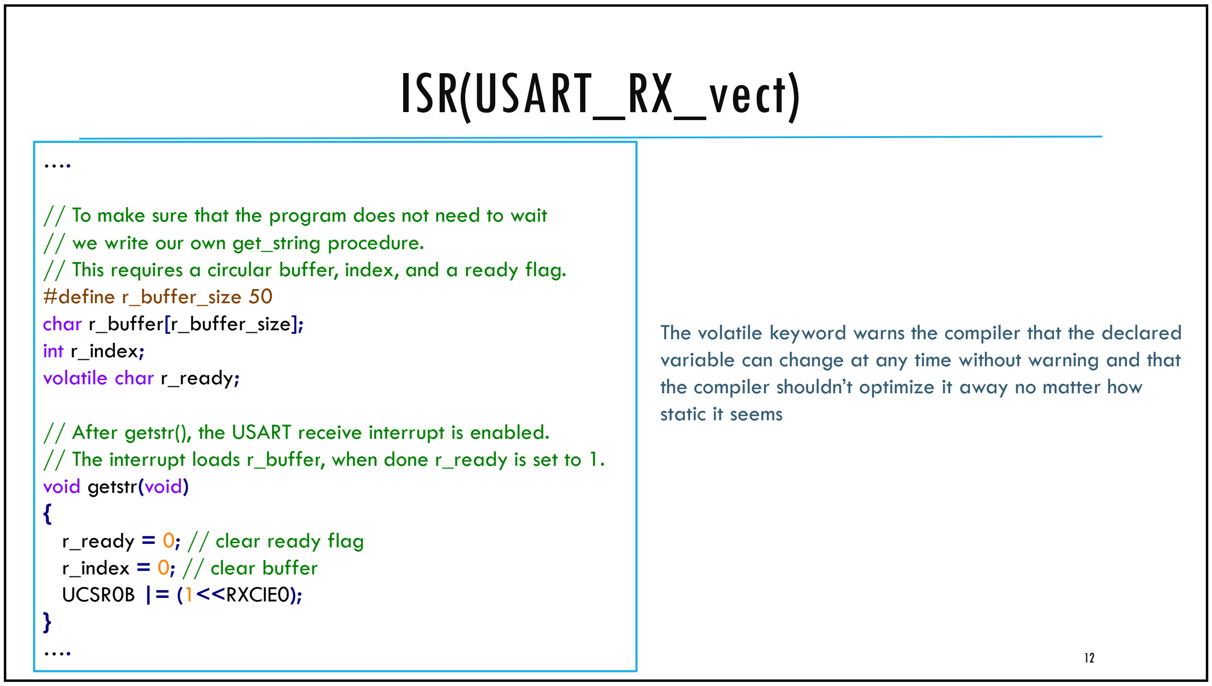
ISR(USART_RX_vect)

- fscanf uses:

```
int uart_getchar(FILE *stream)
{
   …
   while ( !(UCSR0A & (1<<RXC0)) ) ;
   c = UDR0;
   …
   uart_putchar(c, stream);
   …
}
```

- During the while loop other tasks need to wait → fscanf’s implementation is blocking
- Need non-blocking code: write a ISR which waits until the character is there

11

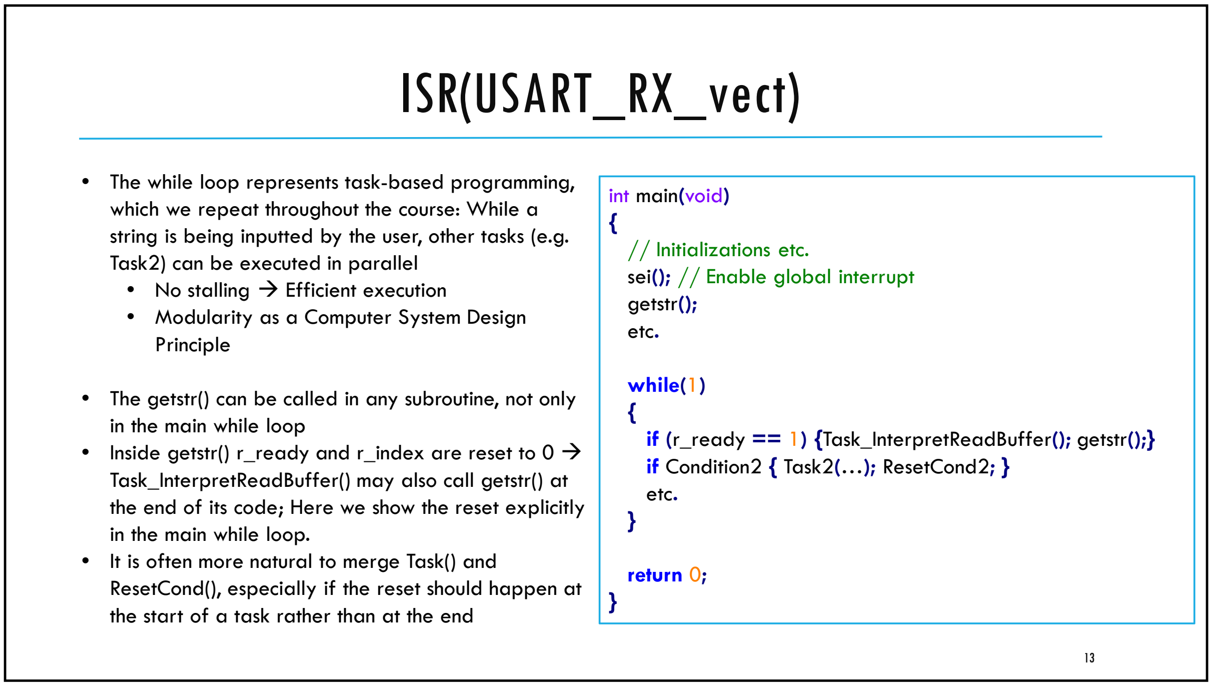


ISR(USART_RX_vect)

```
….

// To make sure that the program does not need to wait
// we write our own get_string procedure.
// This requires a circular buffer, index, and a ready flag.
#define r_buffer_size 50
char r_buffer[r_buffer_size];
int r_index;
volatile char r_ready;

// After getstr(), the USART receive interrupt is enabled.
// The interrupt loads r_buffer, when done r_ready is set to 1.
void getstr(void)
{
   r_ready = 0; // clear ready flag
   r_index = 0; // clear buffer
   UCSR0B |= (1<<RXCIE0);
}
….
```

The volatile keyword warns the compiler that the declared variable can change at any time without warning and that the compiler shouldn’t optimize it away no matter how static it seems

12

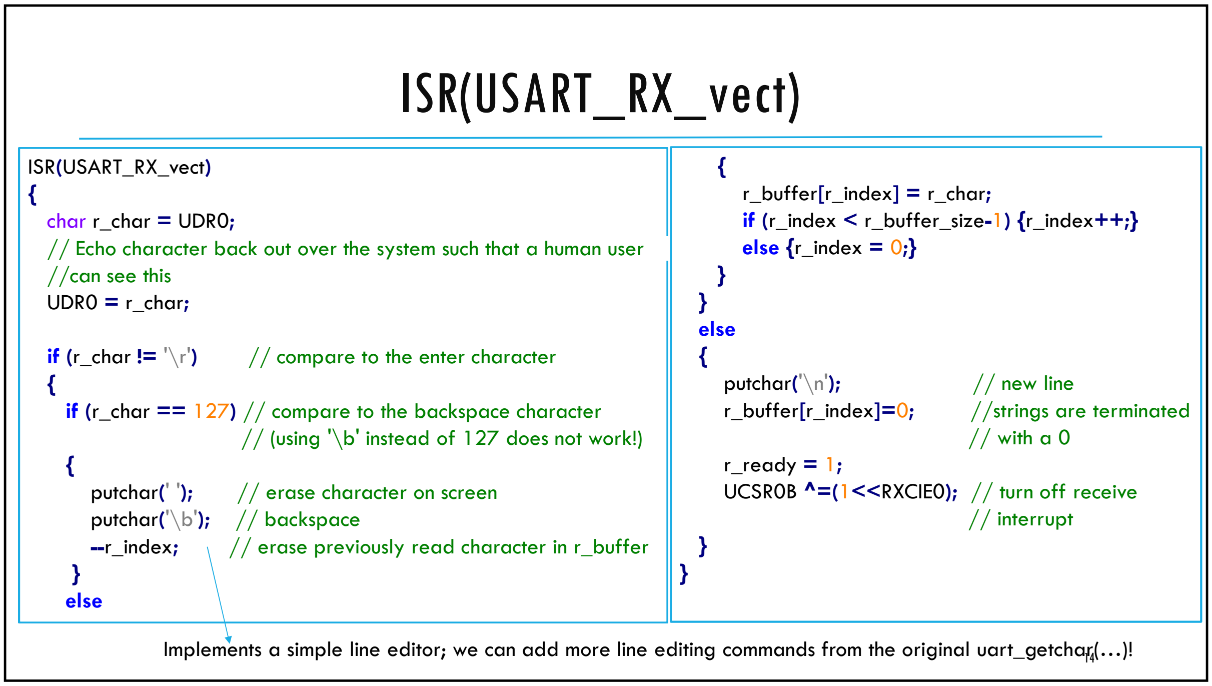


ISR(USART_RX_vect)

- The while loop represents task-based programming, which we repeat throughout the course: While a string is being inputted by the user, other tasks (e.g. Task2) can be executed in parallel
  - No stalling → Efficient execution
  - Modularity as a Computer System Design Principle
- The getstr() can be called in any subroutine, not only in the main while loop
- Inside getstr() r_ready and r_index are reset to 0 → Task_InterpretReadBuffer() may also call getstr() at the end of its code; Here we show the reset explicitly in the main while loop.
- It is often more natural to merge Task() and ResetCond(), especially if the reset should happen at the start of a task rather than at the end

```
int main(void)
{
  // Initializations etc.
  sei(); // Enable global interrupt
  getstr();
  etc.

  while(1)
  {
    if (r_ready == 1) {Task_InterpretReadBuffer(); getstr();}
    if Condition2 { Task2(…); ResetCond2; }
    etc.
  }

  return 0;
}
```

13

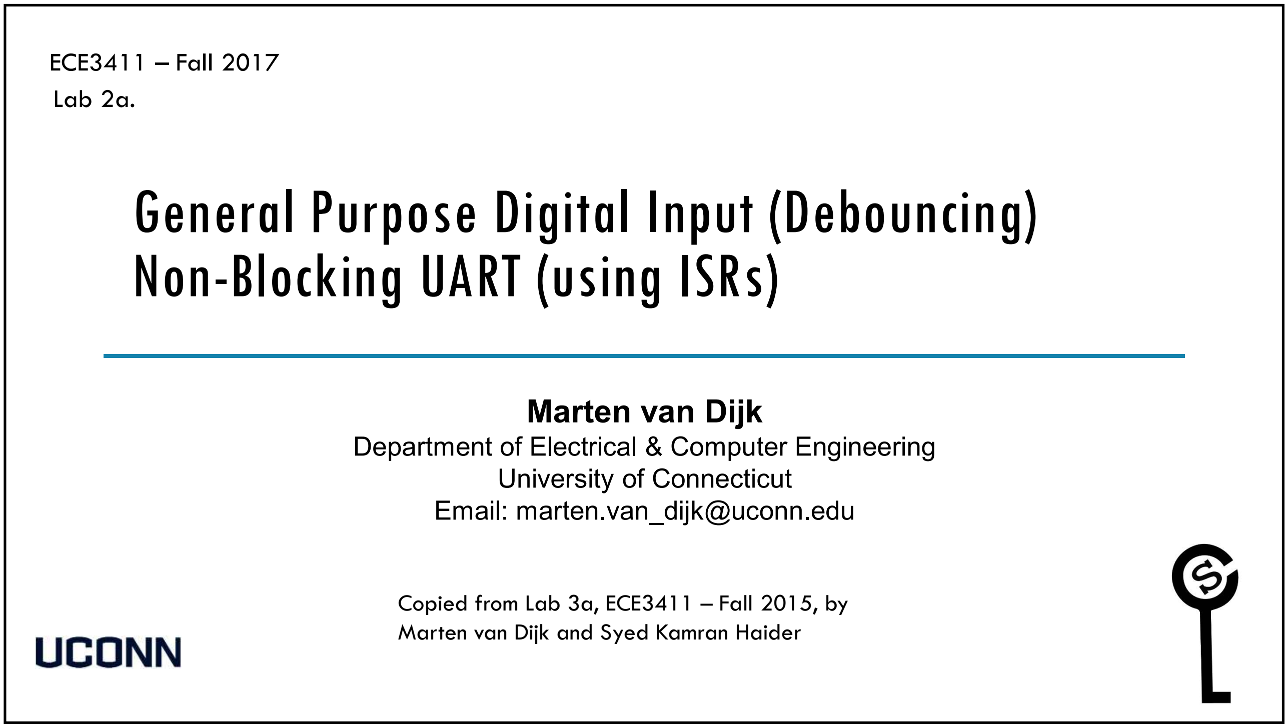


ISR(USART_RX_vect)

```
ISR(USART_RX_vect)
{
  char r_char = UDR0;
  // Echo character back out over the system such that a human user
  //can see this
  UDR0 = r_char;

  if (r_char != '\r')       // compare to the enter character
  {
    if (r_char == 127) // compare to the backspace character
                       // (using '\b' instead of 127 does not work!)
    {
       putchar(' ');      // erase character on screen
       putchar('\b');     // backspace
       --r_index;        // erase previously read character in r_buffer
    }
    else
      {
         r_buffer[r_index] = r_char;
         if (r_index < r_buffer_size-1) {r_index++;}
         else {r_index = 0;}
      }
    }
    else
    {
       putchar('\n');              // new line
       r_buffer[r_index]=0;        //strings are terminated
                                   // with a 0
       r_ready = 1;
       UCSR0B ^=(1<<RXCIE0);  // turn off receive
                              // interrupt
    }
}
```

Implements a simple line editor; we can add more line editing commands from the original uart_getchar(…)!

14

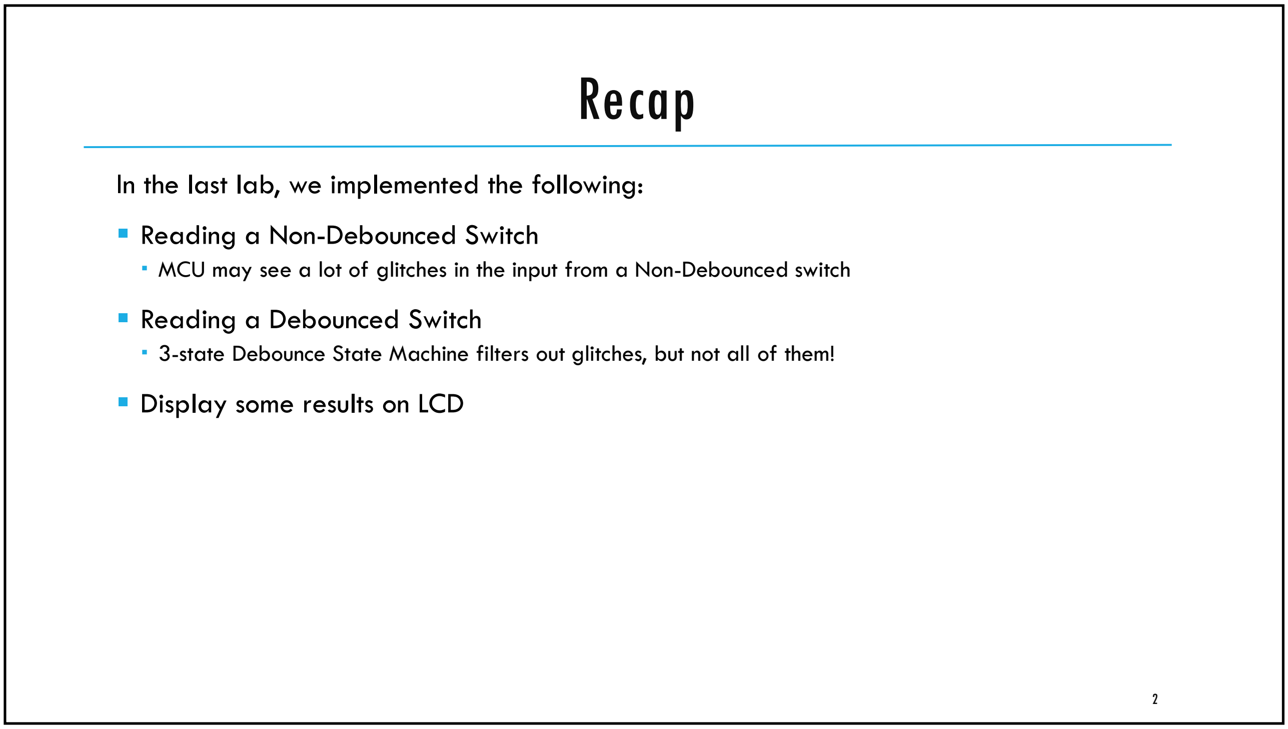


ECE3411 – Fall 2017
Lab 2a.

General Purpose Digital Input (Debouncing) Non-Blocking UART (using ISRs)

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

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

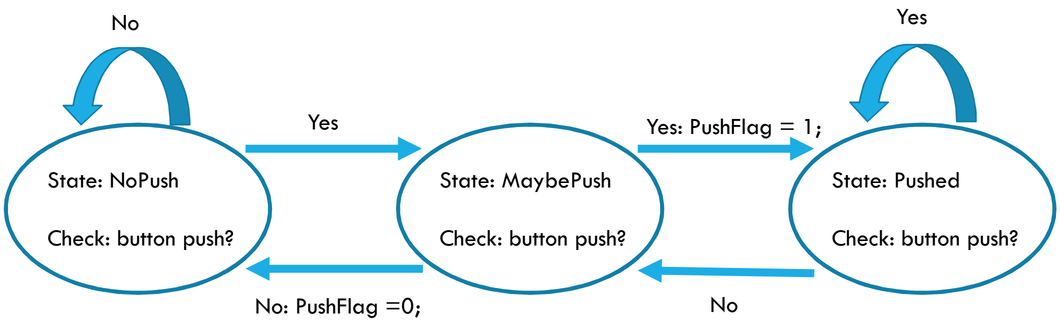


Recap

In the last lab, we implemented the following:

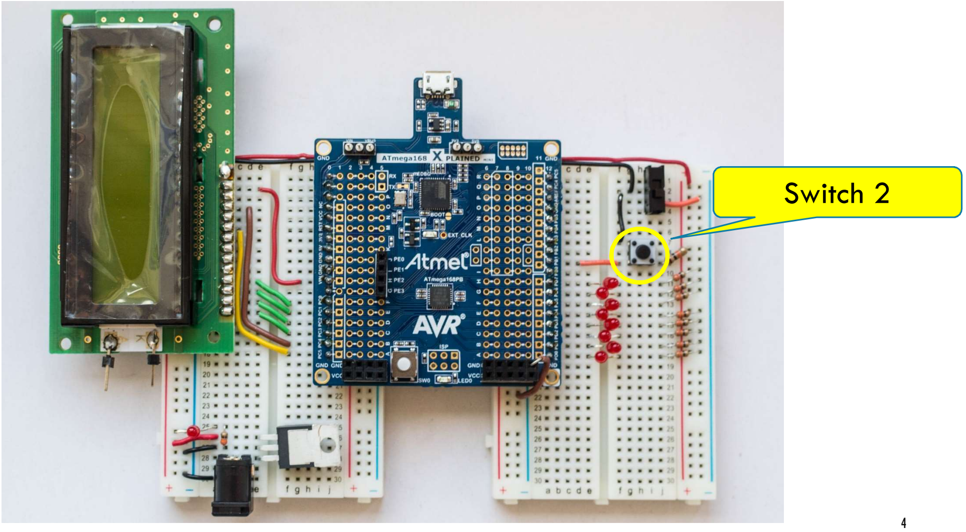
- Reading a Non-Debounced Switch
  - MCU may see a lot of glitches in the input from a Non-Debounced switch
- Reading a Debounced Switch
  - 3-state Debounce State Machine filters out glitches, but not all of them!
- Display some results on LCD

Do you see any problems with this Debounce State Machine?



- This state machine filters out glitches which result in **NoPush** → **MaybePush** → **NoPush** transitions
- What happens if a glitch causes **Pushed** → **MaybePush** → **Pushed** transitions sequence?
  - The software mistakenly thinks that a new button-push has occurred
  - Fix this problem in Task 1 of this lab

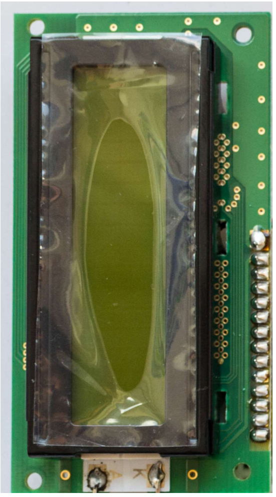
Push Switch to use



LCD Interfacing

- We are going to use the LCD in 4-bit mode
  - Only 4 data wires are required instead of 8
- LCD pin assignment is as follows:

No.	Symbol	Connections with ATmega328P
1, 3	V_{SS}, V_{EE}	GND
2	V_{CC}	5V
4	RS	PC4
5	R/W	GND (Always Write to LCD)
6	E	PC5
7-10	DB0-DB3	Not Connected
11-14	DB4-DB7	PC0-PC3



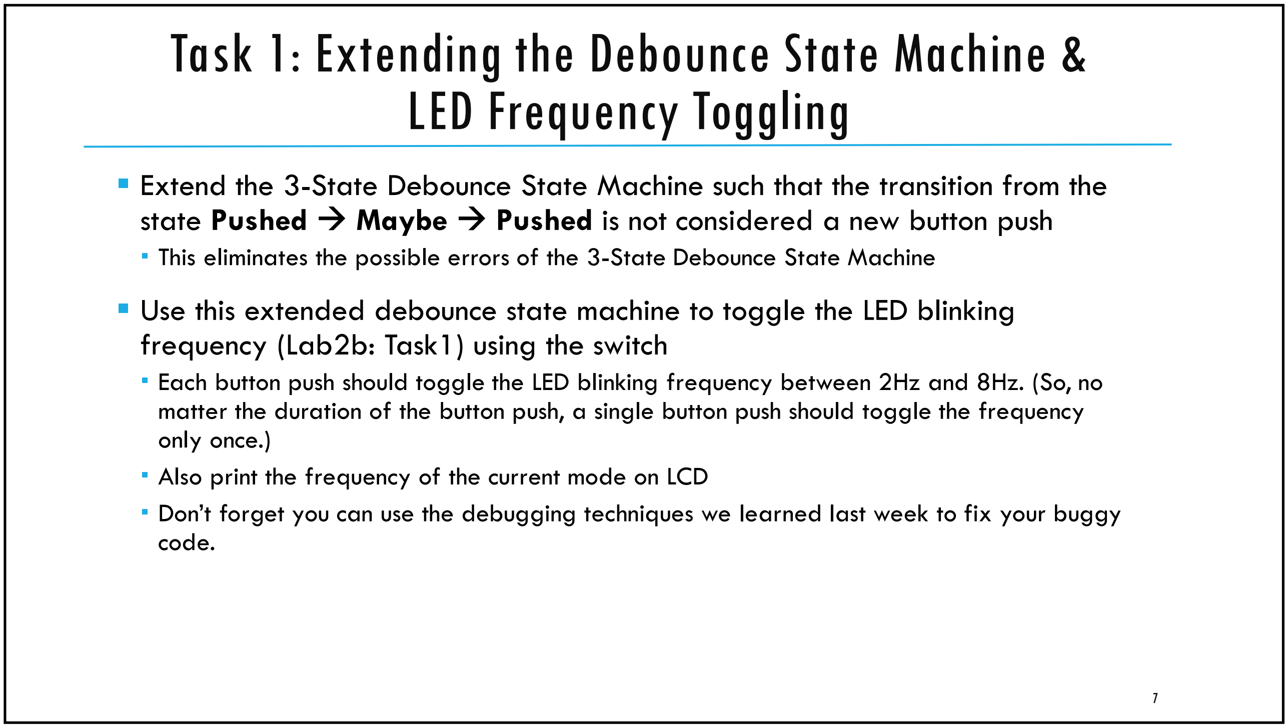
Pin1: V_{SS}	→ GND
Pin2: V_{CC}	→ 5V
Pin3: V_{EE}	→ GND
Pin4: RS	→ PC4
Pin5: R/W	→ GND
Pin6: E	→ PC5
Pin7: DB0	→ N/C
Pin8: DB1	→ N/C
Pin9: DB2	→ N/C
Pin10: DB3	→ N/C
Pin11: DB4	→ PC0
Pin12: DB5	→ PC1
Pin13: DB6	→ PC2
Pin14: DB7	→ PC3

Pin16: ANODE → 5V

Pin15: CATHODE → GND

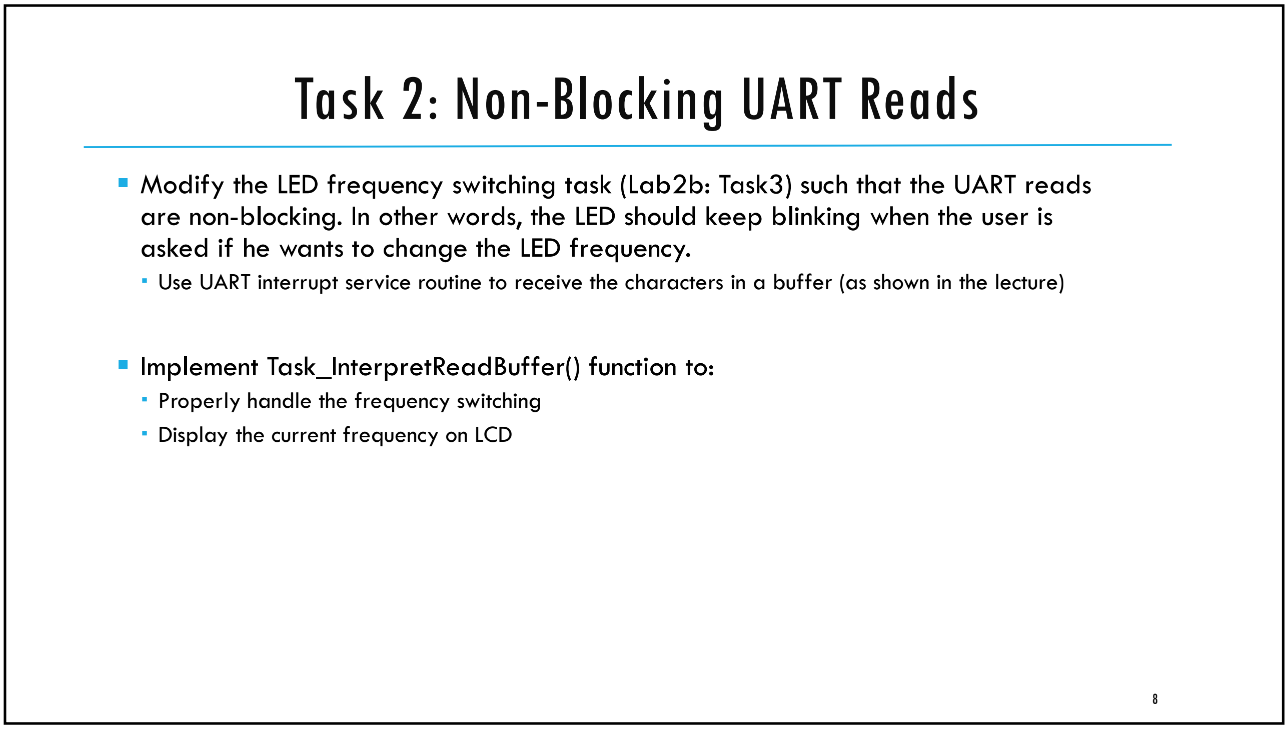
LCD Test Program

```
// ------- Preamble -------- //
#define F_CPU 16000000UL      /* Tells the Clock Freq to the Compiler. */
#include <avr/io.h>           /* Defines pins, ports etc. */
#include <util/delay.h>       /* Functions to waste time */
#include "lcd_lib.h"          /* LCD Library */

int main(void) {
  // -------- Inits --------- //
   initialize_LCD();           /* Initialize LCD */

   LcdDataWrite('A');         /* Print a few characters for test */
   LcdDataWrite('B');
   LcdDataWrite(‘C');

  // ------ Event loop ------ //
  while (1) {
        /* Nothing to do */
  } /* End event loop */
  return (0);
}
```



Task 1: Extending the Debounce State Machine & LED Frequency Toggling

- Extend the 3-State Debounce State Machine such that the transition from the state **Pushed → Maybe → Pushed** is not considered a new button push
  - This eliminates the possible errors of the 3-State Debounce State Machine
- Use this extended debounce state machine to toggle the LED blinking frequency (Lab2b: Task1) using the switch
  - Each button push should toggle the LED blinking frequency between 2Hz and 8Hz. (So, no matter the duration of the button push, a single button push should toggle the frequency only once.)
  - Also print the frequency of the current mode on LCD
  - Don't forget you can use the debugging techniques we learned last week to fix your buggy code.



Task 2: Non-Blocking UART Reads

- Modify the LED frequency switching task (Lab2b: Task3) such that the UART reads are non-blocking. In other words, the LED should keep blinking when the user is asked if he wants to change the LED frequency.
  - Use UART interrupt service routine to receive the characters in a buffer (as shown in the lecture)
- Implement Task_InterpretReadBuffer() function to:
  - Properly handle the frequency switching
  - Display the current frequency on LCD

ECE3411 – Fall 2017
Lecture 2b.

ISRs, Timer0
Task Based Programming

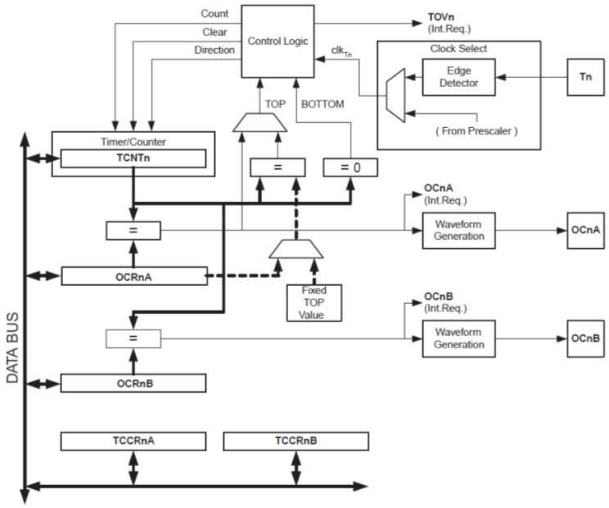
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

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

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

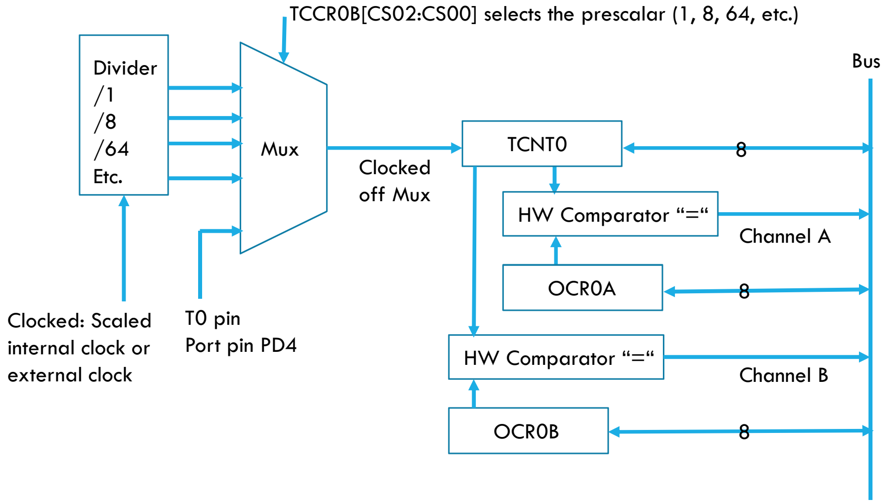
Timer 0

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



2

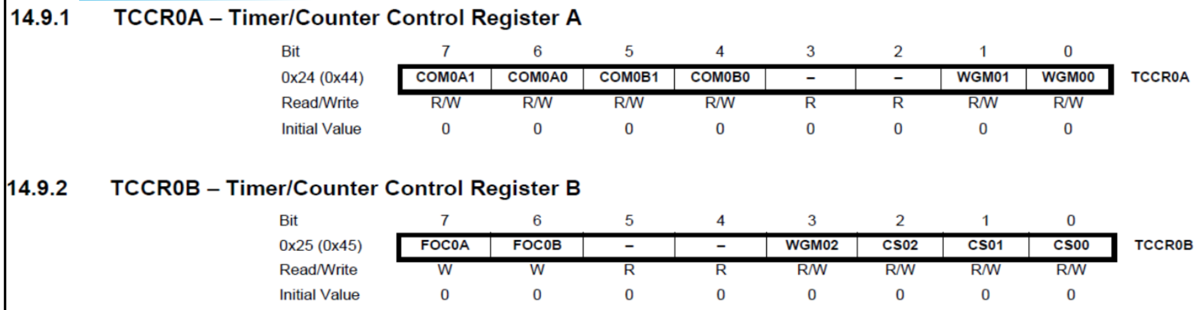
Timer 0



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

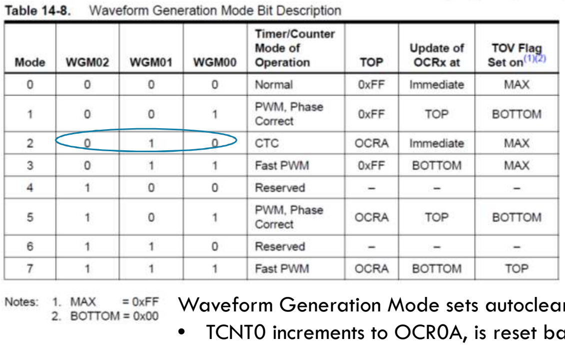


Table 14-8. Waveform Generation Mode Bit Description

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

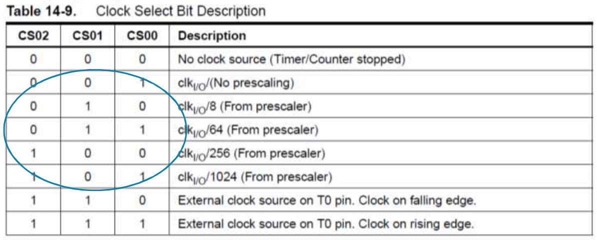


Table 14-9. Clock Select Bit Description

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);

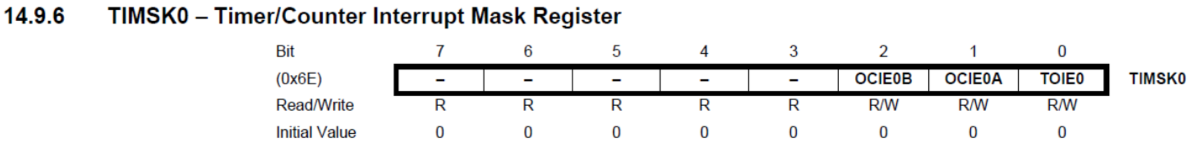
- TCNT0 increments to OCR0A, is reset back to 0, and starts incrementing again
- TCNT0 follows a sawtooth

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

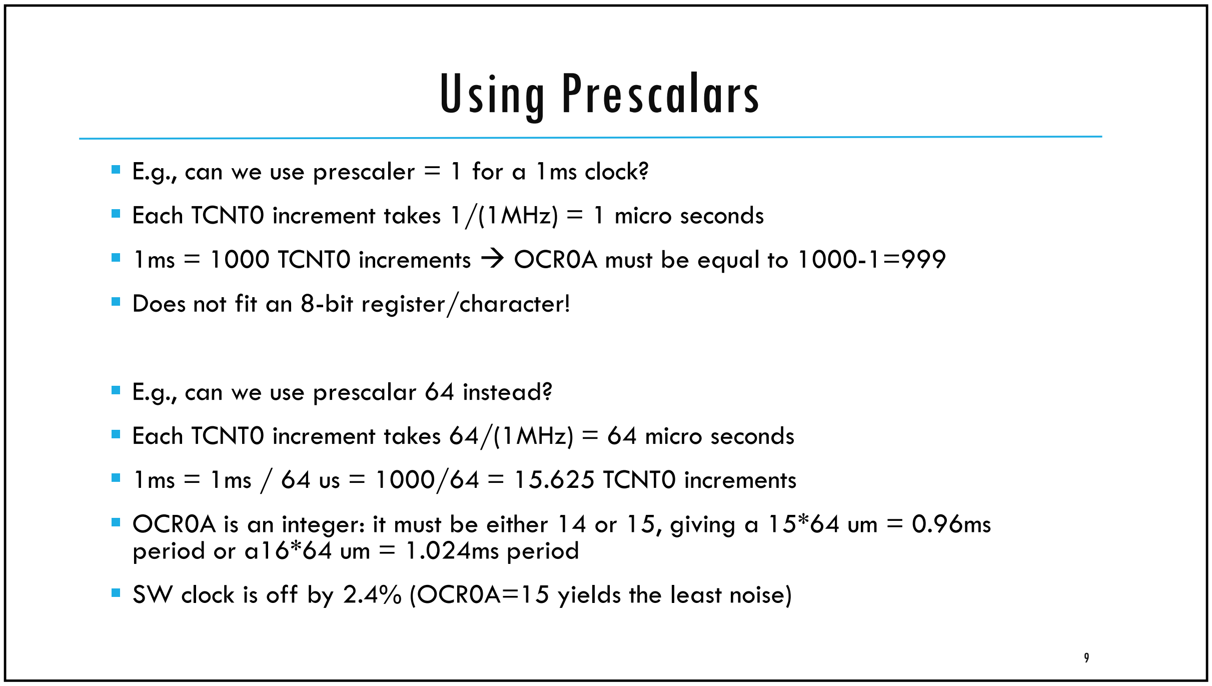
7

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;
}
```

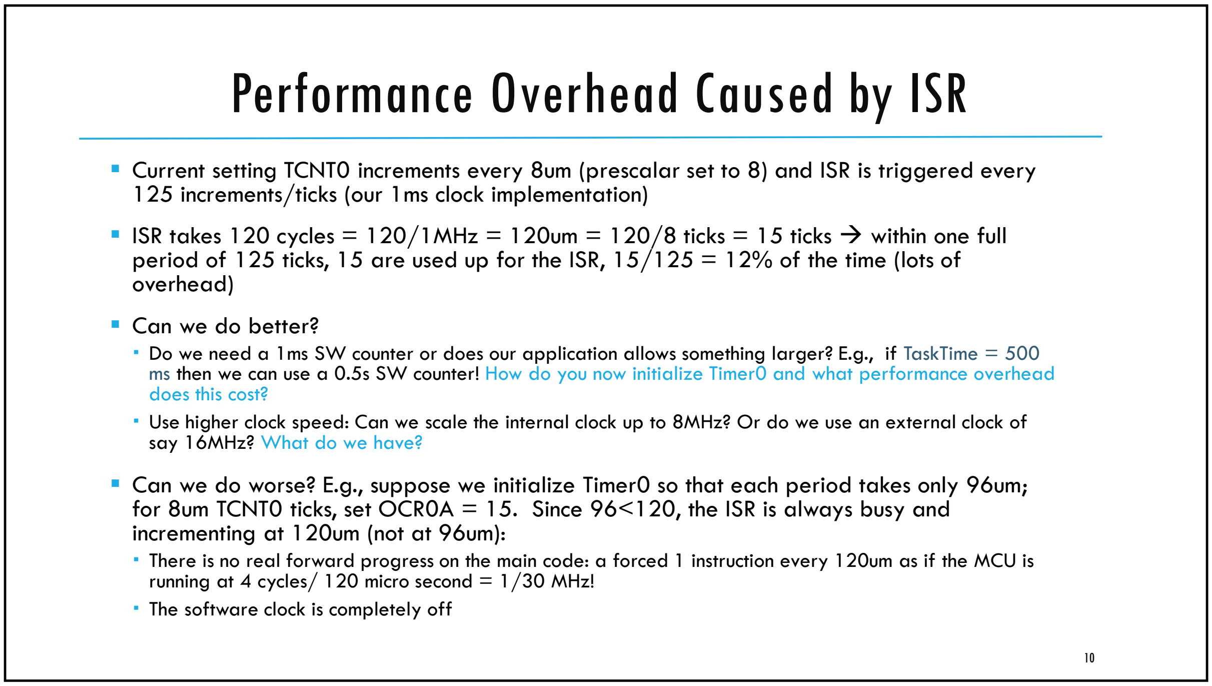
8



Using Prescalars

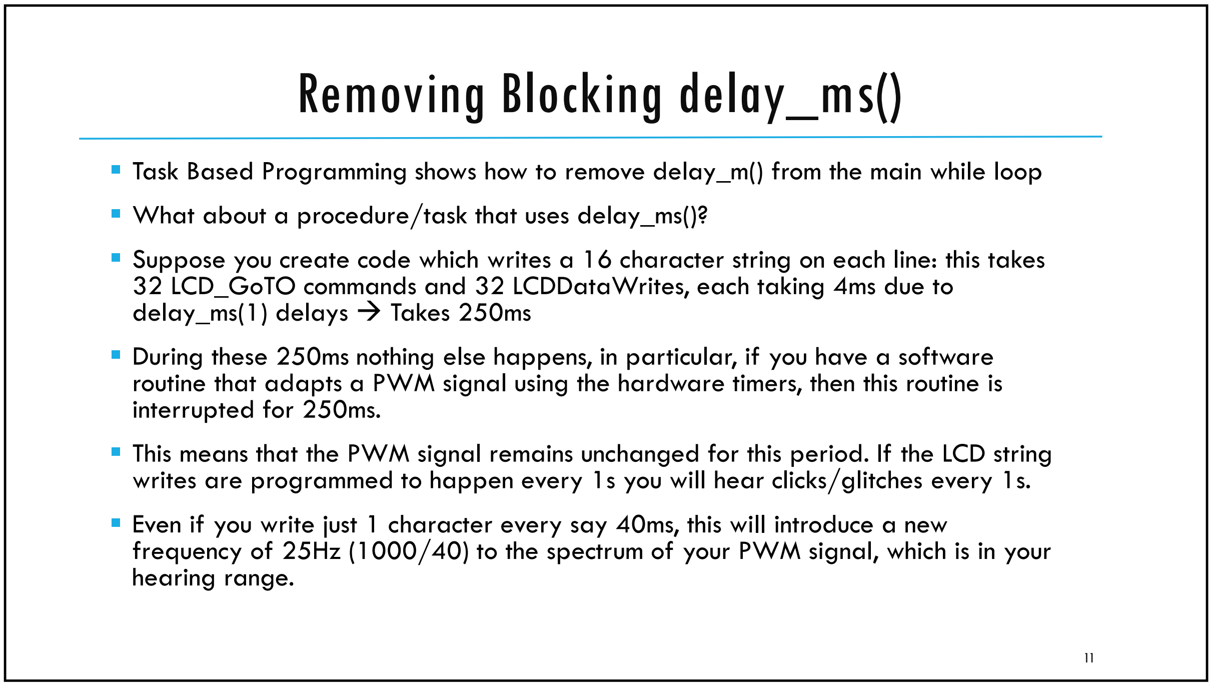
- E.g., can we use prescaler = 1 for a 1ms clock?
- Each TCNT0 increment takes 1/(1MHz) = 1 micro seconds
- 1ms = 1000 TCNT0 increments → OCR0A must be equal to 1000-1=999
- Does not fit an 8-bit register/character!

- 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

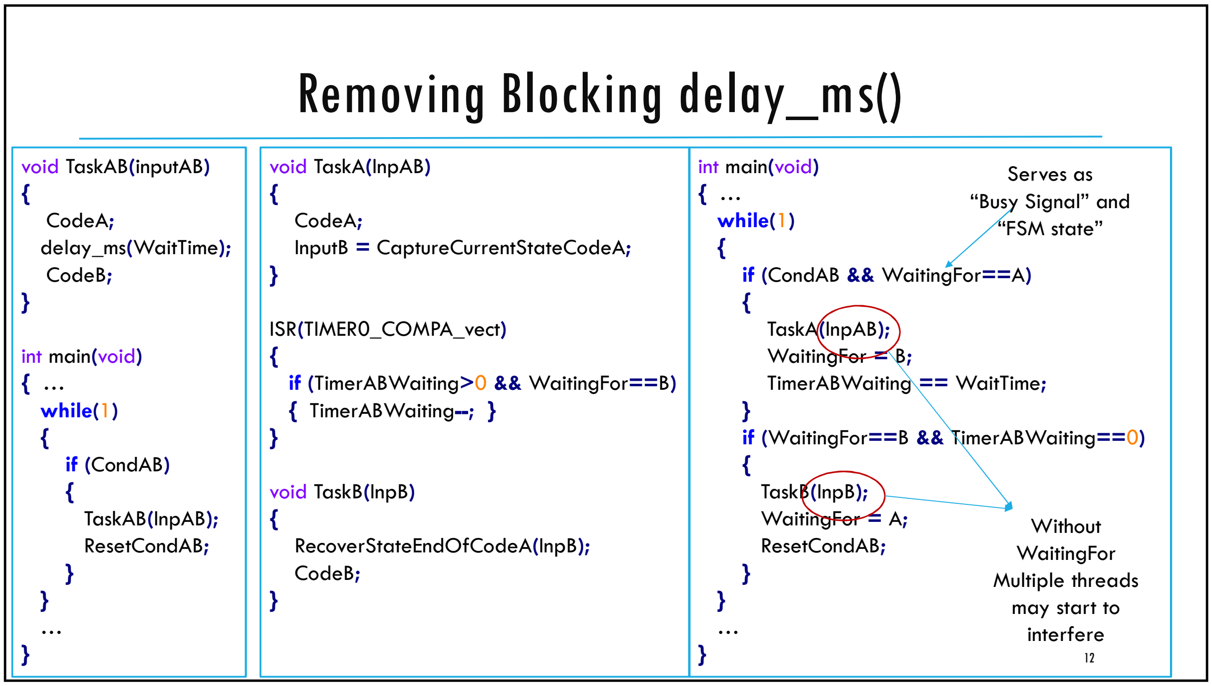
- Current setting TCNT0 increments every 8um (prescalar set to 8) and ISR is triggered every 125 increments/ticks (our 1ms clock implementation)
- ISR takes 120 cycles = 120/1MHz = 120um = 120/8 ticks = 15 ticks → within one full period of 125 ticks, 15 are used up for the ISR, 15/125 = 12% of the time (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.

11



Removing Blocking delay_ms()

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

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

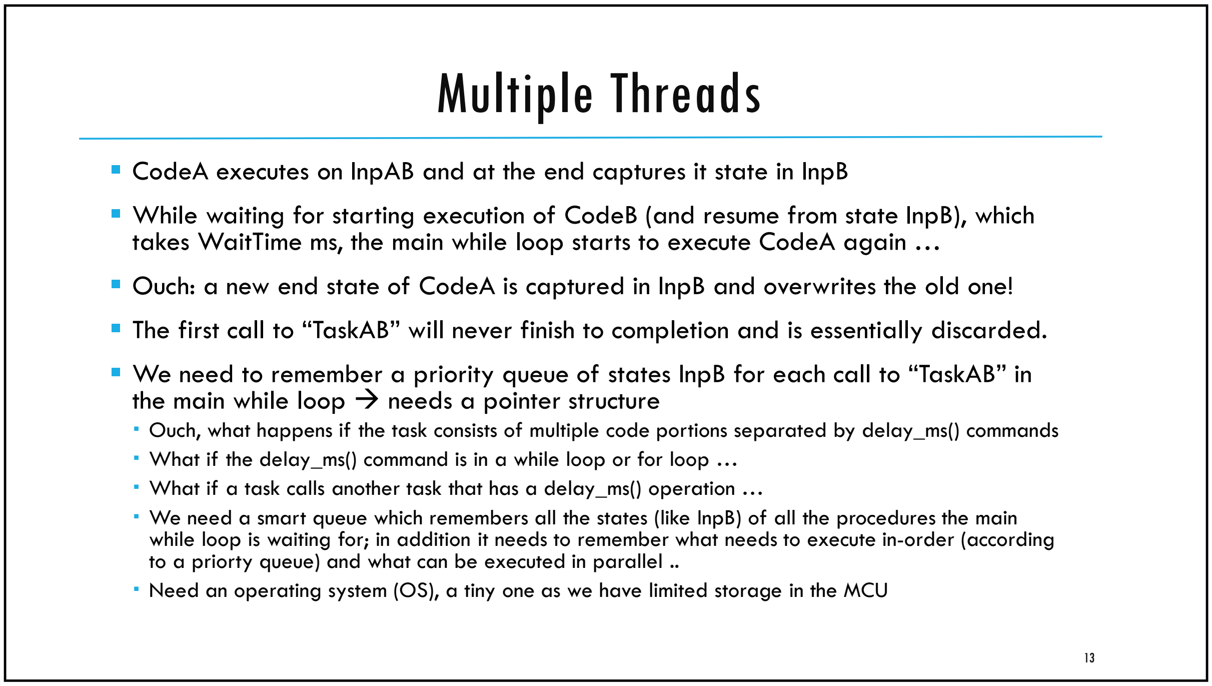
```
void TaskA(InpAB)
{
    CodeA;
    InputB = CaptureCurrentStateCodeA;
}

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

12



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

ECE3411 – Fall 2017
Lab2b

Non-Blocking LCD (using Timer ISR)

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

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

LCD functions so far!

- So far the LCD functions are blocking…
  - These functions use _delay_ms() or _delay_us() routines.
- We need to make use of the wasted CPU cycles
  - Hence we need to get rid of _delay_ms() & _delay_us() routines
- We use timer interrupts to trigger an event once certain time period has elapsed
  - Meanwhile do other work instead of waiting idle for the time to pass.

2

LCD Blocking Command Write Example

```
void LcdCommandWrite(uint8_t cm)
{
        // First send higher 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (cm >> 4);
        CTRL_PORT &= ~(1<<RS);
        CTRL_PORT |= (1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES
        CTRL_PORT &= ~(1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES

        // Send lower 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (cm & 0x0f);
        CTRL_PORT &= ~(1<<RS);
        CTRL_PORT |= (1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES
        CTRL_PORT &= ~(1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES
}
```

LCD Blocking Data Write Example

```
void LcdDataWrite(uint8_t da)
{
        // First send higher 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (da >> 4);
        CTRL_PORT |= (1<<RS);
        CTRL_PORT |= (1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES
        CTRL_PORT &= ~(1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES

        // Send lower 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (da & 0x0f);
        CTRL_PORT |= (1<<RS);
        CTRL_PORT |= (1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES
        CTRL_PORT &= ~(1<<ENABLE);
        _delay_ms(1);                    // WASTED CYCLES
}
```

Task 1: Debounce State Machine with Timer ISR

- Implement the LED Frequency Toggling task from Lab3a:Task1 **using the Timer Interrupts**, i.e.
  - Use the Extended Debounce State Machine to read a Push Switch.
  - On a button push, toggle the LED blinking rate between 2Hz & 8Hz.
  - You don't need to print anything on LCD or UART.
  - You are **NOT ALLOWED** to use _delay_ms() or _delay_us() functions.

Task2: Non-Blocking LCD Writes

Implement Non-Blocking LCD Writes using the Timer Interrupts and demonstrate LCD refresh rate of **exactly** 1Hz.

- In particular, implement the following:
  - Non-Blocking LcdDataWrite(uint8_t data) and LcdCommandWrite(uint8_t command) functions.
  - Print a different character on LCD after exactly 1 second to show a refresh rate of 1Hz, e.g. first print '0' then '1' after a second, and so on.
  - You are **NOT ALLOWED** to use _delay_ms() or _delay_us() functions.

ECE3411 – Fall 2017
Lecture 2c.

Timers 0, 1 & 2

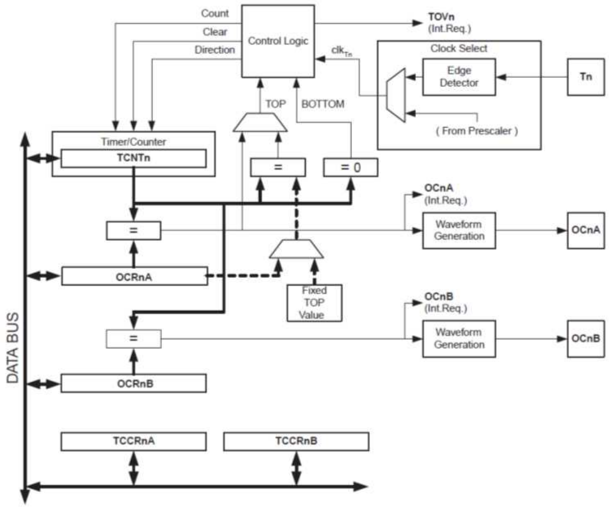
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

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

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

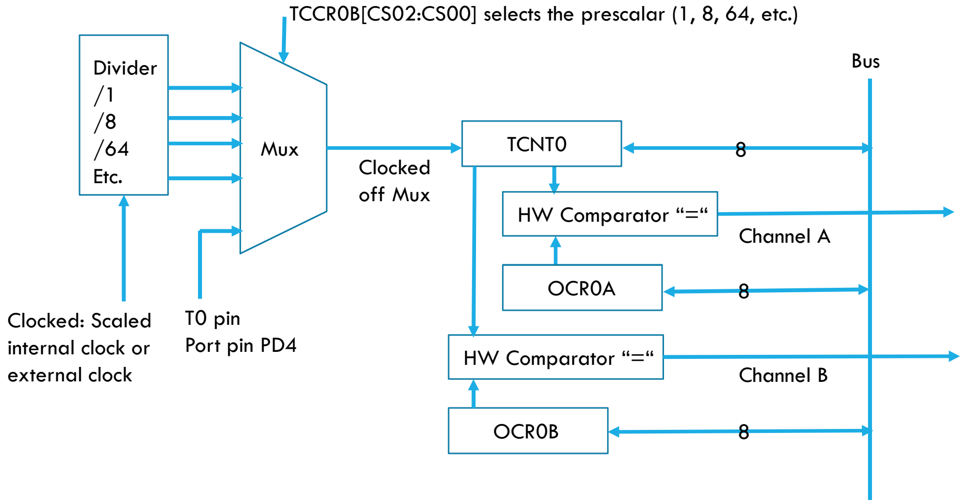
Timer 0 (Same as Timer 2)

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



2

Timer 0



TCCR0B[CS02:CS00] selects the prescalar (1, 8, 64, etc.)

Divider
/1
/8
/64
Etc.

Mux

Clocked off Mux

TCNT0

HW Comparator "="

OCR0A

HW Comparator "="

OCR0B

Bus

8

Channel A

8

Channel B

8

Clocked: Scaled internal clock or external clock

T0 pin
Port pin PD4

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); //Clear on Compare A
  OCR0A = 124; //Set number of ticks for Compare A
  TIMSK0 =2;  //Enable Timer 0 Compare A ISR
  TCCR0B = 2; //Set Prescalar & Timer 0 starts
}
....
```

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

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

  return 0;
}
```

Example Timer 0

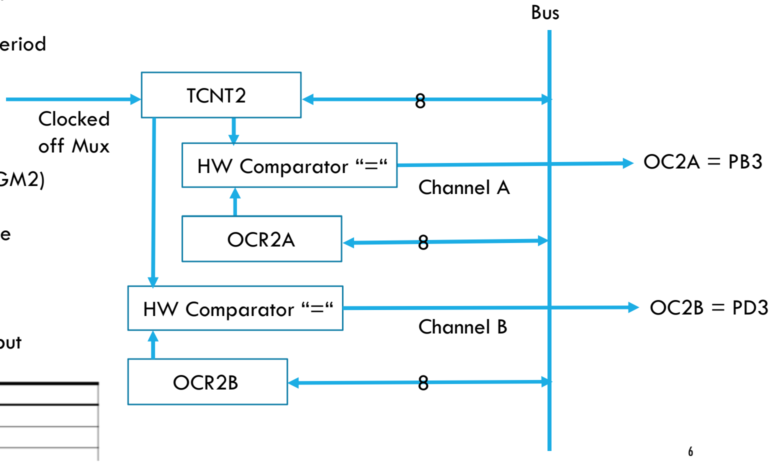
- 16MHz, 1ms ticks:

```
// 1ms ISR for Timer 0 assuming F_CPU = 16MHz
void InitTimer0(void)
{
  TCCR0A |= (1<<WGM01); //turn on clear-on-match with OCR0A
  OCR0A = 249;              //Set the compare register to 250 ticks
  TIMSK0 = (1<<OCIE0A);     //Enable Timer 0 Compare A ISR
  TCCR0B = 3;               // Set Prescalar to divide by 64 & Timer 0 starts
}
....
```

Timer 2

Create an autonomous 200 cycle long square wave at PB3:

- OCR2A = 99 sets a 100 cycle half period
- TCCR2B = 1 sets prescalar at 1, i.e., counting is done at full rate (F_CPU)
- TCCR2A= (1<<COM2A0) | (1<<WGM21);
    - See Timer 0 discussion: (1<<WGM2) gives us a clear on match
    - (1<<COM2A0) is new: it tells the MCU to use channel A, i.e., it connects the comparator to the output pin OC2A=PB3
- DDRB = (1<<PINB3) sets PB3 as output



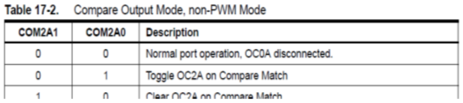


Table 17-2. Compare Output Mode, non-PWM Mode

COM2A1	COM2A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	Toggle OC2A on Compare Match
1	0	Clear OC2A on Compare Match

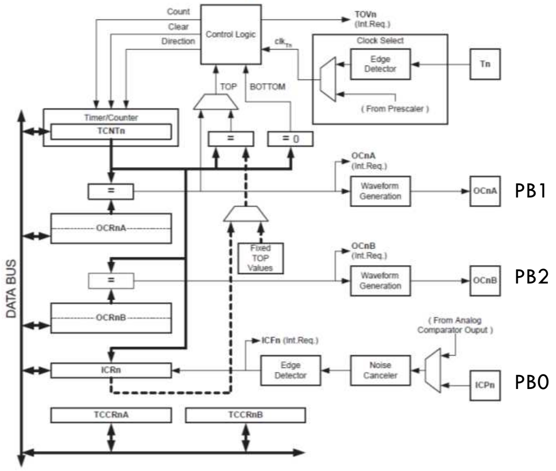
Example Timer 2

- 200 cycle square waveform at PB3 (a first example of PWM):

```
void InitTimer2(void)
{
  TCCR2A |= (1<<COM2A0) | (1<<WGM21); //turn on clear-on-match with OCR0A
                                      // transmit comparator result to pin OC2A
  OCR2A = 99;           //Set the compare register to 100 ticks, i.e., one half period
  TCCR2B = 1;           // Set Prescalar to divide by 1, i.e., full speed
  DDRB = (1<<PINB3); //Set OC2A = PB3 to output
}
....
```

Timer 1

Figure 15-1. 16-bit Timer/Counter Block Diagram[1]



(See pin assignments)

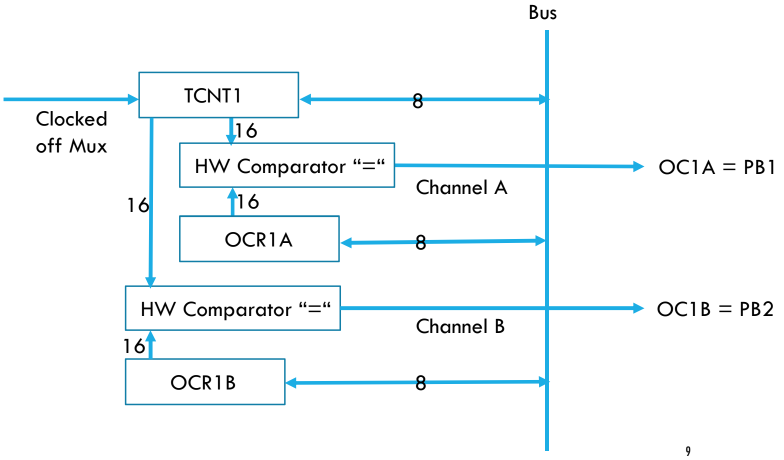
Note: 1. Refer to Figure 1-1 on page 2, Table 13-3 on page 82 and Table 13-9 on page 88 for Timer/Counter1 pin placement and description.

Timer 1

TCNT1 is 16 bits, a high and low register:

- Need 2 bus cycles to read TCNT1: it reads the lower byte and also copies the higher byte to a special location
- Stores the 8-bit high till you read it, if you do not read it, you will never read a value again
- If you read the high value first and then the low value, timer 1 is frozen
- Use the 16 bit reads built into C

Not drawn: TCNT1 can be captured by register ICR1 clocked of a mux …. see next slide



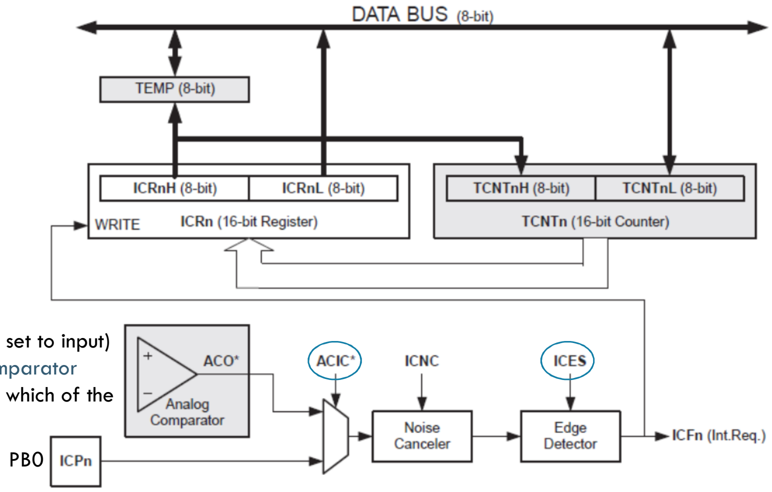
TCNT1 Can Be Captured by ICR1

Figure 15-3. Input Capture Unit Block Diagram

Edge change indicates when to capture.

- Caused by an edge change on PB0 (if set to input)
- Or by an edge change on Analog Comparator

Select bit ACIC in ACSR register indicates which of the two is chosen

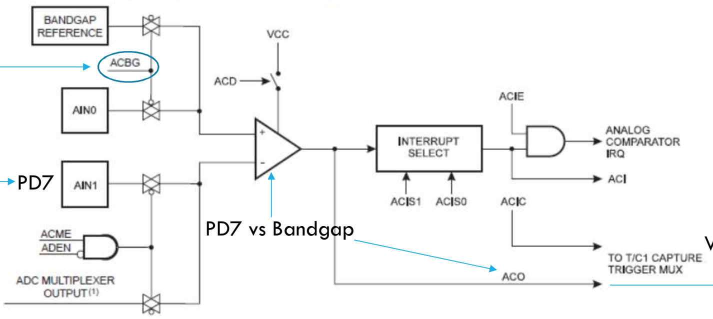


Register Description Timer 1

- Control register TCCR1A:
  - Positions 7&6 → COM1A (COM1A0 and COM1A1)
  - Positions 5&4 → COM1B (COM1B0 and COM1B1)
  - Positions 1&0 → WGM11 and WGM10 (waveform)
- Control register TCCR1B:
  - Positions 3&4 → WGM13 and WGM12 (waveform continued)
  - Positions 0,1,2 → Prescalar as before
  - Position ICES1=6 → Sets input capture edge select:
    - 1 = rising
    - 0 = falling
  - Position ICNC1=7 → Sets input capture noise canceler
    - This requires 2 measurements in a row making sure one transmission actually occurred
- Control register TIMSK1:
  - Positions 0,1,2 → As before
    - TOIE1 (timer overflow interrupt enable)
    - OCIE1A and OCIE1B (on compare and match interrupt enable)
  - Positions ICIE1=5 → Interrupt capture interrupt enable

Analog Comparator Output

Figure 22-1. Analog Comparator Block Diagram(2)



Select bandgap reference

Let's connect the Timer 2 square waveform (PB3)

PD7

PD7 vs Bandgap

Want to capture at rising edge ICES1set to 1; Since PB3 generates a waveform on a cycle by cycle basis, prescalar set to 1 for full speed

Notes: 1. See Table 22-1 on page 247.
2. Refer to Figure 1-1 on page 2 and Table 13-9 on page 88 for Analog Comparator pin placement.

PIN Assignment

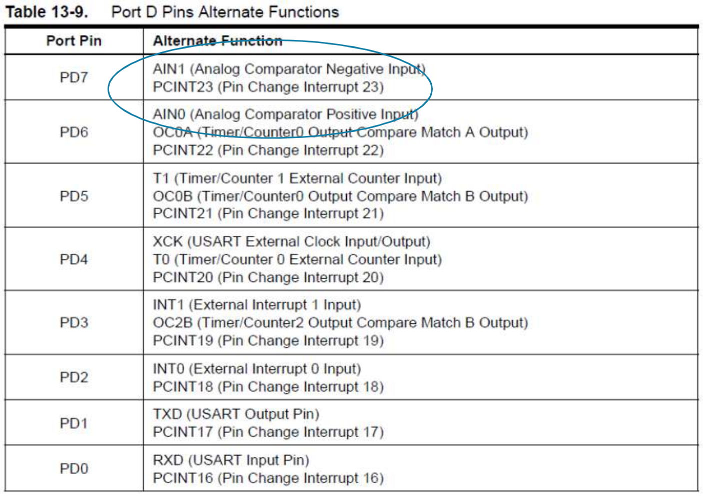
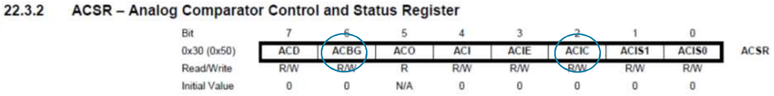


Table 13-9. Port D Pins Alternate Functions

Port Pin	Alternate Function
PD7	AIN1 (Analog Comparator Negative Input) PCINT23 (Pin Change Interrupt 23)
PD6	AIN0 (Analog Comparator Positive Input) OC0A (Timer/Counter0 Output Compare Match A Output) PCINT22 (Pin Change Interrupt 22)
PD5	T1 (Timer/Counter 1 External Counter Input) OC0B (Timer/Counter0 Output Compare Match B Output) PCINT21 (Pin Change Interrupt 21)
PD4	XCK (USART External Clock Input/Output) T0 (Timer/Counter 0 External Counter Input) PCINT20 (Pin Change Interrupt 20)
PD3	INT1 (External Interrupt 1 Input) OC2B (Timer/Counter2 Output Compare Match B Output) PCINT19 (Pin Change Interrupt 19)
PD2	INT0 (External Interrupt 0 Input) PCINT18 (Pin Change Interrupt 18)
PD1	TXD (USART Output Pin) PCINT17 (Pin Change Interrupt 17)
PD0	RXD (USART Input Pin) PCINT16 (Pin Change Interrupt 16)

13

Register Description Timer 1

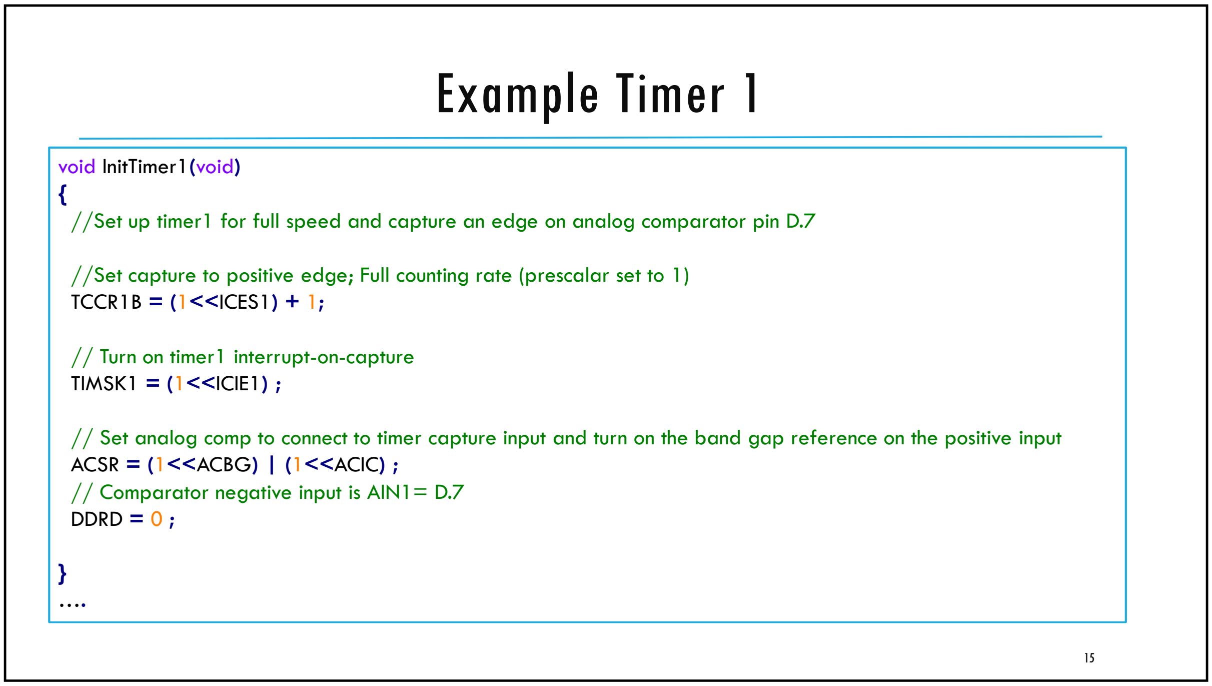


22.3.2 ACSR – Analog Comparator Control and Status Register

Bit	7	6	5	4	3	2	1	0	
0x30 (0x50)	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	

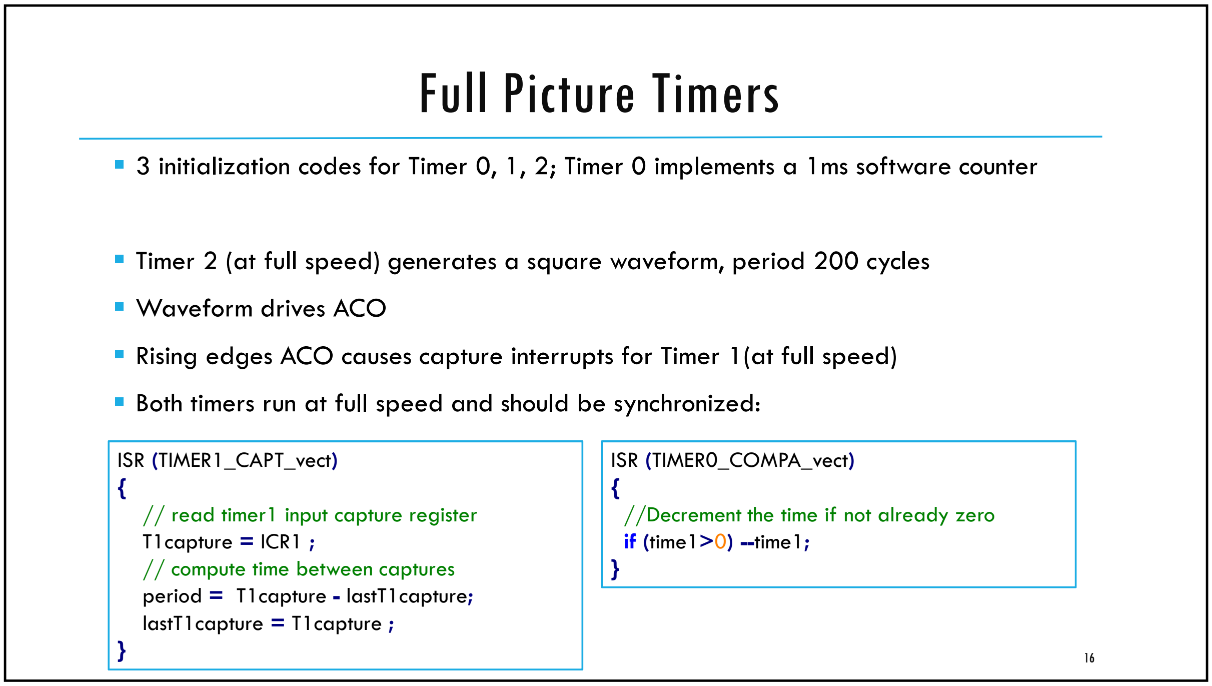
- Let's program a capture interrupt using the analog comparator
- Need to set register ACSR:
  - Positions 0&1 → Interrupt on toggle rise or fall
  - Positions 2&3 → Capture and Comparator interrupt enable
    - We want to enable the capture interrupt (not the comparator interrupt)
  - Position 4 → Comparator interrupt flag:
    - Is set when this interrupt happens, and
    - clears when a corresponding ISR executes the final (atomic) RETI instruction
  - Position 5 → Records ACO raw comparator output:
    - in real time nanosec by nanosec
    - digital output of the analog comparator (signal ACO)
  - Position 6 → Connects positive input to a bandgap reference (a temperature independent voltage reference circuit)
    - If 1, then (see description datasheet) fixed bandgap reference is used as input to the analog comparator (usually do not want this)
  - Position 7 → When switched to 1 the analog comparator is turned off

14



Example Timer 1

```
void InitTimer1(void)
{
  //Set up timer1 for full speed and capture an edge on analog comparator pin D.7

  //Set capture to positive edge; Full counting rate (prescalar set to 1)
  TCCR1B = (1<<ICES1) + 1;

  // Turn on timer1 interrupt-on-capture
  TIMSK1 = (1<<ICIE1) ;

  // Set analog comp to connect to timer capture input and turn on the band gap reference on the positive input
  ACSR = (1<<ACBG) | (1<<ACIC) ;
  // Comparator negative input is AIN1= D.7
  DDRD = 0 ;

}
....
```

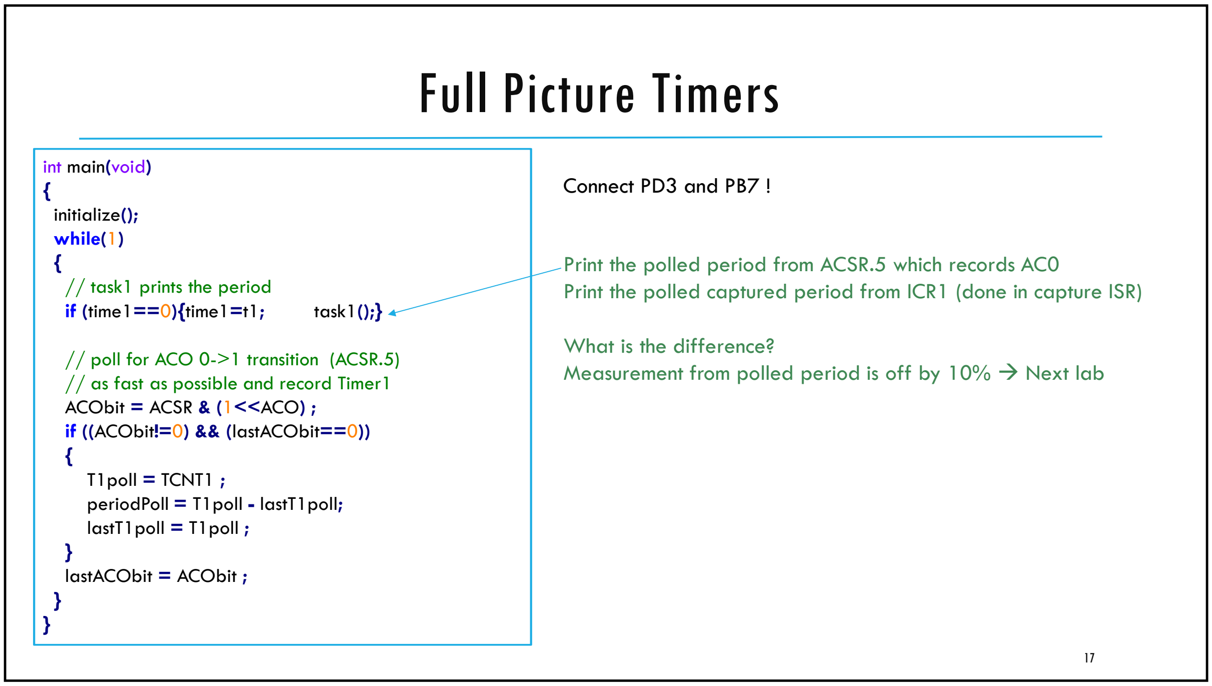


Full Picture Timers

- 3 initialization codes for Timer 0, 1, 2; Timer 0 implements a 1ms software counter
- Timer 2 (at full speed) generates a square waveform, period 200 cycles
- Waveform drives ACO
- Rising edges ACO causes capture interrupts for Timer 1(at full speed)
- Both timers run at full speed and should be synchronized:

```
ISR (TIMER1_CAPT_vect)
{
    // read timer1 input capture register
    T1capture = ICR1 ;
    // compute time between captures
    period =  T1capture - lastT1capture;
    lastT1capture = T1capture ;
}
```

```
ISR (TIMER0_COMPA_vect)
{
  //Decrement the time if not already zero
  if (time1>0) --time1;
}
```



Full Picture Timers

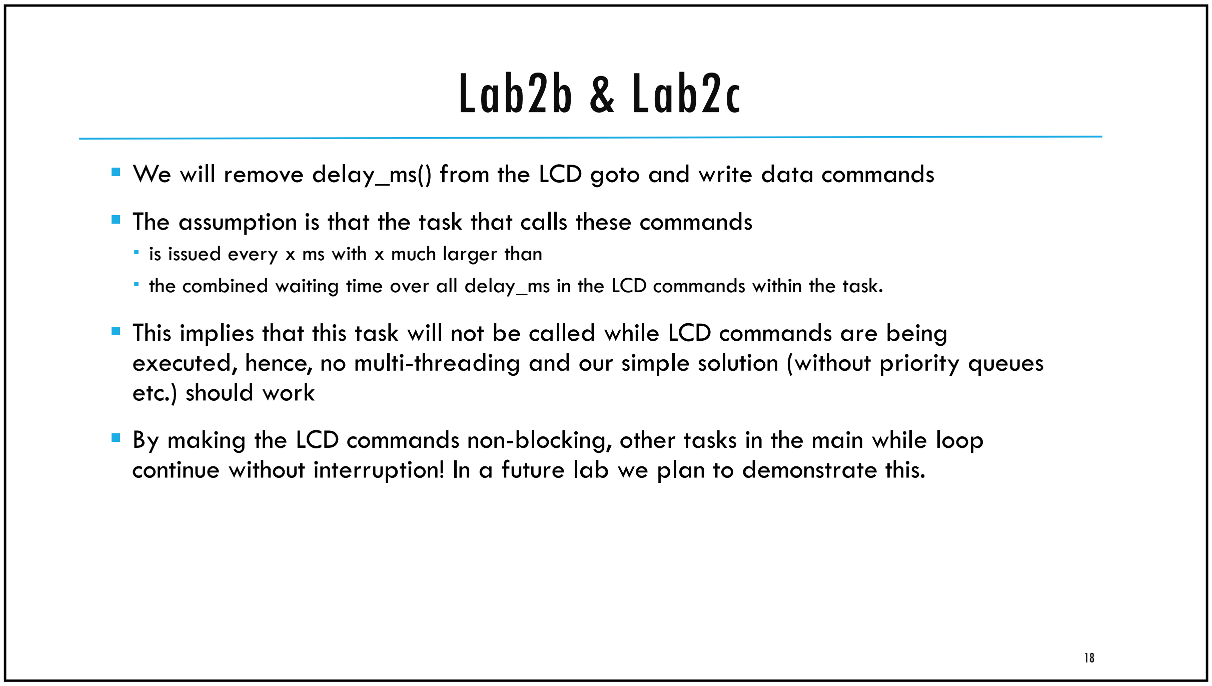
```
int main(void)
{
 initialize();
 while(1)
 {
   // task1 prints the period
   if (time1==0){time1=t1;        task1();}

   // poll for ACO 0->1 transition  (ACSR.5)
   // as fast as possible and record Timer1
   ACObit = ACSR & (1<<ACO) ;
   if ((ACObit!=0) && (lastACObit==0))
   {
      T1poll = TCNT1 ;
      periodPoll = T1poll - lastT1poll;
      lastT1poll = T1poll ;
   }
   lastACObit = ACObit ;
 }
}
```

Connect PD3 and PB7 !

Print the polled period from ACSR.5 which records ACO
Print the polled captured period from ICR1 (done in capture ISR)

What is the difference?
Measurement from polled period is off by 10% → Next lab

17



Lab2b & Lab2c

- We will remove delay_ms() from the LCD goto and write data commands
- The assumption is that the task that calls these commands
  - is issued every x ms with x much larger than
  - the combined waiting time over all delay_ms in the LCD commands within the task.
- This implies that this task will not be called while LCD commands are being executed, hence, no multi-threading and our simple solution (without priority queues etc.) should work
- By making the LCD commands non-blocking, other tasks in the main while loop continue without interruption! In a future lab we plan to demonstrate this.

18

ECE3411 – Fall 2017
Lab 2c.

Non-Blocking LCD (Extended State Machine)

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

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

UCONN

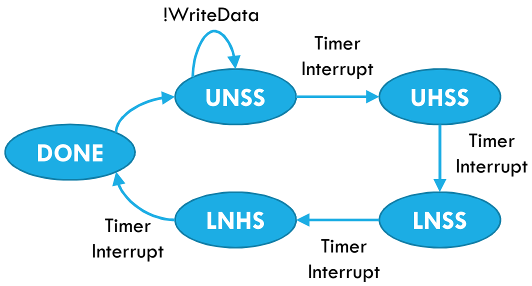
LCD Blocking Data Write Example

```
void LcdDataWrite(uint8_t da)
{
        // First send higher 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (da >> 4);
        CTRL_PORT |= (1<<RS);
        CTRL_PORT |= (1<<ENABLE);
        _delay_ms(1);                   // WASTED CYCLES
        CTRL_PORT &= ~(1<<ENABLE);
        _delay_ms(1);                   // WASTED CYCLES

        // Send lower 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (da & 0x0f);
        CTRL_PORT |= (1<<RS);
        CTRL_PORT |= (1<<ENABLE);
        _delay_ms(1);                   // WASTED CYCLES
        CTRL_PORT &= ~(1<<ENABLE);
        _delay_ms(1);                   // WASTED CYCLES
}
```

LCD Non-Blocking Write

In last lab, we implemented the following state machine:

- Split the Blocking-write into 4 states
  - Upper Nibble Setup State (UNSS)
  - Upper Nibble Hold State (UNHS)
  - Lower Nibble Setup State (LNSS)
  - Lower Nibble Hold State (LNHS)
- Transition to next state upon timer interrupt
  - Meanwhile do something else.

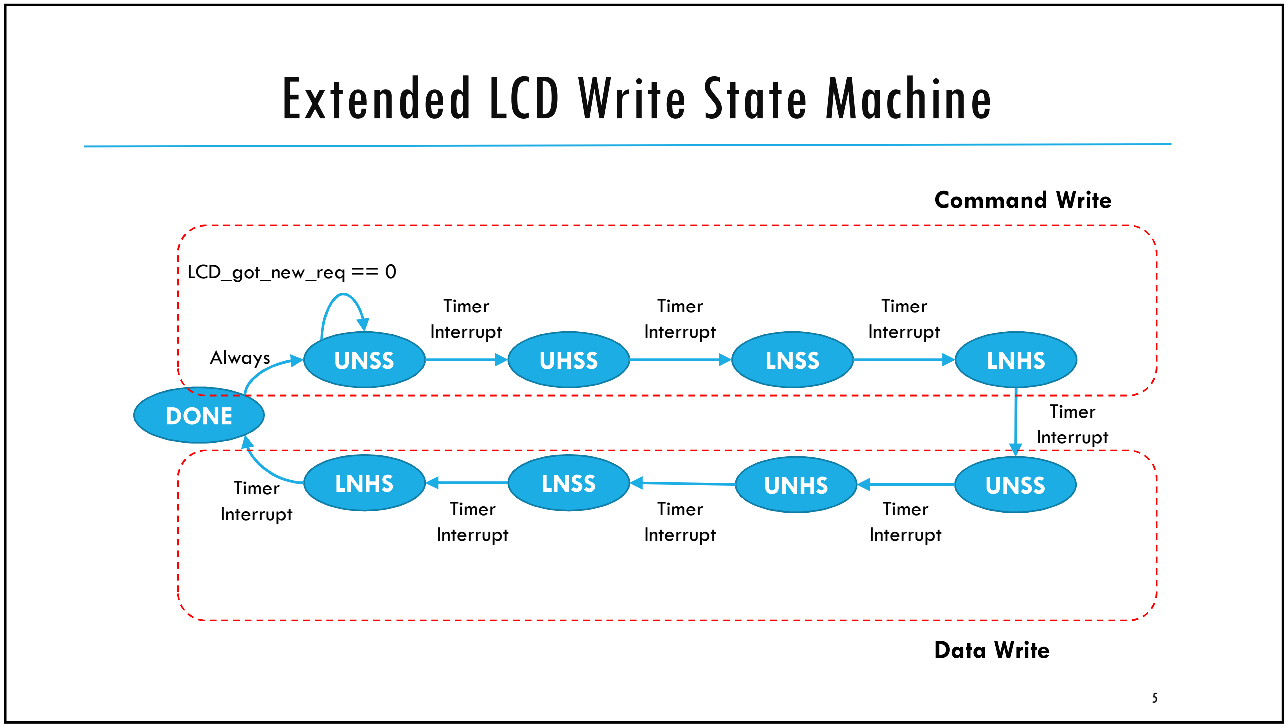


Task 1: Non-Blocking LCD Command+Data Write

Implement Non-Blocking LCD Writes using the Timer Interrupts and demonstrate LCD refresh rate of **exactly** 1Hz.

In particular, implement the following:

- Implement a non-blocking ***LCD_GoTo_and_Write(int x, int y, char data)*** function that writes a LCD GoTo command and a data to the resulting location back to back in a truly non-blocking fashion.
  - Implement a large state machine that can handle two back to back LCD write operations.
- Print a different character on LCD after exactly 1 second to show a refresh rate of 1Hz, e.g. first print '0' then '1' after a second, and so on.
- You are **NOT ALLOWED** to use _delay_ms() or _delay_us() functions.



Extended LCD Write State Machine

5

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set P2

There are 4 questions in this problem set. Answer each question according to the instructions given in at least 3 sentences on own words.

If you find a question ambiguous, be sure to write down any assumptions you make.
Be neat and legible. If we can't understand your answer, we can't give you credit!

Any form of communication with other students is considered cheating and will merit an F as final grade in the course.

SUBMIT YOUR ANSWERS IN A HARDCOPY FORMAT

Do not write in the box below

1 (x/35)	2 (x/40)	3 (x/15)	4 (x/10)	Total (xx/100)

Name:

Student ID:

1. [35 points]: Answer the following questions related to ISR:

a. Once an interrupt occurs, how does an AVR know where to find the code for the corresponding Interrupt Service Routine(ISR)?

b. Is the following statement True or False? ”Upon an interrupt, the instruction which is currently being executed in the main code is finished first before executing the Interrupt Service Routine.”

c. The following code shows a typical polling based system.

```
int main(void){
   // Event Loop
   while(1){
      if (Button1_Pressed())
      {
       Task_1();
       if (Button2_Pressed()) Task_2();
      }
   }
}
```

Which statement is correct about this system?

(a) Task_1() has higher priority than Task_2().

(b) Task_2() has higher priority than Task_1().

(c) Both Task_1() and Task_2() have the same priority.

(d) None of the above

Hint: Task_1() is said to have higher priority than Task_2() if, while in the middle of executing Task_2(), the AVR is ready to stop executing Task_2() and execute Task_1() immediately if it needs to react to a change coming in from the outside world.

d. Which one of the following systems may potentially waste and/or inefficiently utilize the useful CPU cycles?

(a) Interrupt Based System
(b) Polling Based System
(c) Both (a) and (b)
(d) None of the above

e. What is the return value of `ISR()` function?

f. Can a **user defined** variable be passed to an `ISR()`? If not, how can a variable be made accessible inside an `ISR()`?

g. Suppose that you do not press the on-board button, which generates a logic HIGH at PORTB7. What is the value of variable "a"?

```
#define PINB7 7
uint8_t a;
a = PINB7;
```

2. [40 points]: Answer the following questions related to Timers:

a. In ‘Normal Mode’, when does the 8-bit Timer/Counter `Timer0` overflow?

(a) When `TCNT0` matches with `OCR0A`

(b) When `TCNT0` matches with `OCR0B`

(c) When `TCNT0` = 255

(d) None of the above

b. In ‘Clear Timer on Compare Match’ (CTC) mode, `Timer0` resets itself automatically when it reaches the value that is stored in the register:

(a) `OCR0A`

(b) `TCCR0A`

(c) `TIMSK0`

(d) None of the above

c. Assume an MCU with crystal clock frequency 16MHz with `Timer0` initialized as follows:

```
/* Normal mode (default), just counting */
TCCR0B |= 0x01;  /* Clock Pre-scaler @ 1 */
```

At what rate, the register `TCNT0` is incremented?

d. For `Timer 0`, which register actually serves as a counter and stores the ticks-count?

(a) `TCNT0`

(b) `OCR0A`

(c) `OCR0B`

(d) None of the above

e. For `Timer 0` running in ‘Clear Timer on Compare Match’ (CTC) mode, the values of which two registers are compared with each other to determine a ‘Compare Match’?

(a) `OCR0B` and `TCCR0A`

(b) `OCR0A` and `TCNT0`

(c) `TIMSK0` and `TCNT0`

(d) None of the above

f. Assume an MCU with crystal clock frequency 16MHz with `Timer0` initialized as follows:

```
TIMSK0 = 2;    // Enable Timer0 Compare Match interrupt
TCCR0A = 0x02; // Clear Timer on Compare Match (CTC) mode
TCCR0B = 0x03; // Prescalar @ 64 hence Timer0 increments every 4 microseconds
OCR0A  = X;    // Value that controls the rate of ‘Compare Match’ interrupt
```

Calculate the value of `X` that should be loaded into `OCR0A` register in order to generate the Compare Match interrupt after every $1ms$.

`X =`

g. Which timer register chooses the type of timer-based interrupt vector?

(a) TCCRnA

(b) TCCRnB

(c) TIMSK

(d) OCRnA/OCRnB

h. The figure below shows Input Capture Unit block diagram for Timer 1.

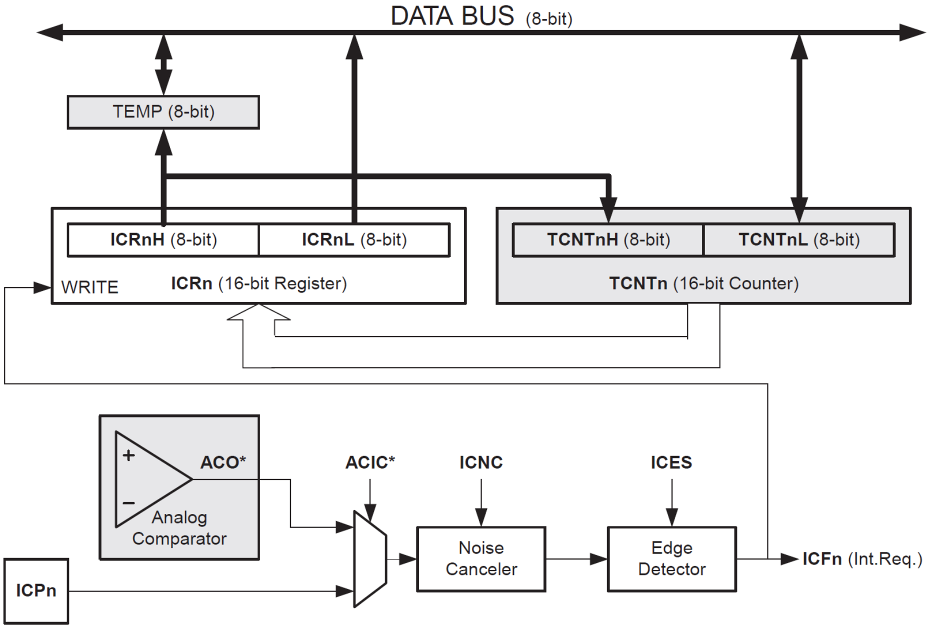


Figure 1: Input Capture Unit Block Diagram.

List the two sources (shown in the block diagram) that can be configured to generate an “Input Capture Interrupt”.

Initials:

3. [15 points]: The figure below shows the state diagram of a simple Finite State Machine (FSM). The FSM has four states and an input called `Flag`. Complete the `switch` statement given below to implement these state transitions.

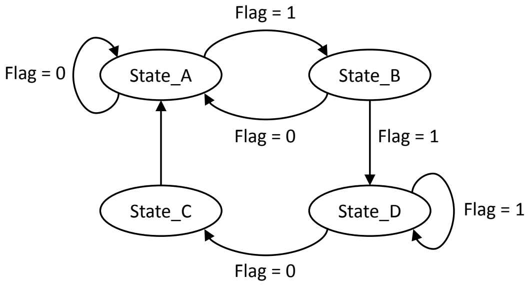


Figure 2: A Finite State Machine.

```
/* FSM Implementation */
switch (StopWatch_State)
{
    case State_A:





    break;
    case State_B:





    break;
    case State_C:





    break;
    case State_D:




    break;
}
```

Initials:

4. [10 points]: Can you shortly describe what you have learned and feel confident about using in the future?

Initials:

End of Problem Set

Please double check that you wrote your name on the front of the quiz.

Initials:

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set A2

There are 4 questions in this problem set. Answer each question according to the instructions given in at least 3 sentences on own words.

If you find a question ambiguous, be sure to write down any assumptions you make.
Be neat and legible. If we can't understand your answer, we can't give you credit!

Any form of communication with other students is considered cheating and will merit an F as final grade in the course.

SUBMIT YOUR ANSWERS IN A HARDCOPY FORMAT.

Do not write in the box below

1 (x/20)	2 (x/30)	3 (x/30)	4 (x/20)	Total (xx/100)

Name:

Student ID:

1. [20 points]: Let `Task1()` and `Task2()` be two functions from standard C library (`stdlib.h`). Write a C program for your AVR such that it calls `Task1()` every $10ms$ and `Task2()` every $100ms$. Use of `_delay_ms()` functionality is allowed in this question. Assume that the execution of `Task1()` and `Task2()` virtually takes no time.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <util/delay.h>

/* Declare any variables here */






int main(void)
{
    /* Write your code below */















} /* End of main() */
```

Initials:

2. [30 points]: The code given below uses Timer 1 'Compare Match A' ISR to blink a LED connected to PB5. If the clock frequency ($clk_{I/O}$) is 16MHz, complete the "`initialize_all()`" function below such that the LED toggles after every 250 milliseconds.

```
/* Initialization function */
void initialize_all(void)
{
    // Set the LED pin as Output here


    // Configure Timer 1 here.













     // Enable Global Interrupts here.


} /* End of initialize_all() */
//-----------------------------------------------------------------------

/* Timer 1 Compare Match ISR */
ISR (TIMER1_COMPA_vect)
{
    PORTB ˆ= (1<<PORTB5);    // Toggle the LED
}
//-----------------------------------------------------------------------

/* Main Function */
int main(void)
{
    // Initialize everything
    initialize_all();

    while(1);    /* Nothing to do */

} /* End of main() */
//-----------------------------------------------------------------------
```

3. [30 points]: You want to toggle a LED connected to PB5 after every 250 milliseconds. One way to do it is by using Timer 1 'Overflow' ISR and a software counter. If the clock frequency ($clk_{I/O}$) is 16MHz, complete the "initialize_all()" function and "ISR(TIMER1_OVF_vect)" below such that the error in LED toggling period is **less than 1 millisecond**.
Hint: Running the Timer on higher frequencies provides more accurate results.
Hint: Overflow occurs when the counter reaches its maximum 16-bit value (MAX = 0xFFFF).

```
/* Global variable declarations */
volatile uint8_t software_counter;
volatile uint8_t counter_reset_value;

/* Initialization function */
void initialize_all(void)
{
    // Set the LED pin as Output here



    // Configure Timer 1 here.













    // Initialize 'counter_reset_value' with appropriate value here.




    // Initializing 'software_counter'
    software_counter = counter_reset_value;


     // Enable Global Interrupts here.



} /* End of initialize_all() */
//-----------------------------------------------------------------------
```

Initials:

```
/* Timer 1 Overflow ISR */
ISR(TIMER1_OVF_vect)
{
    /* Your code for ISR goes here */








}
//----------------------------------------------------------------------

/* Main Function */
int main(void)
{
    // Initialize everything
    initialize_all();

    while(1)
    {
        if( software_counter == 0 )
        {
            PORTB ^= (1<<PORTB5);    // Toggle the LED
            software_counter = counter_reset_value;
        }
    }

} /* End of main() */

//----------------------------------------------------------------------
```

Initials:

4. [20 points]: The ISR given below triggers periodically every $1ms$ and implements a simple Finite State Machine (FSM).

```
// Timer 0 Compare Match ISR
ISR (TIMER0_COMPA_vect)
{
    /* FSM Implementation */
    switch (Current_State)
    {
        case State_A:
        if(Flag == 0)    Current_State = State_B;
        else             Current_State = State_D;
        break;

        case State_B:
        if(Flag != 0)    Current_State = State_A;
        break;

        case State_C:
        Current_State = State_A;
        break;

        case State_D:
        if(Flag != 0)    Current_State = State_C;
        break;
    }
}
```

(a) Draw the state transition diagram of this FSM.

(b) Fill in the state transition table given below for this FSM.

Table 1: FSM State Transition Table

Time (ms)	**Flag**	**Current_State**
0	0	State_B
1	0	
2	1	
3	1	
4	0	
5	1	
6	0	
7	0	

End of Problem Set

Please double check that you wrote your name on the front of the quiz.

Initials: