Resources:

- **A good reference for getting started is**:
  - **Programming in AVR assembler language:** *** Commands sorted by function, Commands sorted by alphabet, Ports, Abbreviations
    - http://www.avr-asm-tutorial.net/avr_en/beginner/COMMANDS.html
  - **Beginners Programming in AVR Assembler -** many topics and tables
    - http://www.avr-asm-tutorial.net/avr_en/beginner/index.html
  - **Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors** by Gerhard Schmidt
    - http://www.avr-asm-download.de/beginner_en.pdf
  - **AVR-Assembler-Tutorial** Learning AVR Assembler with practical examples
    - http://www.avr-asm-tutorial.net
- **User Guides:**
  - **AVR Assembler User Guide** – *** a good, complete list of Assembler commands
    - www.atmel.com/Images/doc1022.pdf
  - **AVR Assembler2 User's Guide**
    - www.ic.unicamp.br/~celio/mc404.../avrassembler2-addendum.pdf
  - **AVR Assembler Help**
    - http://proton.ucting.udg.mx/tutorial/AVR/index.html
- **Subroutines**
  - **Writing subroutines**: After completing this tutorial readers should be able to: -Give a definition for the term subroutine -Write an assembly subroutine -Discuss the usefulness of macros.
    - http://www.avr-tutorials.com/assembly/writing-assembly-subroutines-avr-microcontroller

Useful Assembler Features:

- The Assembler supports a number of directives. The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory and so on. An nearly complete overview of the directives is given in the table at the right. The commands highlighted in blue are discussed on the following slides…

- These exerts taken from:
  - http://support.atmel.no/knowledgebase/avrstudiohelp/mergedProjects/AVRASM/Html/directives.html

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
CSEGSIZE	Program memory size
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define Constant word(s)
ENDM, ENDMACRO	EndMacro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn Macro expansion in list file on
MACRO	Begin Macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

Useful Assembler Features:

- **EQU - Set a symbol equal to a constant expression**
  - The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and can not be changed or redefined.

Syntax:

```
.EQU label = expression
```

Example:

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2

.CSEG                       ; Start code segment
          clr r2            ; Clear register 2
          out porta,r2      ; Write to Port A
```

Useful Assembler Features:

- **SET - Set a symbol equal to an expression**
  - The SET directive assigns a value to a label. This label can then be used in later expressions. While the function is very much like .EQU, it is different from the .EQU directive - because a label assigned to a value by the SET directive can be changed (redefined) later in the program.

Syntax:

```
.SET label = expression
```

Example:

```
.SET FOO = 0x114     ; set FOO to point to an SRAM
                     ;   location
      lds r0, FOO    ; load location into r0

.SET FOO = FOO + 1   ; increment (redefine) FOO. This
                     ;   would be illegal if using
                     ;   .EQU
      lds r1, FOO    ; load next location into r1
```

Useful Assembler Features:

- **DEF -Set a symbolic name on a register**
  - The DEF directive allows the registers to be referred to through symbols. A defined symbol can be used in the rest of the program to refer to the register it is assigned to. A register can have several symbolic names attached to it. A symbol can be redefined later in the program.

Syntax:

```
.DEF Symbol=Register
```

Example:

```
.DEF temp=R16
.DEF ior=R0

.CSEG
  ldi   temp,0xf0    ; Load 0xf0 into temp register
  in    ior,0x3f     ; Read SREG into ior register
  eor   temp,ior     ; Exclusive or temp and ior
```

Useful Assembler Features:

- **MACRO - Begin macro**
  - The MACRO directive tells the Assembler that this is the start of a Macro. The MACRO directive takes the Macro name as parameter. When the name of the Macro is written later in the program, the Macro definition is expanded at the place it was used. A Macro can take up to 10 parameters. These parameters are referred to as @0-@9 within the Macro definition. When issuing a Macro call, the parameters are given as a comma separated list. The Macro definition is terminated by an ENDMACRO directive.
  - By default, only the call to the Macro is shown on the listfile generated by the Assembler. In order to include the macro expansion in the listfile, a LISTMAC directive must be used. A macro is marked with a + in the opcode field of the listfile..

Syntax:

```
.MACRO macroname
```

Example:

```
.MACRO SUBI16 ; Start macro definition
subi @1,low(@0) ; Subtract low byte
sbci @2,high(@0) ; Subtract high byte
.ENDMACRO ; End macro definition
.CSEG ; Start code segment
SUBI16 0x1234,r16,r17 ; Sub.0x1234 from r17:r16
```

Additional Macro Resources:

- **Macros**
  - **Macros in AVR Assembler:** Macros are a good way to make code more readable (if it contains code that is often reused or if a lot of 16-bit calculations are done). Macros in AVR assembler can be defined anywhere in the code as long as they're created before they are used. They must take arguments which are replaced during assembly. They cannot be changed during runtime. The arguments are used in the form @0 or @1 (while 0 or 1 are the argument numbers starting from 0). The arguments can be almost everything the assembler can handle: integers, characters, registers, I/O addresses, 16 or 32-bit integers, binary expressions...
    - http://www.avrbeginners.net/assembler/macros.html
  - A very handy set of more advanced macros posted on a forum
    - http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=101529

#define to create a preprocessor function:

- Defining a preprocessor function-style macro "functions" using preprocessor directive #define

```
; bit mask macro identical to EXP2(),note use of shift operator
#define BITMASK(X) (1<<X)
#define BITMASK3(X1,X2,X3) (BITMASK(X1)+BITMASK(X2)+BITMASK(X3))
```

- Built-in functions for use in expressions – The following functions are defined; 'built-in' for the programmer's use
  - http://www.atmel.com/Images/doc1022.pdf#page17
    - **LOW** (expression) returns the low byte of an expression
    - **HIGH** (expression) returns the second byte of an expression
    - **BYTE2** (expression) is the same function as HIGH
    - **BYTE3** (expression) returns the third byte of an expression
    - **BYTE4** (expression) returns the fourth byte of an expression
    - **LWRD** (expression) returns bits 0-15 of an expression
    - **HWRD** (expression) returns bits 16-31 of an expression
    - **PAGE** (expression) returns bits 16-21 of an expression
    - **EXP2** (expression) returns 2^expression
    - **LOG2** (expression) returns the integer part of log2(expression)

#define to create a preprocessor function:

- Additional built-in functions are added in an addendum:
  - http://www.ic.unicamp.br/~celio/mc404-2008/docs/avrassembler2-addendum.pdf#page176

  - **INT**(expression) Truncates a floating point expression to integer (ie discards fractional part)
  - **FRAC**(expression) Extracts fractional part of a floating point expression (ie discards integer part).
  - **Q7**(expression) Converts a fractional floating point expression to a form suitable for the FMUL/FMULS/FMULSU instructions. (sign + 7-bit fraction)
  - **Q15**(expression) Converts a fractional floating point to the form returned by the FMUL/FMULS/FMULSY instructions (sign + 15-bit fraction).
  - **ABS**(expression) Returns the absolute value of a constant expression.

Stack and Functions:

- Using functions requires the stack. Using the stack requires that the stack pointer be initialized.

- The following code shows how to initialize the stack pointer:

```
.DEF SomeReg = R16

     LDI   SomeReg, HIGH(RAMEND) ; upper byte
     OUT   SPH,SomeReg           ;
     LDI   SomeReg, LOW(RAMEND)  ; lower byte
     OUT   SPL,SomeReg           ;
```

- Then, commands such as the following may be used:

```
PUSH   SomeReg
POP    SomeReg
RCALL  SomeLabel
RET
```

Function Example:

- The following AVR assembly program toggles the logic value on the pins of portB of an ATMega8515 AVR microcontroller with a delay after each change. Here the delay is provided by the **"Delay"**subroutine.

```
.include"m8515def.inc"
        ;Initialize the microcontroller stack pointer
        LDI R16,low(RAMEND)
        OUT SPL,R16
        LDI R16,high(RAMEND)
        OUT SPH,R16

        ;Configure portB as an output port
        LDI R16,0xFF
        OUT DDRB,R16

        ;Toggle the pins of portB
        LDI R16,0xFF
        OUT PORTB,R16
        RCALL Delay
        LDI R16,0x00
        OUT PORTB,R16
        RCALL Delay

;Delay subroutine
Delay:  LDI R17,0xFF
loop:   DEC R17
        BRNE loop
        RET
```

ASM Example Setting IO with different access methods:

- The following AVR assembly program toggles the logic value on the pins of portB of an ATMega8515 AVR microcontroller with a delay after each change. Here the delay is provided by the **"Delay"** subroutine.

```
;.INCLUDE "m169Pdef.inc"

.ORG 0x00000

;compute memory mapped io address
.EQU io_offset = 0x20
.EQU PORTA_MM = io_offset + PORTA ; PORTA is 0x02

;set i/o bits using i/o register direct commands(cannot be used w/ ext. i/o regs)
SBI PORTA, 6            ;set bit I/O using bit number
CBI PORTA, 6            ;clear bit I/O
SBI PORTA, 5            ;set bit I/O

;clear bit I/O using indirect access
LDI ZL,  low(PORTA_MM)  ;load immediate to register
LDI ZH,  high(PORTA_MM) ;low() and high() byte macros provided automatically
LD  R16, Z              ;load indirect from memory to reguster R16 using memory address
                        ;  R31,R30
CBR R16, EXP2(7)        ;clear bitS in register requires mask instead of a bit number
ST  Z,   R16            ;store indirect to memory address R31,R30 from reguster R16

;set bit I/O using indirect access
LDI ZL,  low(PORTA_MM)
LDI ZH,  high(PORTA_MM)
LD  R16, Z
SBR R16, EXP2(7)
ST  Z,   R16

;clear bit I/O using direct (memory) access
LDS R16 , PORTA_MM
CBR R16, EXP2(6)
STS PORTA_MM, R16

;set bit in I/O using direct (memory) access
LDS R16 , PORTA_MM
SBR R16, EXP2(7)
STS PORTA_MM, R16
```

Implementing Delays:

- Some options for implementing delays:
  - **Create a loop**

```
ldi RTEMP, 255 ; 255 could also be a variable here
the_delay:
dec RTEMP
brne the_delay;
```

  - **Use a few nop instructions**

```
nop ; 1 clock
nop ; 1 clock
nop ; 1 clock
```

  - **A combination for longer delay**

```
ldi RTEMP, 255 ; 255 could also be a variable here
the_delay:
nop
nop
nop
nop
nop
dec RTEMP
brne the_delay
```

Implementing Delays:

- Additional options for implementing delays:
  - **Create loops within loops**

```
; outer loop
ldi RTEMPB, 255 ; 255 could be a variable so the ; inner loop
sets the delay step size
outer_delay:
; inner loop
ldi RTEMPA, 122 ;
inner_delay:
nop
nop
nop
nop
nop
dec RTEMPA
brne inner_delay
dec RTEMPB
brne outer_delay
```

  - **Using double-word operations for longer delays**

```
LDI 25, 0x01 //high
LDI 24, 0xFF //low
Loop:
SUBIW R25:R24, 1
BRNE Loop
```

Implementing Delays:

- In case it wasn't obvious, these types of software delays are dependent on the CPU clock frequency.
- Note on delays using `nops`: there may be some productive work that can be done instead of using `nops`... maybe you can check button status while implementing a delay for blinking an LED. Replace the `nops` with productive instructions.
- More convenient precise delays use hardware timers, but we will learn that later.

Jumping based on single register bits:

- **Conditional Jumps**
  - SBIC - Skip if Bit in I/O Register Cleared
  - SBIS - Skip if Bit in I/O Register Set
  - SBRC - Skip if Bit in Register Cleared
  - SBRS - Skip if Bit in Register Set

- **Unconditional jumps**
  - RJMP k :
    - Program execution continues at address PC + k + 1.
    - 2 clock cycles.
    - The relative address k is from -2048 to 2047 (12 bits).
  - JMP k:
    - Program execution continues at address k
    - 3 clock cycles
    - Can jump anywhere

Jumping based on single register bits:

- **Combining conditional Jumps with Unconditional Jumps**

```
.def = JOYSTICK_INPORT  = PINB
.equ UP_BUTTON_BIT      = 5;
.equ UP_BUTTON_MASK     = (1<<UP_BUTTON_BIT);
.equ UP_BUTTON_MASK_CMP = (0xFF -UP_BUTTON_MASK);
.def RTEMP              = r16

;skip if bit in register set followed by branch
;  - branch occurs if button was pressed
sbis JOYSTICK_PORT, UP_BUTTON_BIT
rjmp somewhere
```

Jumping based on single register bits:

- **In comparison**

```
in   RTEMP, JOYSTICK_PORT
andi RTEMP, UP_BUTTON_MASK
brne somewhere
```

- **Example: waiting on a bit to change to a 1**

```
back_here:  SBIS PINB, 0
            rjmp back_here
            <some other code>
```

- **How to check on or wait for one of multiple bits?**
  - Not as simple, but that is HW
  - Hint:

```
.equ BUTTON_CHECK_MASK = (UP_BUTTON_MASK + DOWN_BUTTON_MASK)
```

Serial Code Using Function Call: asm

Init, Send,Receive, uses
Examples shown as functions/procedures

```
.include "m8515def.inc"
.def regA = r16
.def regB = r17
Serial_Init:
        ;Load UBRRH:UBRRL FCPU/BAUDRATE*16
        ;9600 at 4MHz
        ldi regA,00
        out UBRRH,regA
        ldi regA,25
        out UBRRL,regA
        ;Clear all error flags
        ldi regA,00
        out UCSRA,regA
        ;Enable Transmission and Reception
        ldi regA,(1<<RXEN)+(1<<TXEN)
        out UCSRB,regA
        ;Set Frame format
        ;8,N,1
        ldi regA,(1<<URSEL)|(3<<UCSZ0)
        out UCSRC,regA
        ret
```

```
;assumes data to send is in regB
;waits for one-character-buffer to empty and downloads uploads data
from regB
Serial_Send:
        ;wait for empty transmit buffer flag
        sbis UCSRA, UDRE
        rjmp Serial_Send
        ;If the flag is set
        ;Then move the data to send in UDR
        out UDR,regB
        ret

;waits for a character and downloads it to regB
Serial_Read:
        ;Wait for Receive flag
        sbis UCSRA,RXC
        rjmp Serial_Read
        ;If falg is set
        ;Then read data from UDR
        in regB,UDR
        ret
```