Resources:

- **User Guides:**
  - **AVR Assembler Help** - Probably one of the best listing of codes designed to help you get through…
    - http://proton.ucting.udg.mx/tutorial/AVR/index.html
  - **AVR Assembler User Guide** – *** a good, complete list of Assembler commands
    - www.atmel.com/Images/doc1022.pdf
  - **AVR Assembler2 User's Guide** – *** an addendum to the above.
    - www.ic.unicamp.br/~celio/mc404.../avrassembler2-addendum.pdf
- **A good reference for getting started is**:
  - **Programming in AVR assembler language: *** Commands sorted by function, Commands sorted by alphabet, Ports, Abbreviations – the second best summary out of all the following. The commands are all hyperlinked.
    - http://www.avr-asm-tutorial.net/avr_en/beginner/COMMANDS.html
  - **Beginners Programming in AVR Assembler -** many topics and tables – upfront table made up of hyperlinks for commands and functions. Like the above but reduced – doesn't show flags or clocks
    - http://www.avr-asm-tutorial.net/avr_en/beginner/index.html
  - ***AVR-Assembler-Tutorial*** Learning AVR Assembler with practical examples – contains the page above.
    - *http://www.avr-asm-tutorial.net*
  - **Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors** by Gerhard Schmidt – an overview of not only programming languages, but a complete paper on how hardware interacts with commands.
    - http://www.avr-asm-download.de/beginner_en.pdf
- **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://www.atmel.com/webdoc/avrassembler/avrassembler.wb_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)
```

One more time with academic example: Fibonacci Example

Fibonacci Numbers: By definition, the first two numbers in the Fibonacci sequence are either 1 and 1, (or 0 and 1, depending on the chosen starting point of the sequence,) and each subsequent number is the sum of the previous two.

Example:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

OPCODE		MNEMONIC	MEANING
000		ADD	Add memory to accumulator
001		SUB	Subtract memory from accumulator
010		LDA	Load accumulator from memory
011		STO	Store accumulator in memory
100		IOT	Input/Output Transfer: 0 = Input Int, 1 = Input Char, 2 = Output Int, 3 = Output Char
101		PSH	Push accumulator to top of stack
110		POP	Pop top of stack to accumulator
111	000	INS	Initialize stack pointer
111	001	BRA	Unconditional branch
111	010	BMI	Branch if accumulator is negative
111	011	JSR	Jump to subroutine
111	100	RTS	Return from subroutine
111	101	HLT	Program halt, return to monitor
		BSS	Reserve Space
		END	Set Pointer to Label
		ORG	Set next address at this location
		OCT	Set an Octal Constant
		DEC	Number entered as decimal

```
LABEL OPCODE ADDRESS        ADDRESS    DATA
-------------------         ----------------
UNUSED EQU    2047
*
       ORG    20
TEMP   DEC    140            020      0214
NEXT   OCT    UNUSED         021      2047
LATEST BSS    1              022
LIMIT  BSS    1              023
SUM    BSS    1              024
*
       ORG    40
START  INS    777            040      7000  0777
       JSR    INPUT          042      7300  0045
*
       LDA    NEXT           044      2021
       IOT    2              045      4002
       LDA    LATEST         046      2022
       IOT    2              047      4002
LOOP   LDA    LATEST         050      2022
       ADD    NEXT           051      0021
       STO    SUM            052      3024
       IOT    2              053      4002
       LDA    LIMIT          054      2023
       SUB    SUM            055      1024
       BMI    STOP           056      7200  0066
       LDA    LATEST         060      2022
       STO    NEXT           061      3021
       LDA    SUM            062      2024
       STO    LATEST         063      3022
       BRA    LOOP           064      7100  0050
STOP   HLT                    066      7500
*
INPUT  IOT    0              067      4000
       STO    NEXT           070      3021
       IOT    0              071      4000
       STO    LATEST         072      3022
       IOT    0              073      4000
       STO    LIMIT          074      3023
       RTS                   075      7400
*
       END    START
```

Actual UART Example in AVR: 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
```

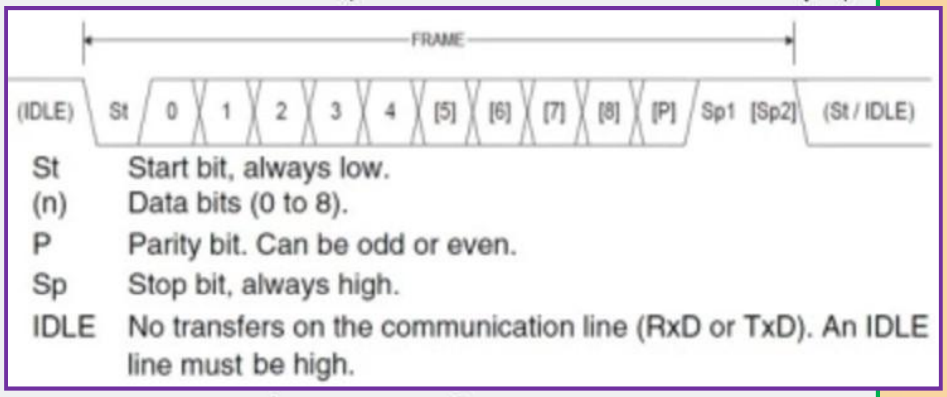
Helpful Website for UART programming in AVR: (notes, ref on following slides)
http://maxembedded.com/2013/09/the-usart-of-the-avr/

UART: review and AVR usage of UBRRH & UBRRL

- UART stands for Universal Asynchronous Receiver/Transmitter. From the name itself, it is clear that it is asynchronous i.e. the data bits are not synchronized with the clock pulses.

```
void USART_initial (void){
#define BAUD 9600 // We set the desired baud rate( here we set it at 9600bps)
#include <util/setbaud.h>  //
UBRRH = UBRRH_VALUE;
UBRRL = UBRRL_VALUE;
#if USE_2X
UCSRA |= (1 << U2X);
#else
UCSRA &= ~(1 << U2X);
#endif

UCSRB = (1<<RXEN)|(1<<TXEN); // Enable transmitter/receiver.
UCSRC = (1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1); // Character size : 8 bit
```



- The UART needs a clock signal that determines the baud rate. It is generated in the chip by dividing the CPU clock frequency by the UBRR register value. It must be 16x higher than the desired baud rate. The x16 factor is used by the UART to sub-sample the received serial data, it improves noise immunity by calculating the received bit value from the average of 16 samples. (Detailed Register bit assignments on following slides)

- So if the desired baudrate is 9600 baud and the CPU clock is 16 MHz then UBBR is (16000000 / (16 x 9600)) - 1 = 103.167. Round that to the closest integer = 103. Which makes UBRRL = 0x67, UBRRH = 0x00. You should see this calculation being made in the util/setbaud.h source file.

UART registers in AVR continued:

UBRR: USART Baud Rate Register (16-bit) – Previous Slide…

Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	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	
	0	0	0	0	0	0	0	0	

Bit 15: URSEL: This bit selects between accessing the UBRRH or the UCSRC Register. It is read as zero when reading UBRRH. The URSEL must be zero when writing the UBRRH. Detailed usage is found on the previous slide.

UDR: USART Data Register (16-bit)

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR (Read)
	TXB[7:0]								UDR (Write)
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	

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDR. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB). For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver.

UCSRA: USART Control and Status Register A (8-bit)

Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Bit 7: RxC – USART Receive Complete Flag: This flag bit is set by the CPU when there are unread data in the Receive buffer and is cleared by the CPU when the receive buffer is empty. This can also be used to generate a Receive Complete Interrupt (see description of the RXCIE bit in UCSRB register).
Bit 6: TxC – USART Transmit Complete Flag: This flag bit is set by the CPU when the entire frame in the Transmit Shift Register has been shifted out and there is no new data currently present in the transmit buffer (UDR). The TXC Flag bit is automatically cleared when a Transmit Complete Interrupt is executed, or it can be cleared by writing a *one (yes, one and NOT zero)* to its bit location. The TXC Flag can generate a Transmit Complete Interrupt (see description of the TXCIE bit in UCSRB register).
Bit 5: UDRE – USART Data Register Empty: The UDRE Flag indicates if the transmit buffer (UDR) is ready to receive new data. If UDRE is one, the buffer is empty, and therefore ready to be written. The UDRE Flag can generate a Data Register Empty Interrupt (see description of the UDRIE bit in UCSRB register). UDRE is set after a reset to indicate that the Transmitter is ready.
Bit 4: FE – Frame Error: This bit is set if the next character in the receive buffer had a Frame Error when received (i.e. when the first stop bit of the next character in the receive buffer is zero). This bit is valid until the receive buffer (UDR) is read. The FE bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRA.
Bit 3: DOR – Data Overrun Error: This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), and a new start bit is detected. This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.
Bit 2: PE – Parity Error: This bit is set if the next character in the receive buffer had a Parity Error when received and the parity checking was enabled at that point (UPM1 = 1). This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.
Bit 1: U2X – Double Transmission Speed: This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.
Bit 0: MPCM – Multi-Processor Communication Mode: This bit enables the Multi-processor Communication mode. When the MPCM bit is written to one, all the incoming frames received by the USART Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the MPCM setting. This is essential when the receiver is exposed to more than one transmitter, and hence must use the address information to extract the correct information.

UART registers in AVR continued:

UCSRB: USART Control and Status Register B (8-bit)

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit 7: RXCIE – RX Complete Interrupt Enable: Writing this bit to one enables interrupt on the RXC Flag. A USART Receive Complete interrupt will be generated only if the RXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXC bit in UCSRA is set. The result is that whenever any data is received, an interrupt will be fired by the CPU.
Bit 6: TXCIE – TX Complete Interrupt Enable: Writing this bit to one enables interrupt on the TXC Flag. A USART Transmit Complete interrupt will be generated only if the TXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXC bit in UCSRA is set. The result is that whenever any data is sent, an interrupt will be fired by the CPU.
Bit 5: UDRIE – USART Data Register Empty Interrupt Enable: Writing this bit to one enables interrupt on the UDRE Flag (remember – bit 5 in UCSRA?). A Data Register Empty interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDRE bit in UCSRA is set. The result is that whenever the transmit buffer is empty, an interrupt will be fired by the CPU.
Bit 4: RXEN – Receiver Enable: Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the RxD pin when enabled.
Bit 3: TXEN – Transmitter Enable: Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxD pin when enabled.
Bit 2: UCSZ2 – Character Size: The UCSZ2 bits combined with the UCSZ1:0 bits in UCSRC register sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use. More information given along with UCSZ1:0 bits in UCSRC register.
Bit 1: RXB8 – Receive Data Bit 8: RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. It must be read before reading the low bits from UDR.
Bit 0: TXB8 – Transmit Data Bit 8: TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. It must be written before writing the low bits to UDR.

UCSRC: USART Control and Status Register C (8-bit)

Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

Bit 7: URSEL – USART Register Select: This bit selects between accessing the UCSRC or the UBRRH Register. It is read as one when reading UCSRC. The URSEL must be one when writing the UCSRC.
Bit 6: UMSEL – USART Mode Select: This bit selects between Asynchronous and Synchronous mode of operation.

0	Asynchronous Operation
1	Synchronous Operation

Bit 5:4: UPM1:0 – Parity Mode: This bit helps you enable/disable/choose the type of parity.

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

Bit 3: USBS – Stop Bit Select: This bit helps you choose the number of stop bits for your frame.

0	1-bit
1	2-bit

Bit 2:1: UCSZ1:0 – Character Size: These two bits in combination with the UCSZ2 bit in UCSRB register helps choosing the number of data bits in your frame.
Data Bit Settings (Click to Enlarge)

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Bit 0: UCPOL – Clock Polarity: This bit is used for Synchronous mode only. Write this bit to zero when Asynchronous mode is used. The UCPOL bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK).