Introduction to AVR (Atmega 16/32)

C Programming

Sagar B Bhokre

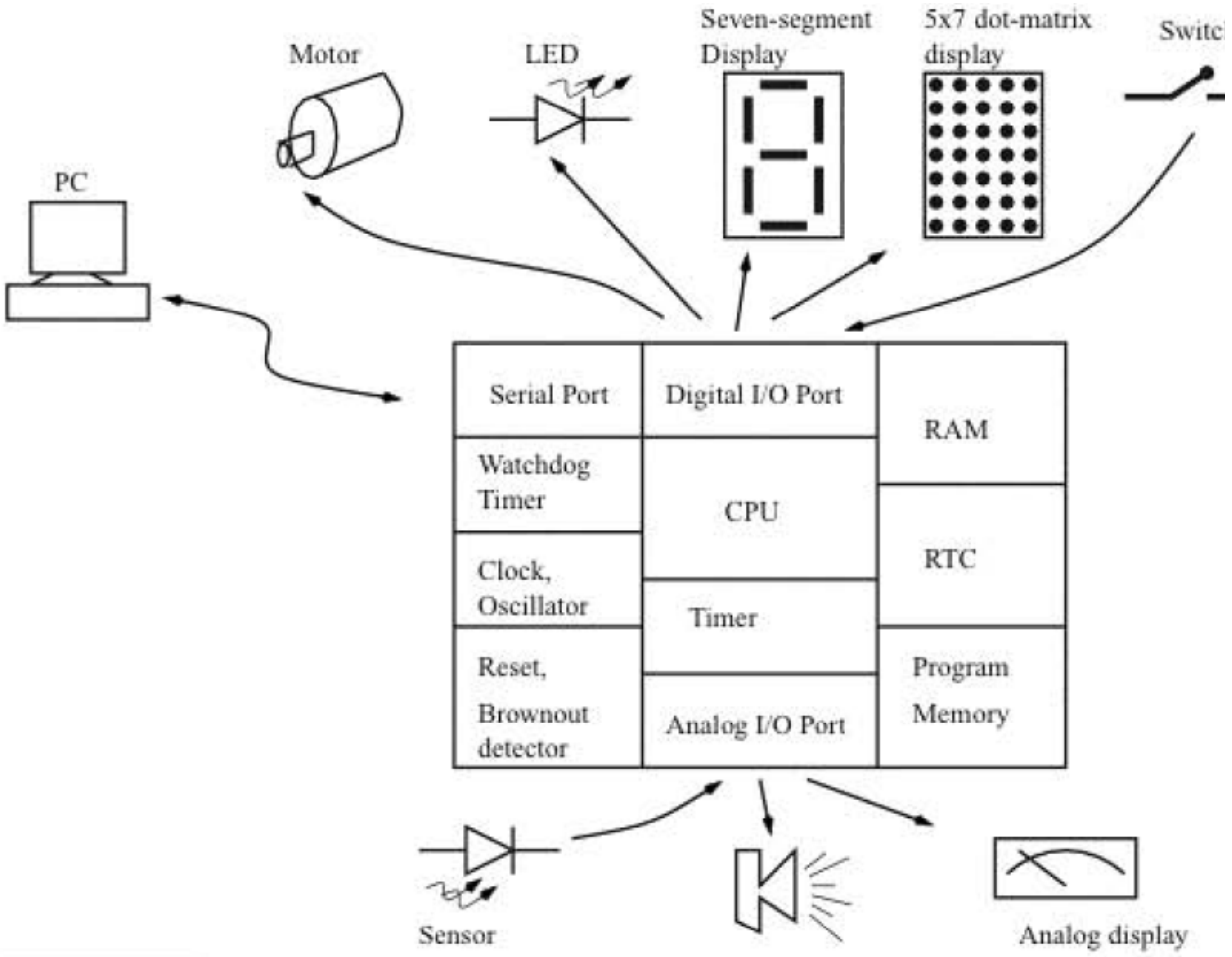
Research Associate, WEL LAB, IITB Powai, Mumbai - 76

Note

- The assembly language codes mentioned in these slides are just for understanding, most of the aspects will be handled by the C program. However ensuring the working (is handled by C) is left up to the programmer.

Microcontrollers

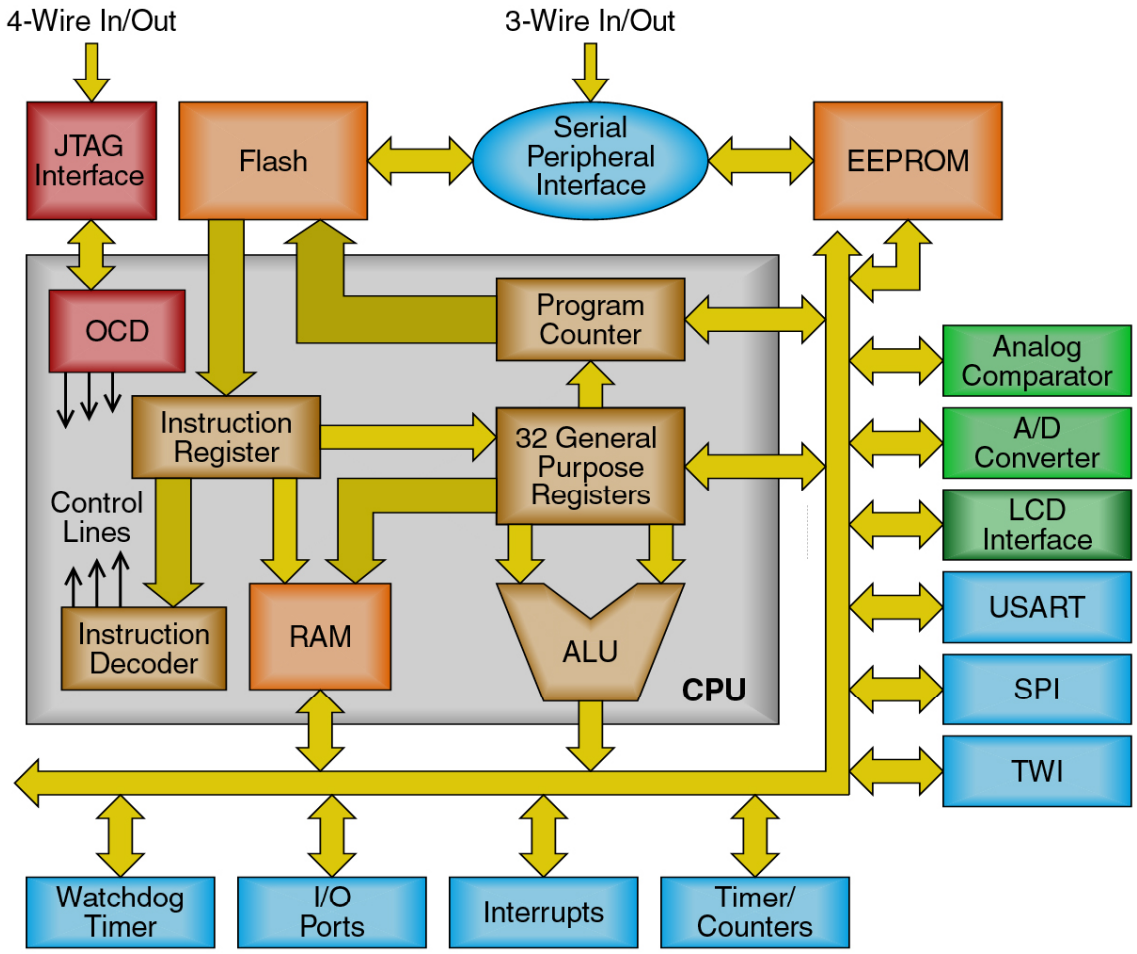
A microcontroller interfaces to external devices with a minimum of external components



AVR General Features

- The architecture of AVR makes it possible to use the storage area for constant data as well as instructions.
- Instructions are 16 or 32-bits
  - Most are 16-bits and are executed in a single clock cycle.
- Each instruction contains an opcode
  - Opcodes generally are located in the initial bits of an instruction

AVR Architecture



AVR General Features

- RISC architecture with mostly fixed-length instruction, load-store memory access and 32 general-purpose registers.
- A two-stage instruction pipeline that speeds up execution
- Majority of instructions take one clock cycle
- Up to 16-MHz clock operation

AVR General Features

- The ATMega16 can use an internal or external clock signal
  - Clock signals are usually generated by an RC oscillator or a crystal
    - The internal clock is an RC oscillator programmable to 1, 2, 4, or 8 MHz
    - An external clock signal (crystal controlled) can be more precise for time critical applications

AVR General Features

- Up to 12 times performance speedup over conventional CISC controllers.
- Wide operating voltage from 2.7V to 6.0V
- Simple architecture offers a small learning curve to the uninitiated.

What is an Interrupt

- A condition or event that interrupts the normal flow of control in a program
- Interrupt hardware inserts a function call between instructions to service the interrupt condition
- When the interrupt handler is finished, the normal program resumes execution

Interrupt Sources

- Interrupts are generally classified as
  - internal or external
  - software or hardware
- An external interrupt is triggered by a device originating off-chip
- An internal interrupt is triggered by an on-chip component

Interrupt Sources

- Hardware interrupts occur due to a change in state of some hardware
- Software interrupts are triggered by the execution of a machine instruction

Interrupt Handler

- An interrupt handler (or interrupt service routine) is a function ending with the special return from interrupt instruction (RETI)
- Interrupt handlers are not explicitly called; their address is placed into the processor's program counter by the interrupt hardware

AVR Interrupt System

- The ATMega16 can respond to 21 different interrupts
- Interrupts are numbered by priority from 1 to 21
  - The reset interrupt is interrupt number 1
- Each interrupt invokes a handler at a specific address in program memory
  - The reset handler is located at address $0000

Interrupt Vectors

- The interrupt handler for interrupt k is located at address 2(k-1) in program memory
  - Address $0000 is the reset interrupt
  - Address $0002 is external interrupt 0
  - Address $0004 is external interrupt 1
- Because there is room for only one or two instructions, each interrupt handler begins with a jump to another location in program memory where the rest of the code is found
  - *jmp handler* is a 32-bit instruction, hence each handler is afforded 2 words of space in this low memory area

Interrupt Vector Table

- The 21 instructions at address $0000 through $0029 comprise the interrupt vector table
- These jump instructions vector the processor to the actual service routine code
  - A long JMP is used so the code can be at any address in program memory
- An interrupt handler that does nothing could simply have an RETI instruction in the table
- The interrupt vector addresses are defined in the include file

Interrupt Enabling

- Each potential interrupt source can be individually enabled or disabled
  - The reset interrupt is the one exception; it cannot be disabled
- The global interrupt flag must be set (enabled) in SREG, for interrupts to occur
  - Again, the reset interrupt will occur regardless

Interrupt Actions

- If
  - global interrupts are enabled
  - AND a specific interrupt is enabled
  - AND the interrupt condition is present
- Then the interrupt will occur
- What actually happens?
  - At the completion of the current instruction,
    - the current PC is pushed on the stack
    - global interrupts are disabled
    - the proper interrupt vector address is placed in PC

Return From Interrupt

- The RETI instruction will
  - pop the address from the top of the stack into the PC
  - set the global interrupt flag, re-enabling interrupts
- This causes the next instruction of the previously interrupted program to be executed
  - At least one instruction will be executed before another interrupt can occur

Stack

- Since interrupts require stack access, it is essential that the reset routine initialize the stack before enabling interrupts
- Interrupt service routines should use the stack for temporary storage so register values can be preserved

Status Register

- Interrupt routines MUST LEAVE the status register unchanged
- Optional: Handled by C Program.

```
typical_interrupt_handler:
    push r0
    in r0, SREG
       …
    out SREG, r0
    pop r0
    reti
```

Interrupt Variations

- AVR Interrupts fall into two classes
  - Event based interrupts
    - Triggered by some event; must be cleared by taking some program action
  - Condition based interrupts
    - Asserted while some condition is true; cleared automatically when the condition becomes false

Event-based Interrupts

- Even if interrupts are disabled, the corresponding interrupt flag may be set by the associated event
- Once set, the flag remains set, and will trigger an interrupt as soon as interrupts are enabled
  - This type of interrupt flag is cleared
    - manually by writing a 1 to it
    - automatically when the interrupt occurs

Condition-based Interrupts

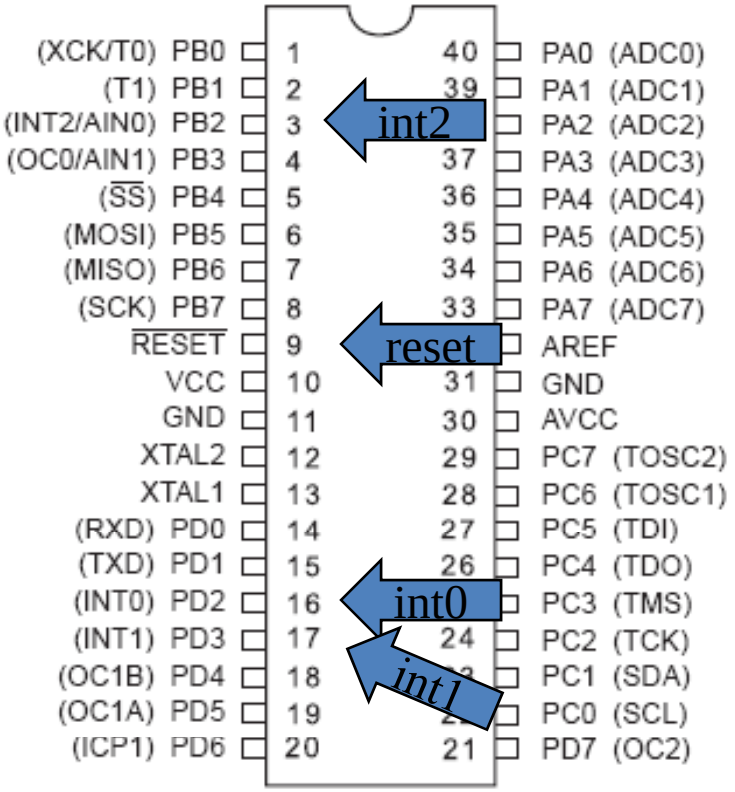
- Even if interrupts are disabled, the interrupt flag will be set when the associated condition is true
- If the condition becomes false before interrupts are enabled, the flag will be cleared and the interrupt will be missed
  - These flags are cleared when the condition becomes false
  - Some program action may be required to accomplish this

Sample Interrupts

- Event-based
  - Edge-triggered external interrupts
  - Timer/counter overflows and output compare
- Condition-based
  - Level triggered external interrupts
  - USART Data Ready, Receive Complete
  - EEPROM Ready

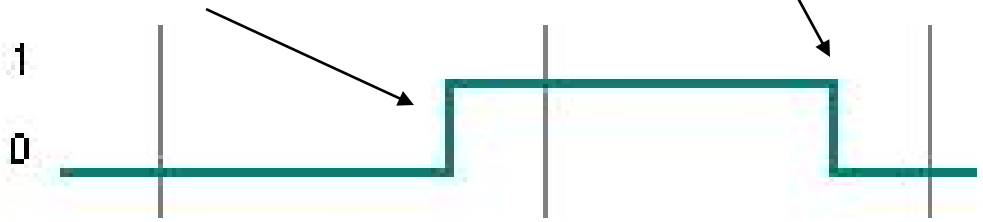
External Interrupts

- The ATMega16 responds to 4 different external interrupts – signals applied to specific pins
- RESET (pin 9)
- INT0 (pin 16 – also PD2)
- INT1 (pin 17 – also PD3)
- INT2 (pin 3 – also PB3)



External Interrupt Configuration

- Condition-based
  - while level is low
- Event-based triggers
  - level has changed (toggle)
  - falling (negative) edge (1 to 0 transition)
  - rising (positive) edge (0 to 1 transition)



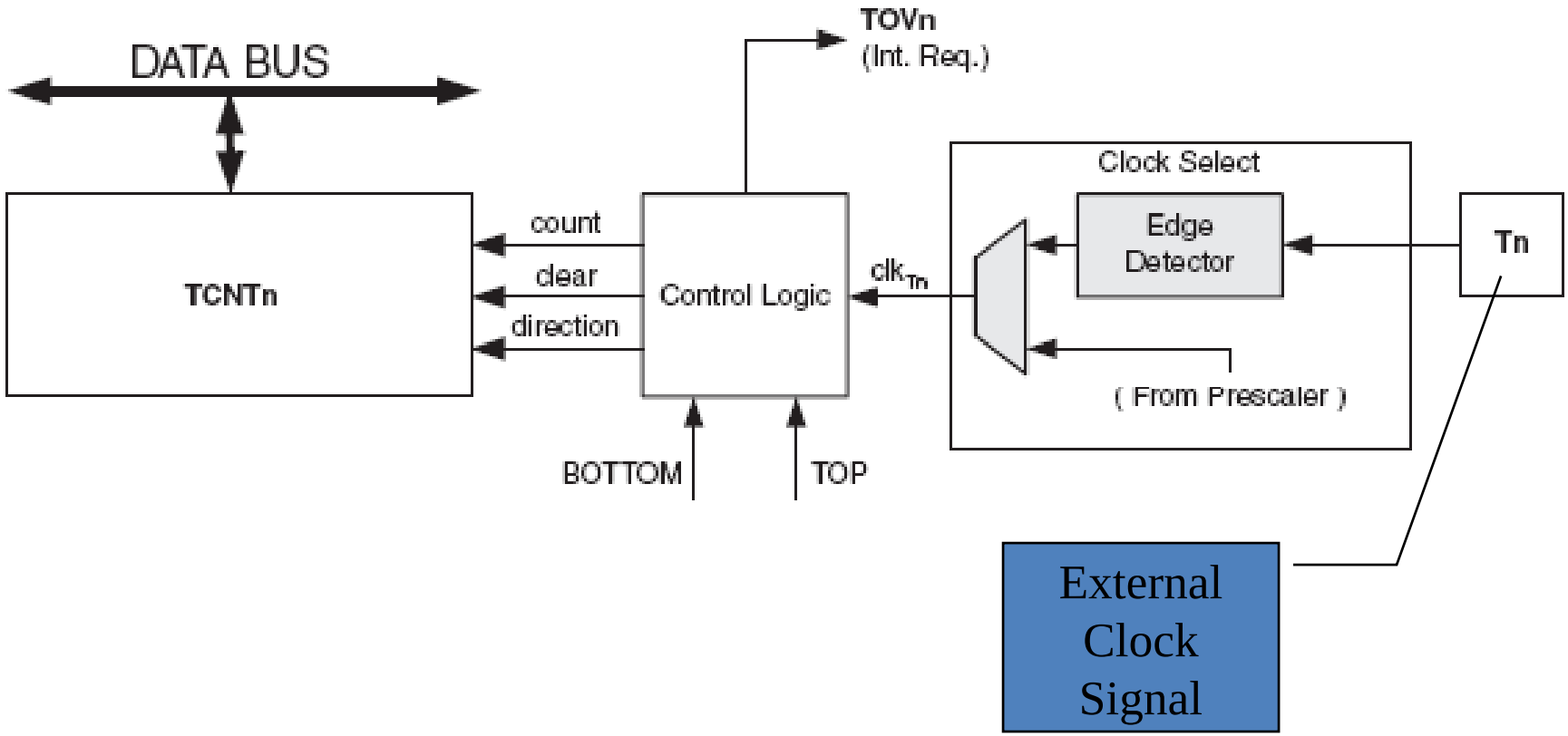
Software Interrupt

- If the external interrupt pins are configured as outputs, a program may assert 0 or 1 values on the interrupt pins
  - This action can trigger interrupts according to the external interrupt settings
- Since a program instruction causes the interrupt, this is called a software interrupt

Timer/Counters

- The ATMega16 has three timer/counter devices on-chip
- Each timer/counter has a count register
- A clock signal can increment or decrement the counter
- Interrupts can be triggered by counter events

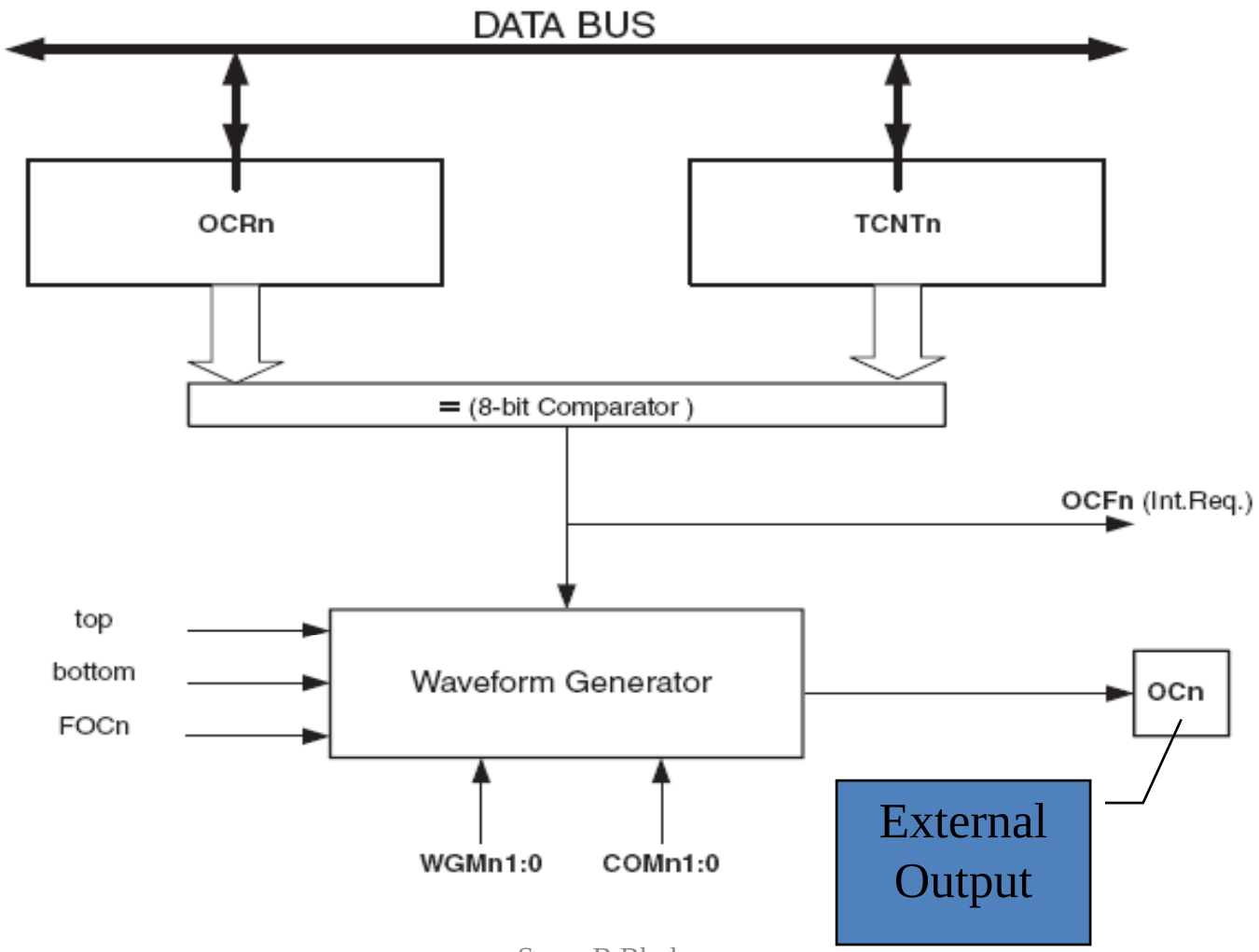
8-Bit Timer/Counter



Timer Events

- Overflow
  - In normal operation, overflow occurs when the count value passes $FF and becomes $00
- Compare Match
  - Occurs when the count value equals the contents of the output compare register
  - This can be used for PWM generation

Output Compare Unit



Status via Polling

- Timer status can be determined through polling
  - Read the Timer Interrupt Flag Register and check for set bits
  - The overflow and compare match events set the corresponding bits in TIFR
    - TOVn and OCFn (n=0, 1, or 2)
      - Timer 1 has two output compare registers: 1A and 1B
    - Clear the bits by writing a 1

Status via Interrupt

- Enable the appropriate interrupts in the Timer Interrupt Mask Register
- Each event has a corresponding interrupt enable bit in TIMSK
  - TOIEn and OCIEn (n = 0, 1, 2)
    - Again, timer 1 has OCIE1A and OCIE1B
  - The interrupt vectors are located at OVFnaddr and OCnaddr

Timer Interrupts

- The corresponding interrupt flag is cleared automatically when the interrupt is processed
  - It may be manually cleared by writing a 1 to the flag bit

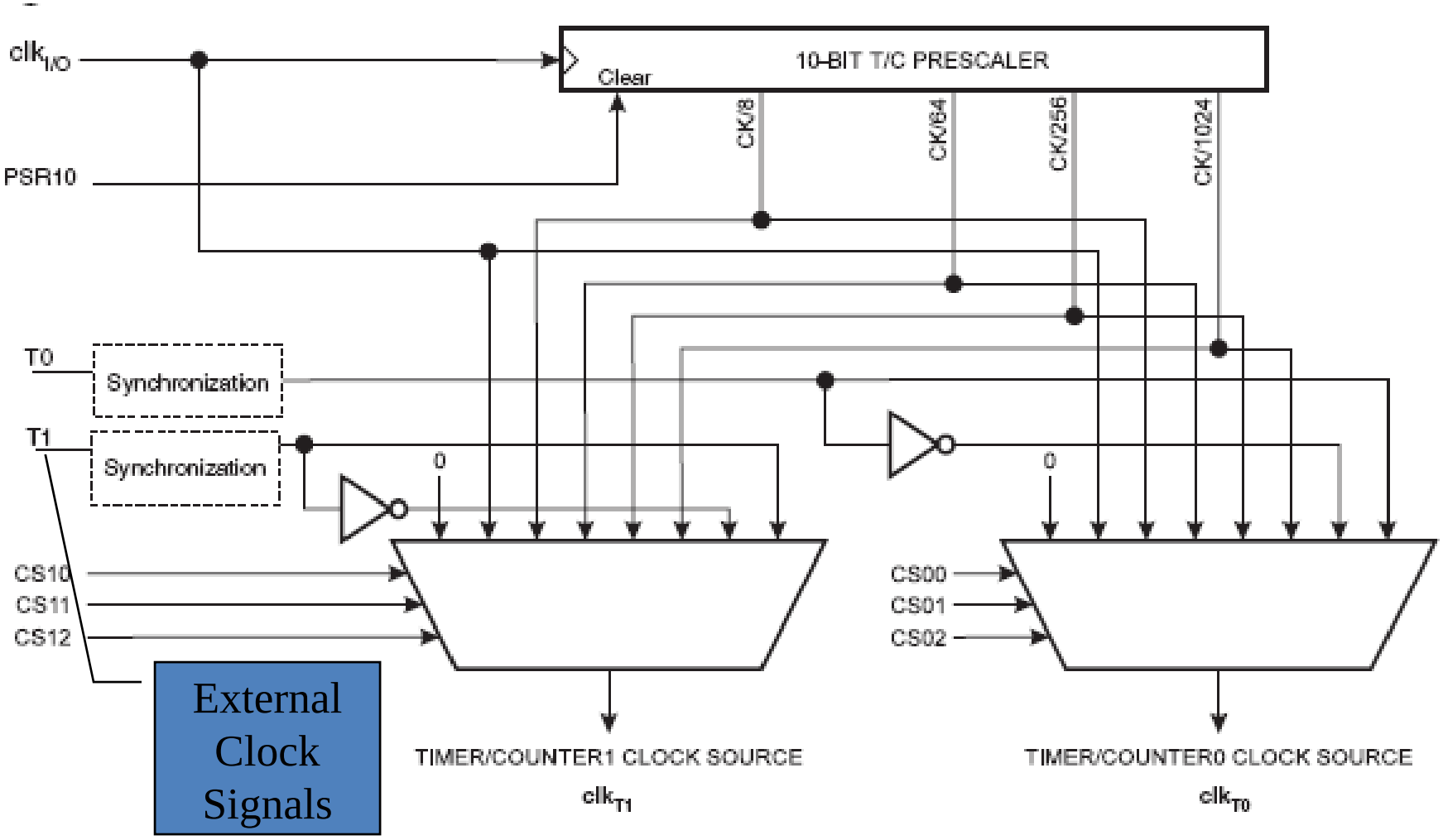
Automatic Timer Actions

- The timers (1 and 2 only) can be configured to automatically clear, set, or toggle related output bits when a compare match occurs
  - This requires no processing time and no interrupt handler – it is a hardware feature
  - The related OCnx pin must be set as an output; normal port functionality is suspended for these bits
    - OC0 (PB3) OC2 (PD7)
    - OC1A (PD5) OC1B (PD4)

Timer Clock Sources

- The timer/counters can use the system clock, or an external clock signal
- The system clock can be divided (prescaled) to signal the timers less frequently
  - Prescaling by 8, 64, 256, 1024 is provided
    - Timer2 has more choices allowing prescaling of an external clock signal as well as the internal clock

ATMega16 Prescaler Unit



Clock Selection

TCCR0 and TCCR1B – Timer/Counter Control Register (counters 0 and 1)

CSn2, CSn1, CSn0 (Bits 2:0) are the clock select bits (n = 0 or 1)

000 = Clock disabled; timer is stopped

001 = I/O clock

010 = /8 prescale

011 = /64 prescale

100 = /256 prescale

101 = /1024 prescale

110 = External clock on pin Tn, falling edge trigger

111 = External clock on pin Tn, rising edge trigger

TCCR2 – Timer/Counter Control Register (counter 2)

CS22, CS21, CS20 (Bits 2:0) are the clock select bits

000 = Clock disabled; timer is stopped

001 = T2 clock source

010 = /8 prescale

011 = /32 prescale

100 = /64 prescale

101 = /128 prescale

110 = /256 prescale

111 = /1024 prescale

ASSR (Asynchronous Status Register), bit AS2 sets the clock source to the internal clock (0) or external pin TOSC1)

Timer/Counter 1

- This is a 16 bit timer
  - Access to its 16-bit registers requires a special technique
- Always read the low byte first
  - This buffers the high byte for a subsequent read
- Always write the high byte first
  - Writing the low byte causes the buffered byte and the low byte to be stored into the internal register

There is only one single byte buffer shared by all of the 16-bit registers in timer 1

Timer/Counter 1 Control Register

- TCCR1A

TCCR1A Timer/Counter 1 Control Register A

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10

- TCCR1B

TCCR1B Timer/Counter 1 Control Register B

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

Timer 1 Data Registers

- TCNT1H:TCNT1L
  - Timer 1 Count
- OCR1AH:OCR1AL
  - Output Compare value – channel A
- OCR1BH:OCR1BL
  - Output Compare value – channel B
- ICR1H:ICR1L
  - Input Capture

Switch Bounce Elimination

- Pressing/releasing a switch may cause many 0-1 transitions
  - The bounce effect is usually over within 10 milliseconds
- To eliminate the bounce effect, use a timer interrupt to read the switch states only at 10 millisecond intervals
  - The switch state is stored in a global location to be available to any other part of the program

Debounce Interrupt

```
.dseg
switchstate: .byte 1

.cseg
switchread:
  push r16
  in R16, PIND
  com r16
  sts switchstate, r16
  pop r16
  reti
```

- Global variable holds the most recently accesses switch data from the input port
  - 1 will mean switch is pressed, 0 means it is not
- The interrupt is called every 10 milliseconds
- It simply reads the state of the switches, complements it, and stores it for global access

Timer Setup

- Use timer overflow interrupt
- Timer will use the prescaler and the internal 8 MHz clock source
  - Time between counts
    - 8Mhz/8 = 1 microsec
    - 8MHz/64 = 8 microsec
    - 8MHz/256 = 32 microsec
    - 8MHz/1024 = 128 microsec
- The maximum resolutions (256 counts to overflow) using these settings are
  - /1: 1.000 millisec
  - /8: 0.512 millisec
  - /32: 3.125 millisec
  - /128: 7.812 millisec
- Using a suitable prescale, find the required count that should be loaded in the timer.

Timer Initialization

```
.equ BOTTOM = 100
ldi temp, BOTTOM
out TCNT0, temp
ldi temp, 1<<TOIE0
out TIMSK, temp
ldi temp, 4<<CS00
out TCCR0, temp
sei
```

- A constant is used to specify the counter's start value
- The Timer Overflow interrupt is enabled
- The clock source is set to use the divide by x prescaler
- Global interrupts are enabled

Interrupt Task

- On each interrupt, we must reload the count value so the next interrupt will occur in 10 milliseconds
- We must also preserve the status register and registers used
- The interrupt will alter one memory location
  - `.dseg`
  - `;debounced PIND values`
  - `switchstate: .byte 1`

Interrupt Routine

```
switchread:
    push temp
    ldi temp, BOTTOM
    out TCNT0, temp
…switch processing details
    pop temp
    reti
```

- The counter has just overflowed (count is 0 or close to 0)
- We need to set the count back to our BOTTOM value to get the proper delay
- Remember to save registers and status flags as required

Application

```
   lds temp, switchstate
   ;1 in bit n means
   ;  switch n is down
   cpi temp, $00
   breq no_press

…process the switches

no_press:
…
```

- The application accesses the switch states from SRAM
  - This byte is updated ever 10 milliseconds by the timer interrupt

```
.dseg
switchstate: .byte 1
```

USART Interrupts

- Interrupt driven receive and transmit routines free the application from polling the status of the USART
  - Bytes to be transmitted are queued by the application; dequeued and transmitted by the UDRE interrupt
  - Received bytes are enqueued by the RXC interrupt; dequeued by the application

Cautions

- The queues are implemented in SRAM
- They are shared by application and interrupt
  - It is likely that there will be critical sections where changes should not be interrupted!

```
;A queue storage area
.dseg
queuecontents .byte MAX_Q_SIZE
front .byte 1
back .byte 1
size .byte 1
```

USART Configuration

- In addition to the normal configuration, interrupt vectors must be setup and the appropriate interrupts enabled
  - The transmit interrupt is only enabled when a byte is to be sent, so this is initially disabled
  - The receive interrupt must be on initially; we are always waiting for an incoming byte

sbi UCSRB, RXCIE

- The UDRE and TXC interrupts are disabled by default
- Other bits of this register must not be changed; they hold important USART configuration information
- The transmit complete interrupt is not needed

USART Interrupt Vectors

```
.org UDREaddr
jmp transmit_byte
.org URXCaddr
jmp byte_received
.org UTXCaddr
reti
```

- The interrupt vectors must be located at the correct addresses in the table
  - The include file has already defined labels for the addresses
  - The TXC interrupt is shown for completeness; it is not used in this example

Byte Received

- This interrupt occurs when the USART receives a byte and makes it available in its internal receive queue
- To prevent overflow of this 2 byte queue, the interrupt immediately removes it and places it in the larger RAM-based queue
- If another byte arrives during this routine, it will be caught on the next interrupt
  - Receive errors?
  - Queue full?
  - Registers saved?

Transmit Byte

- This occurs when UDRE is ready to accept a byte
- If the transmit queue is empty, disable the interrupt
- Otherwise place the byte into UDR
- The t_dequeue function returns a byte in R16
  - If no byte is available, it returns with the carry flag set

◉ Remember to save registers and status! (assembly language)

UDRIE?

- The UDRE Interrupt is enabled by the t_enqueue function
  - When a byte is placed into the queue, there is data to be transmitted
  - This is the logical place to enable the UDRE interrupt (if not already enabled)
    - Enable it after the item is enqueued, or it might occur immediately and find nothing to transmit!

Example: Interrupt Subroutine using WINAVR/AVR Studio

- ISR(SIG_UART_DATA) // Data register empty ISR
- {
- //Insert your code here.....
- }

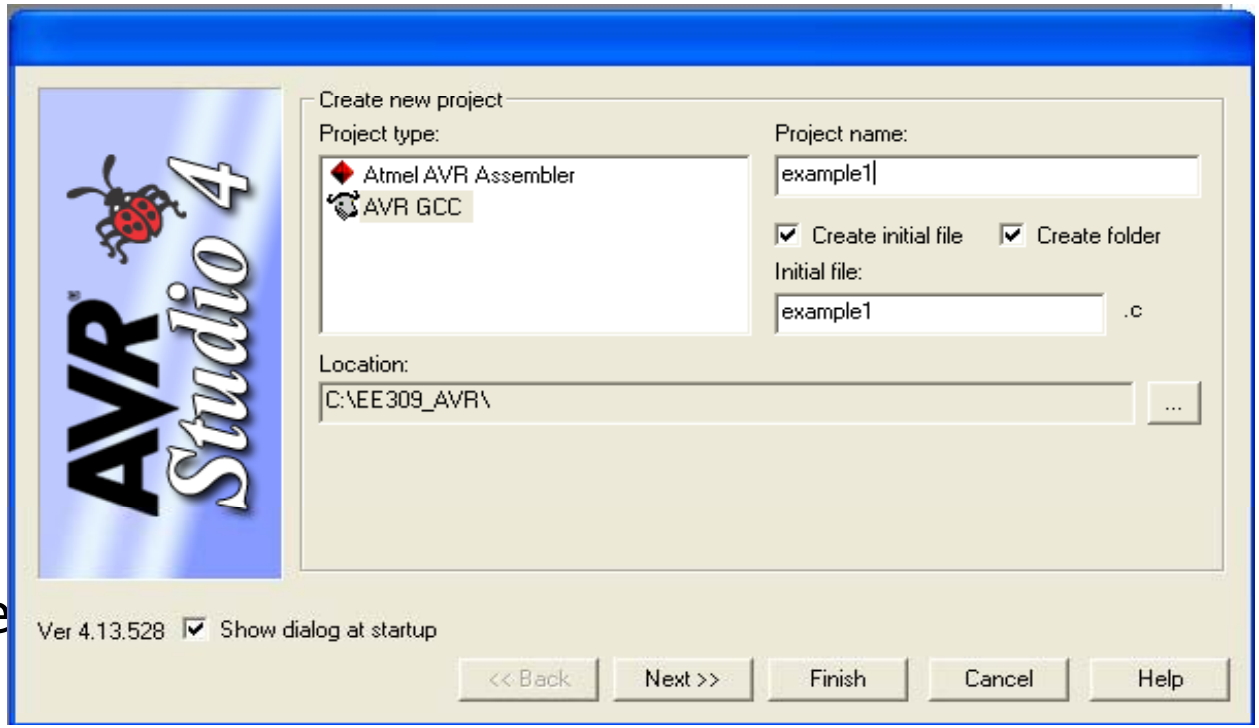
- Applications must ensure that critical sections are not interrupted

AVR Studio

- An integrated development environment
  - Provides a text editor
  - Supports the AVR assembler
  - Supports the gnu C compiler
  - Provides an AVR simulator and debugger
  - Provides programming support for the AVR processors via serial interface

AVR Studio: New Project

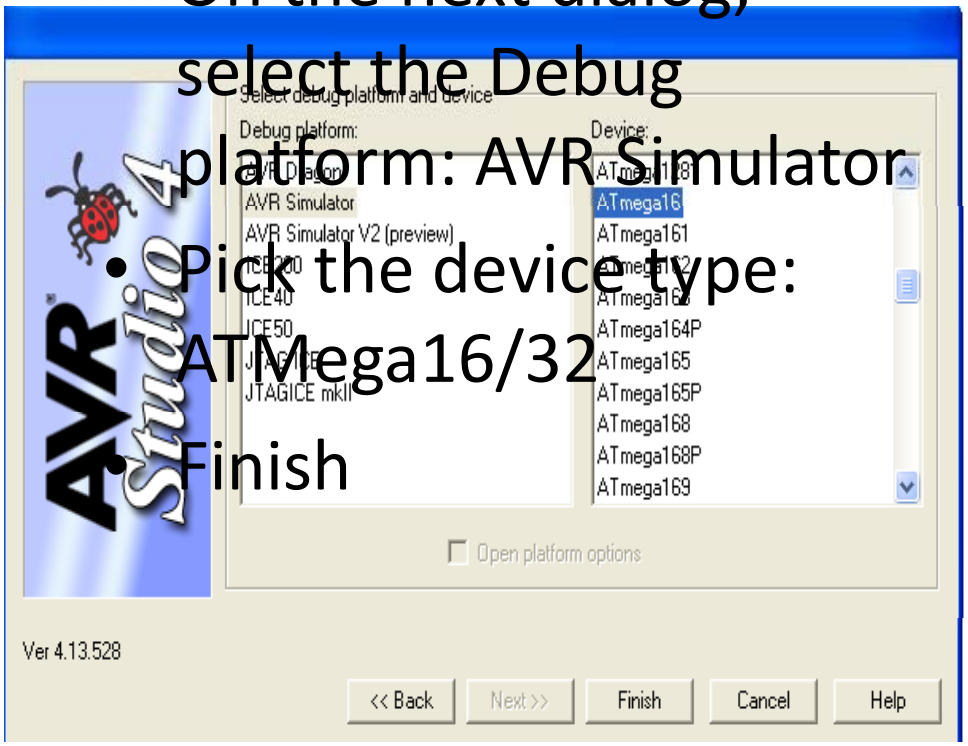
- Start AVR Studio
- Click New Project
- Select type: AVR GCC
- Choose a project name
- Select create options and pick a location



- Location should be a folder to hold all project folders
- Each project should be in its own folder

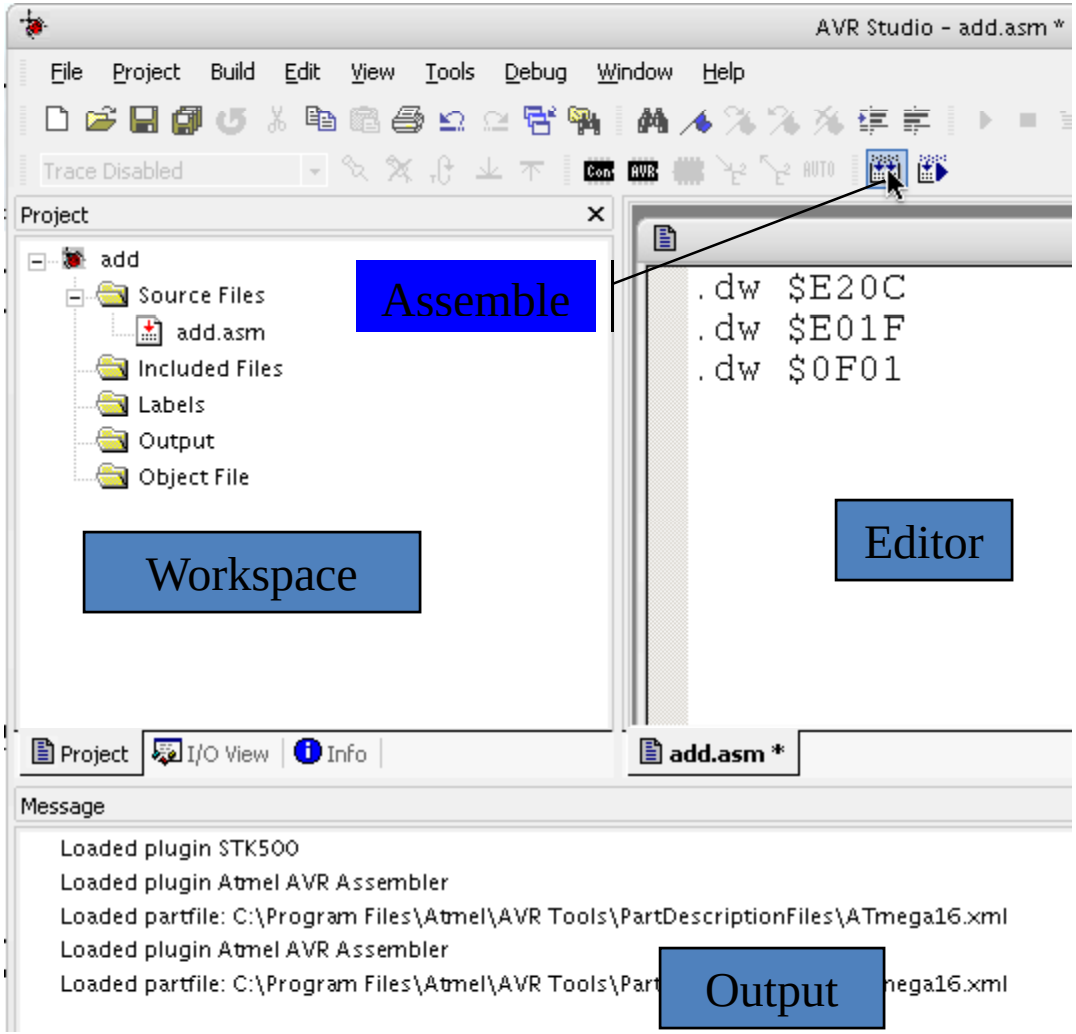
AVR Studio: New Project

- On the next dialog, select the Debug platform: AVR Simulator
- Pick the device type: ATMega16/32
- Finish

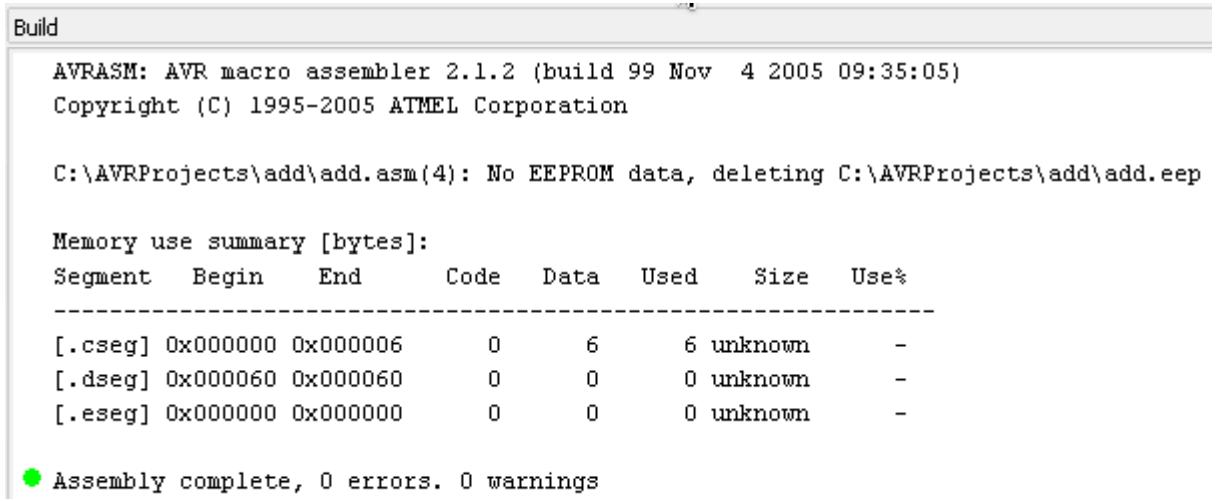


AVR Studio: Interface

- Enter the program in the assembly source file that is opened for you.
- Click the Assemble button (F7)

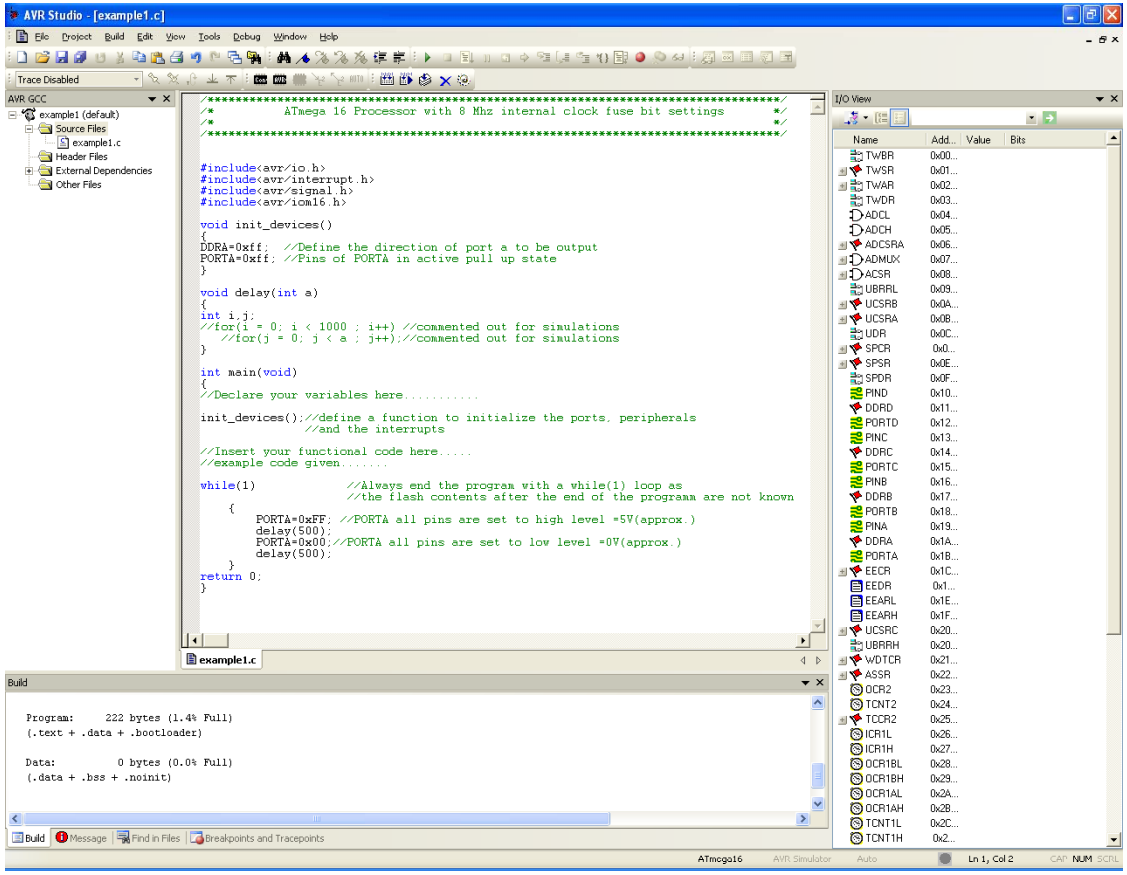


AVR Studio: Assembler-Report

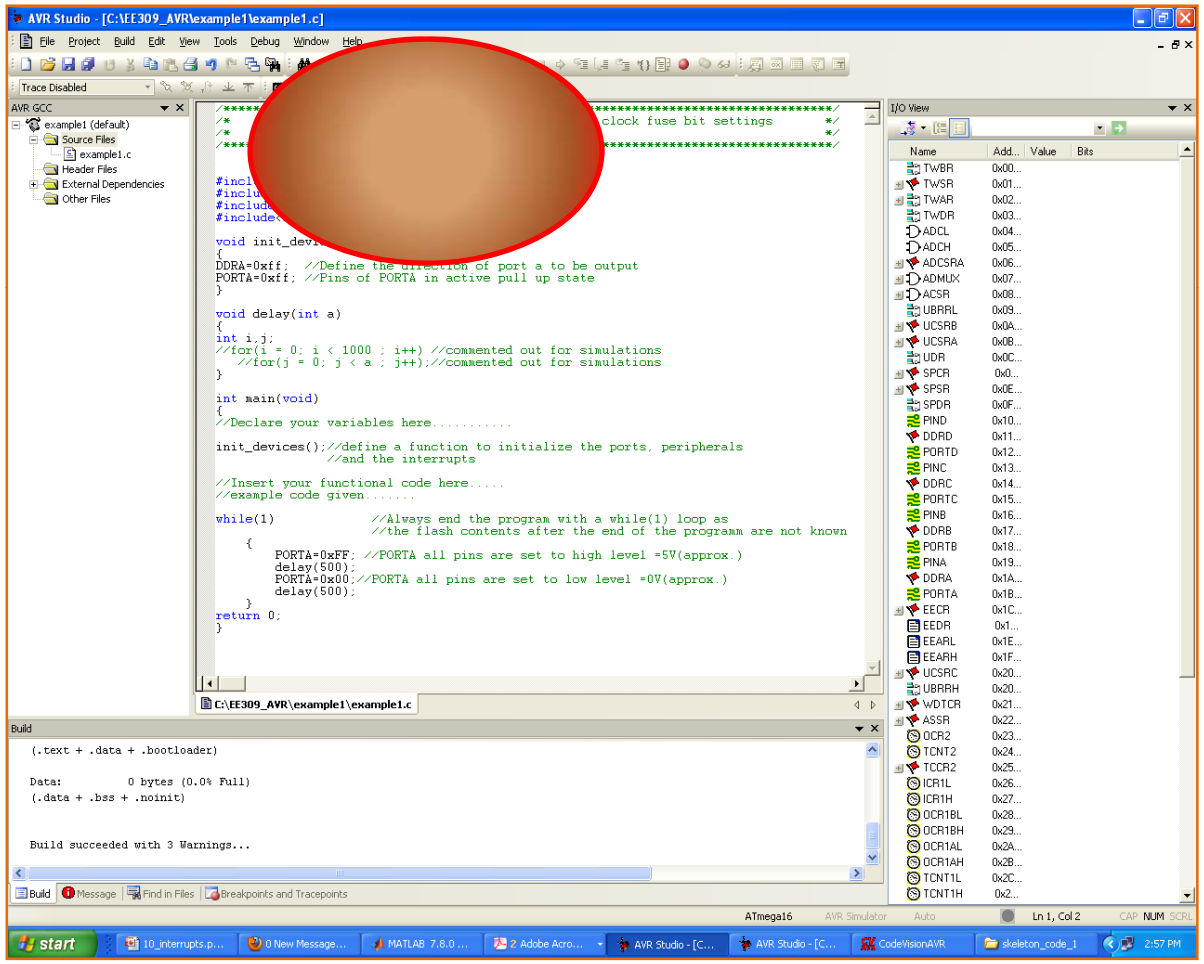


- Assembler summary indicates success
  - 6 bytes of code, no data, no errors

A general Program …

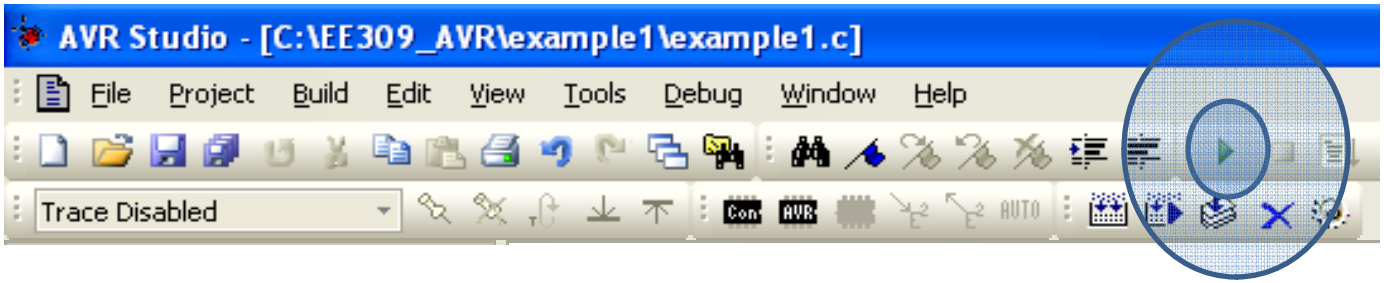


Build and Run…

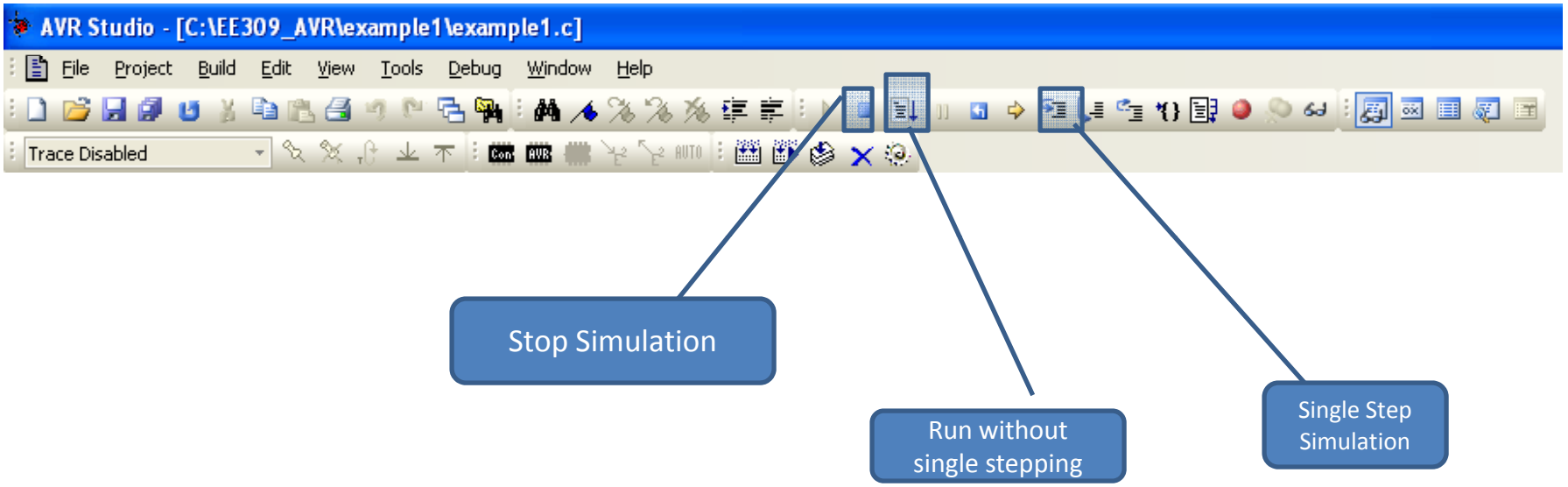


Build and Run…

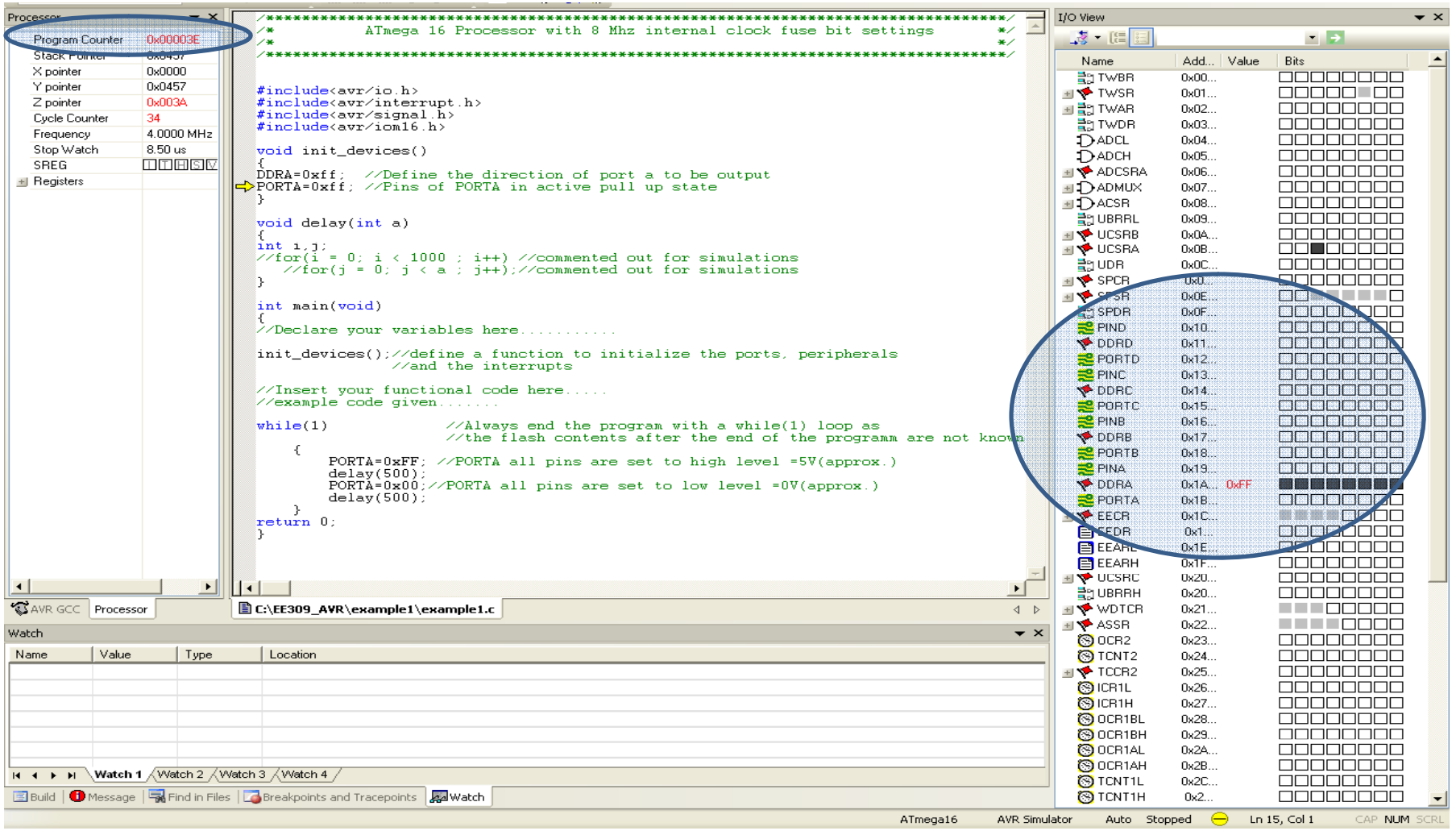
- Build the program and execute the same using the run command



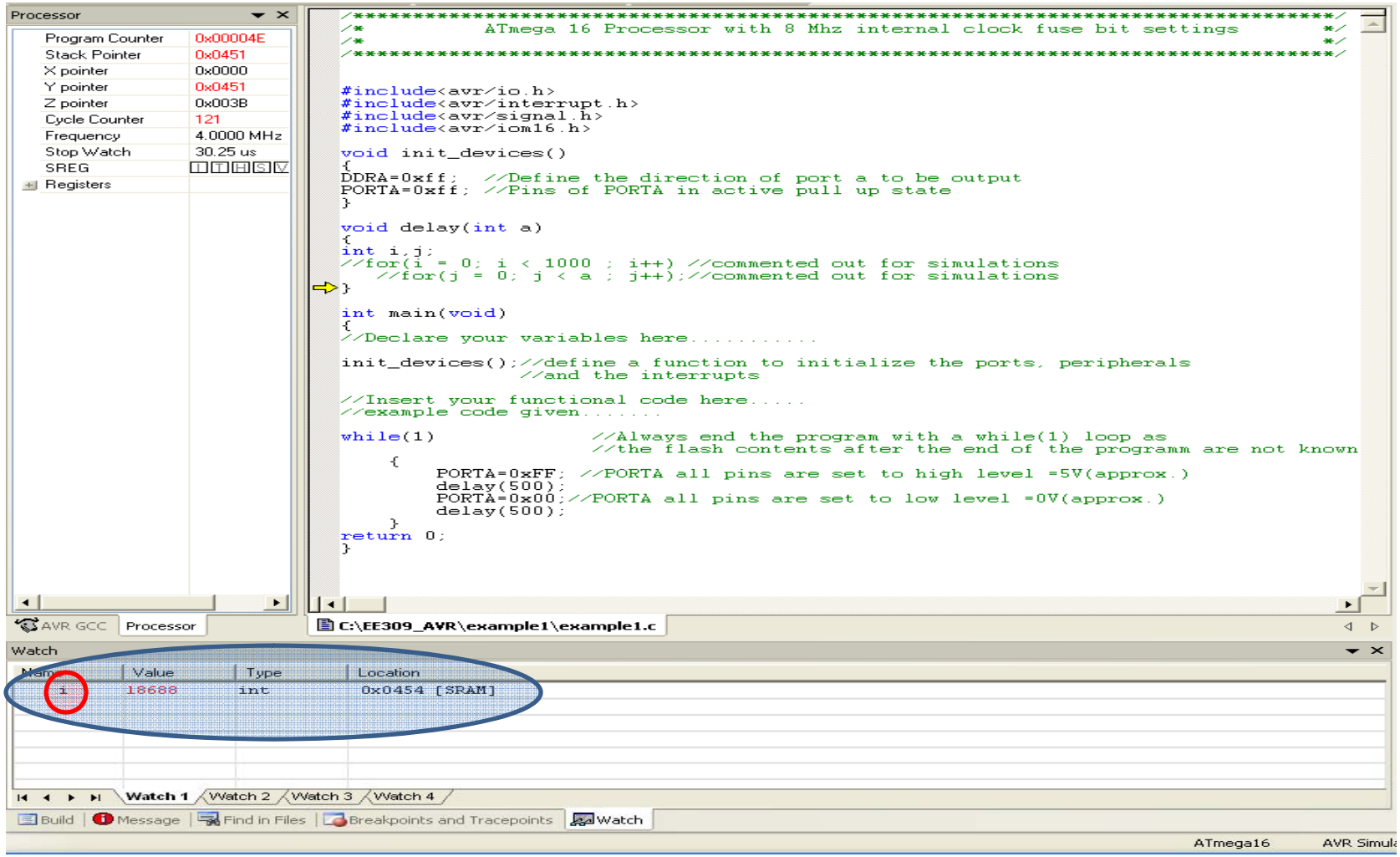
Single Step Simulation



To check the Register Contents

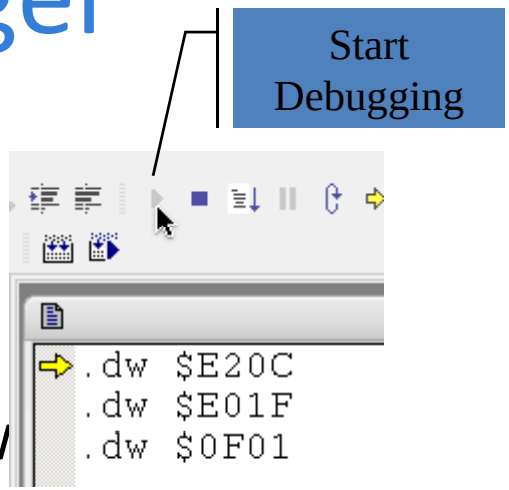


Watch window to monitor variable contents

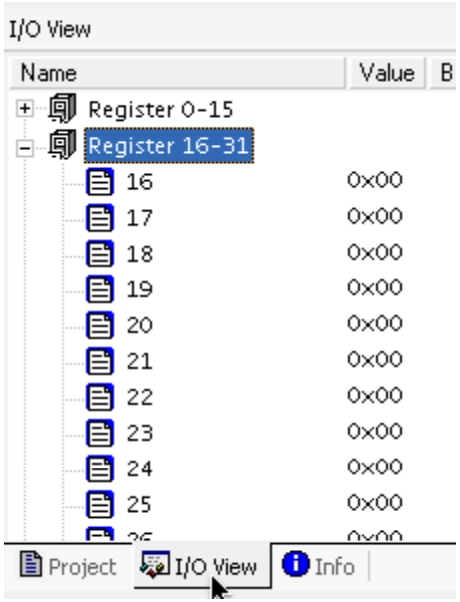



```
/*****************************************************************************/
/*            ATmega 16 Processor with 8 Mhz internal clock fuse bit settings */
/*                                                                           */
/*****************************************************************************/


#include<avr/io.h>
#include<avr/interrupt.h>
#include<avr/signal.h>
#include<avr/iom16.h>

void init_devices()
{
DDRA=0xff;  //Define the direction of port a to be output
PORTA=0xff; //Pins of PORTA in active pull up state
}

void delay(int a)
{
int i,j;
//for(i = 0; i < 1000 ; i++) //commented out for simulations
   //for(j = 0; j < a ; j++);//commented out for simulations
}

int main(void)
{
//Declare your variables here.....

init_devices();//define a function to initialize the ports, peripherals
               //and the interrupts

//Insert your functional code here.....
//example code given.....

while(1)                //Always end the program with a while(1) loop as
                        //the flash contents after the end of the programm are not known
     {
          PORTA=0xFF; //PORTA all pins are set to high level =5V(approx.)
          delay(500);
          PORTA=0x00;//PORTA all pins are set to low level =0V(approx.)
          delay(500);
     }
return 0;
}
```

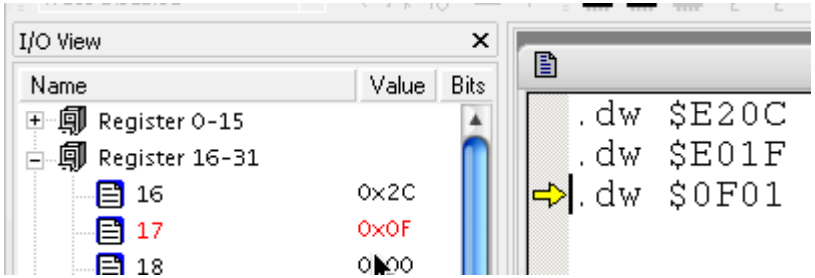
AVR Studio: Debugger



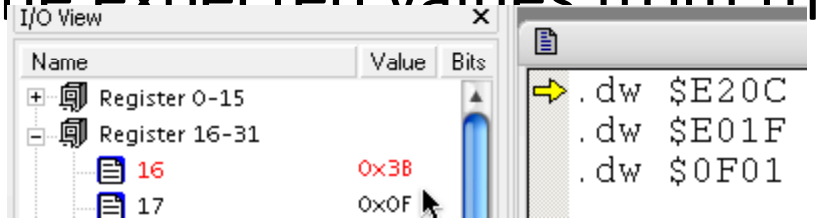
- Start the debugging session
  - Click Start Debugging
  - Next instruction is shown with yellov
- Choose I/O View
  - View registers 16-17
- Step through program
  - F10 is Step Over



AVR Studio: Debugger



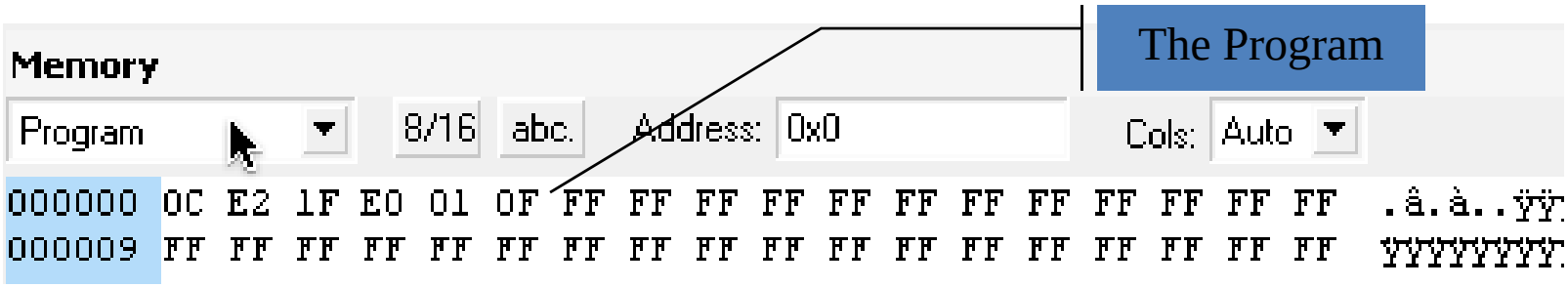
- The first 2 instructions are completed
  - R16 and R17 have the expected values from the LDI instructions



- The sum is placed in R16
  - $3B is the sum

AVR Studio: Memory

- Memory contents may be viewed (and edited) during debugging
  - You can view program (flash), data (SRAM), or EEPROM memory
  - You can also view the general purpose and I/O registers using this tool



What's Next?

- In our sample program, we executed three instructions, what comes next?
  - Undefined! Depends on what is in flash
- How do we terminate a program?
  - Use a loop!

```
0000: $E20C LDI R16, $2C
0001: $E01F LDI R17, $0F
0002: $0F01 ADD R16, R17
0003: $???? ???
```



```
If a program is to simply
   stop, add an instruction
   that jumps to its own
   address
```

References and Downloads

[1] "Assembly language programming" University of Akron Dr. Tim Margush

[2] Atmega16/32 Datasheet

[3] AVR Studio and WINAVR files

- AVR Studio

http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=2725

- WINAVR

http://sourceforge.net/projects/winavr/files/

- Install WINAVR first and then install AVR Studio.
- Sample codes mentioned in the Datasheet are the most reliable
- Try executing the sample code to get used to its procedure

Thank You

- For more details, visit the course website.

  http://sharada.ee.iitb.ac.in/~ee315
- For any further doubts or queries, feel free to contact me.
- email id: sagar@ee.iitb.ac.in

Sample Code

```
#include<avr/io.h>
#include<avr/interrupt.h>
#include<avr/signal.h>
#include<avr/iom16.h>

void init_devices() //initialization of devices
{
DDRA=0xff;  //Define the direction of port a to be output
PORTA=0xff;  //Pins of PORTA in active pull up state
UCSRA=0x00; //refer datasheet for details
UCSRB=0xF8;
UCSRC=0x86;
UBRRH=0x00;
UBRRL=0x33;
}

void delay(int a)
{
int i,j;
for(i = 0; i < 10 ; i++) //commented out for simulations
  for(j = 0; j < 10 ; j++);//commented out for simulations
}
ISR(SIG_UART_DATA) // Data register empty ISR
{
//Insert your code here........
UDR=0xaa;
}

int main(void)
{
//Declare your variables here.....
cli();
init_devices();//define a function to initialize the ports, peripherals
sei();            //and the interrupts

//Insert your functional code here.....
//example code given.......

while(1)          //Always end the program with a while(1) loop as
                  //the flash contents after the end of the programm are
      not known
      {
            PORTA=0xFF; //PORTA all pins are set to high level
      =5V(approx.)
            delay(2);
            PORTA=0x00;//PORTA all pins are set to low level
      =0V(approx.)
            delay(2);
      }
return 0;
}
```