A MORON'S GUIDE TO STARTING IN AVR ASSEMBLER v1.2

by RetroDan@GMail.com

TABLE OF CONTENTS:

INTRODUCTION

Conceptually Assembly Language Programming is very simple, you typically move a byte of data into a register, you do something to it, then you write it out. Practically all languages convert their programs to assembly because assembly is close to what the hardware understands, so there is almost a one-to-one relationship between assembly instructions and hardware binary code (machine language).

Assembly may appear difficult to the novice because of the initial steep learning curve, however because of the limited number of instructions, once over-the-hump it is very simple. Before you can start you need a good understanding of the architecture of the machine, the instruction set, the assembler's syntax and a basic understanding of programming principles such as looping and subroutines before you can create your first program.

One of the benefits of learning Assembly is that they are all very similar, so once you learn one, the rest can be picked up easily since most processors do the exact same thing at the hardware level, basically moving bytes around.

Assembly allows you to create the smallest and fastest code possible, however it can be long and tedious for large projects.

BASIC AVR ARCHITECTURE

In its simplest form the AVRs are made of Registers, Ports and RAM:

```
[REGISTERS]
  [PORTS]
   [RAM]
```

There are 32 registers which can be though of as fast RAM so it is where most of the work is done. As I mention previously, most of Assembly is moving data into these registers and doing something with them.

RAM is where we store our programs and data.

The ports are how we communicate with the outside world but they appear to the MCU as additional registers.

THE REGISTERS

The registers are one byte each (eight bits) and are internal to the MCU so they operate quickly. We can think of them as a work space to get things done before they are sent off somewhere. So before we can write assembly programs for the AVR, a good understanding of the registers is important.

There are 32 internal registers in the AVRs typically referred to as R0-R31. The most often used is R16-R31 because they are easier to use. They can be loaded directly with a constant. For example, if you want to load 100 into register R16 with 100 you would use:

```
LDI R16,100   ;LDI = LoaD Immediate 100 into Register 16
```

LDI means LoaD Immediate, R16 is the register and 100 is our constant. Anything after the semi-colon ";" is a comment which is ignored by the Assembler.

If you want to move the value of 100 into one of the Registers R0-R15 you CANNOT do this:

```
LDI R1,100  ;This is an ERROR!!!
```

To move 100 to one of the registers R0-R15 you would do something like:

```
LDI R16,100    ;Load 100 into Register 16
MOV R1,R16     ;Move the contents of Register 16 to Register 1
```

This moves the 100 into Register R16 first, then we move it from R16 then into R0. Note that the operands are read right-to-left. The MOV is from R16 to R1 and not the reverse, even though it may appear to read that way.

So we can see that Registers R16-R31 are easier to use because they are half the work to load. Out of these, Registers R26-R31 are used as two-byte pointers by more advanced commands, so we should stick to the ten Registers R16-R25 as our main workspace to start.

REGISTER DEFINITIONS

We can use the .DEF command to give our registers meaningful names.

```
.DEF A = R16
.DEF B = R18
.DEF N = R20
.DEF STUDENT_SCORE = R25
```

Now we can load the R16 Register with 100 using the command:

```
LDI A,100
```

CONSTANTS

Constants are values that do not change value while the program is running. They are defined at the time your program is assembled into machine language (binary code) and do not change when your program is executed.

Constants can be given a name with the .SET (or .EQU) command. In our last example we loaded the R16 register with the value of 100. Instead of using the constant 100 we could give it a name like PERFECT_SCORE with the statements:

```
.SET PERFECT_SCORE = 100
.EQU PERFECT_SCORE = 100
```

Then later in the program we can load R16 with 100 using the command:

```
LDI R16,PERFECT_SCORE
```

Constants can be represented in a number of ways. They can be defined as hexadecimal, octal, binary, etc. All of the following define PERFECT_SCORE as 100:

```
.SET PERFECT_SCORE = 100              ;Decimal notation
.SET PERFECT_SCORE = (2000+500)/25    ;2500 divided by 25 = 100
.SET PERFECT_SCORE = 0x0064           ;Hexadecimal notation
.EQU PERFECT_SCORE = $64              ;Hexadecimal notation
.EQU PERFECT_SCORE = 0b0110_0100      ;Binary notation
.EQU PERFECT_SCORE = 0144             ;Octal Notation
.EQU PERFECT_SCORE = 'd'              ;ASCII Notation
```

As we have seen before, a constant can be loaded directly into the Registers from R16 to 31. All of the following will load R16 with 100:

```
LDI R16,100
LDI R16,PERFECT_SCORE
LDI R16,(2000+500)/25
LDI R16,$64
LDI R16,0b0110_0100
LDI R16,'d'
LDI A,PERFECT_SCORE  ;if you have defined A = R16
```

THE INCLUDE FILE DIRECTIVE

The AVRs include a large family of chips. To help us produce code for the various processors, ATMEL provides a file for each one that contains a series of standard .DEF and .EQU definitions tailored to that specific chip. For example here is a small clip from the M69DEF.INC file for the ATmega169 processor that is used in the AVR Butterflys that defines the R26-R31 Registers as two-byte pointers called X, Y and Z:

```
; ***** CPU REGISTER DEFINITIONS *************
.def  XH    = r27
.def  XL    = r26
.def  YH    = r29
.def  YL    = r28
.def  ZH    = r31
.def  ZL    = r30
```

The .INCLUDE directive tells the assembler to read in a file as part of our program. For example at the top of a program for the Butterfly you will typically see:

```
.INCLUDE "M169DEF.INC"    ;BUTTERFLY DEFS
```

Or for a program for the ATtiny13:

```
.INCLUDE "TN13DEF.INC"
```

You could even create your own libraries of commonly used routine or constants and include them yourself.

```
.INCLUDE "MYFILE.ASM"
```

INSTALLING THE ASSEMBLER

The best way to learn Assembler is by doing Assembler, so if you have not done it yet, we simply download the Studio 4 Software from Atmel.com and install it in Windows. Last time I downloaded it, I had to register first, which is a simple and quick process. If you are using another operating system, I assume you have already figured out how to install the assembler and programmer. We run a cable from a COM port on your PC to either your programmer or chip, and you are ready to go.

We are going to enter an assembly program called BEEP for the Butterfly. If you are not using the Butterfly, keep reading along we'll get into how to adapt the program for other chips soon enough.

If this is the first time you run the software, select “Atmel AVR Assembler.” Select “Assembler 2” if asked. Enter a file name of BEEP or something similar, then you will be presented with a list of chips for which to assemble for, if using the Butterfly choose ATmega169.

BEEP: OUR FIRST PROGRAM

Here is our first program (A full explanation will follow), if you cut and paste it into the Assembler editor. Then hit the compile button, it will produce a file called BEEP.HEX ready to be programmed into our chip. Then you hit the connect button followed by the program button.

```
;-------------------------------------------------;
; BEEP.ASM for AVR BUTTERFLY                      ;
; MAKE THAT FAMOUS SOUND!                         ;
; AUTHOR: DANIEL J. DOREY       RETRODAN@GMAIL.COM ;
; CREATED: 01-MAR-06            UPDATED: 01-MAR-06 ;
; NOTE: SPEAKER APPEARS TO BE ON PB5 IN SCHEMATICS ;
;-------------------------------------------------;

.INCLUDE "M169DEF.INC" ;(BUTTERFLY DEFINITIONS)

;-----------------------------------------;
; FIRST WE'LL DEFINE SOME REGISTER TO USE ;
;-----------------------------------------;
.DEF A = R16   ;GENERAL PURPOSE ACCUMULATOR
.DEF I = R21   ;INDEXES FOR LOOP CONTROL
.DEF J = R22

.ORG $0000

;-----------------------------------------;
; FIRST WE SETUP A STACK AREA THEN SET    ;
; DIRECTION BIT ON PORT-B FOR OUTPUT/SPKR ;
;-----------------------------------------;
START:
    LDI A,LOW(RAMEND)   ;SETUP STACK POINTER
    OUT SPL,A           ;SO CALLS TO SUBROUTINES
    LDI A,HIGH(RAMEND)  ;WORK CORRECTLY
    OUT SPH,A           ;
    LDI A,0b1111_1111   ;SET ALL PORTB FOR OUTPUT
    OUT DDRB,A          ;WRITE 1s TO DIRECTN REGS

;--------------;
; MAIN ROUTINE ;
;--------------;
BEEP: CLR I
BLUPE:
    SER A               ;TURN SPKR ON
    OUT  PORTB,A
     RCALL PAUSE        ;WAIT
    CLR A               ;TURN IT OFF
    OUT PORTB,A
     RCALL PAUSE        ;WAIT AGAIN
    DEC I
     BRNE BLUPE
LOOP: RJMP LOOP         ;STAY HERE WHEN DONE

;----------------;
;PAUSE ROUTINE   ;
;----------------;
PAUSE:
    CLR J
PLUPE:
    NOP
    DEC J
     BRNE PLUPE
      RET
```

A BUTTERFLY BEEP

If our programming was successful your Butterfly should emit a small beep then stop. If not be sure to check that the program module is sending the correct file, the assembler-editor does not update the program module when switching between programs.

Now we will look at the program in more detail. By now you should understand the first part of the program. It starts with some comments starting with the ";" which the Assembler will ignore. Then we use the .INCLUDE directive to tell the Assembler to read in the file M169DEF.INC which contains definitions for the ATmega169 chip.

```
;-------------------------------------------------;
; BEEP.ASM for AVR BUTTERFLY                      ;
; MAKE THAT FAMOUS SOUND!                         ;
; AUTHOR: DANIEL J. DOREY       RETRODAN@GMAIL.COM ;
; CREATED: 01-MAR-06            UPDATED: 01-MAR-06 ;
; NOTE: SPEAKER APPEARS TO BE ON PB5 IN SCHEMATICS ;
;-------------------------------------------------;

.INCLUDE "M169DEF.INC" ;(BUTTERFLY DEFINITIONS)
```

ADAPTATIONS FOR OTHER AVR CHIPS

If you are using another chip, like the ATtiny13 you would use AT13DEF.INC. If you have done a standard install in Windows XP these files can be found in C:\Program Files\Atmel\AVR Tools\AvrAssembler2\Appnotes if you need to look up the name of the file for the chip you are using.

Next if you are using 3 Volts to power your chip you will need to connect a small speaker to Port B, Pin 5. Check the pin-outs at the top of the data-sheet for your chip which you can download free from Atmel.com. You can use the small PC speaker from an old computer and connect from Port B, Pin 5 to either +3 Volts or to ground. (+3 Volts should be slightly louder). If you are using 5 Volts to power your chip put a small resister in the range of 100 to 220 ohms in series with the speaker to limit the current.

If Port B, Pin 5 is not available, for example on the ATtiny chips it is used as the reset line. Then connect to another pin on Port B like Pin zero and we will change the program later.

Back to our program. Next we give our own names (A,I,J) to the registers we will be using. Note they are all in the range of R16 to R25:

```
;-----------------------------------------;
; FIRST WE'LL DEFINE SOME REGISTER TO USE ;
;-----------------------------------------;
.DEF A = R16   ;GENERAL PURPOSE ACCUMULATOR
.DEF I = R21   ;INDEXES FOR LOOP CONTROL
.DEF J = R22
```

THE ORIGIN DIRECTIVE

Before we can actually create our first lines of code, we have to tell the Assembler, where to put it in RAM. This is done with the Origin Directive ".ORG" and typically AVR programs start at the bottom of memory at location zero because that is where the AVR chip looks when it is fired-up or reset.

```
.ORG $0000
```

The $0000 is just another constant like the ones we discussed earlier so we could have used .ORG 0 or even something crazy like .ORG 100-100 however traditionally it is done with the hexadecimal $ notation.

LABELS

In order to do jumps, loops and subroutines, we need a way to tell the Assembler where to go. Labels are used for this purpose. They are also used to help us understand what the code is doing. For example, since we are at the beginning of our program, let us label it "START:".

Labels typically start at the far left of the screen and the rest of the program is indented. Labels are made of numbers and letters, but must start with a letter and when we define them they end with the colon":" To use a label we do not include the ":". For example to jump to START: we would code:

```
RJMP START
```

THE LOW AND HIGH EXPRESSIONS

Before we can call any subroutines, we need to set-up a memory structure called the stack in memory. Typically our program goes at the bottom-of-memory and the stack goes at the top. Usually programmers set-up the stack at the start of their programs.

Inside the include files (M169DEF.INC in our case) a constant (RAMEND) is defined to the top of memory for us. The problem is that if we have more than 256 bytes of RAM, it is going to be more than we can fit into a single byte. The expressions LOW() and HIGH() will break-down a sixteen-bit number into two bytes for us. LOW(RAMEND) will give us the lower-byte and HIGH(RAMEND) will give us the high-byte. Note that these two bytes are constants, and if we want to write them to the Stack Pointer we must first load them into a register in the R16-R32 range just like any other constant.

THE STACK POINTER

The Stack Pointer is a special two-byte memory location that always points to the top item on the stack. The two memory locations are defined as SPL (low byte) and SPH (high byte).On the AVRs they are treated like a port so the MOV command will not work and we have to use the OUT command. OUT typically send the contents of a register to a port location.

```
START:
    LDI A,LOW(RAMEND)   ;SETUP STACK POINTER
    OUT SPL,A           ;SO CALLS TO SUBROUTINES
    LDI A,HIGH(RAMEND)  ;WORK CORRECTLY
    OUT SPH,A           ;
```

THE DATA DIRECTION REGISTERS

Each input/output port of the AVRs has an associated Data Direction Register which is defined in our include file (M169DEF.INC). For Port B it is DDRB, for Port C it would be DDRC, etc. If we set a bit in this register to one, then that pin becomes an output pin, on-the-other-hand if we write a zero it becomes and input pin.

Since we need Port B, Pin 5 to be an output pin and we are not using any other pins on Port B, we can simply make them all output pins by writing all ones.

Once again we move a constant, this time the value 0b1111_1111 into the R16 Register. Since the Data Direction Registers are Port Registers we cannot use the MOV command, we must use the OUT command again:

```
    LDI A,0b1111_1111   ;SET ALL PORTB FOR OUTPUT
    OUT DDRB,A          ;WRITE 1s TO DIRECTN REGS
```

THE CLR AND SER COMMANDS

Now that all our initializations are done, we get into the main part of our program. Since our program makes a beep on the speaker I have labelled it BEEP, but we could have labelled it MAIN or any name you wish to use.

We don't want our AVR to beep continuously and be a major annoyance, so we are going to set-up a counter to limit how long it will beep. We are going to use register R21 which we have previously given the name “I” and we are going to start it with the value of zero. Normally we might do this by loading it directly with the command LDI I,0 but we are going to use a new command CLR that will set all the bits in a register to zero.

The advantage of the CLR command is that it can be used on ALL the registers from R0 to R31. The opposite of the CLR command is SER (SEt Register) which will set all the bits in a register to one. In fact, previously we used LDI A,0b1111_1111 to set all the bits in Register A to one, but we could have used SER A instead.

```
;--------------;
; MAIN ROUTINE ;
;--------------;
BEEP: CLR I
```

ACTIVATING THE SPEAKER

The way we are going to create a sound coming from the speaker is to activate it then deactivate it by sending out a series of ones and zeros like 10101010101... etc. This will cause the speaker to move in and out and thus create a tone. To do this efficiently we will use a loop, and a loop will require a label, so we call this section of code BLUPE (Beep Loop):

```
BLUPE:
```

To activate the speaker we need to send out a one to PortB, Pin 5. It is over-kill but since we are not using any of the other pins on Port B we can use the SER A command to set ALL the bits to one then send them out to Port B.

```
SER  A              ;TURN SPKR ON
OUT  PORTB,A
```

SUBROUTINES AND THE RCALL & RET COMMANDS

Since the processor runs very fast, in the Butterfly it is 2MHz and the default on many other chips is 1Mhz, unless we slow things down the tone emitted from our speaker will be too high to hear. So we use a subroutine that we call PAUSE that we jump to with the RCALL command that will simply waste some time for us.

The RCALL command saves our spot on the stack then jumps to the subroutine and continues to execute the commands located there. The opposite of the RCALL command is the RET (RETurn) command, that fetches our previous location from the stack so the program can return to where it came from. So every subroutine must end with a RET command. We indent the RCALL PAUSE command to remind us that the program is jumping to another location.

```
 RCALL PAUSE        ;WAIT
```

Previously we activated the speaker by writing all ones to Port B, now we want to do the opposite so we use the CLR command to set all the bits in Register A to zero, then send them out to Port B.

```
CLR A               ;TURN IT OFF
OUT PORTB,A
 RCALL PAUSE        ;WAIT AGAIN
```

THE ADD & SUBTRACT COMMANDS

Previously we set-up Register I as a counter to limit the length of time the speaker will beep. To do this we are going to subtract one from our counter and stop after we have activated & deactivated the speaker 256 times.

To subtract one from Register I we could first LDI A,1 then SUB I,A which will subtract the value in A from the value in I. Similarly we could also ADD I,A if we wanted to add the contents of A to I.

A better way to subtract one from Register I is the SUBtract Immediate command SUBI I,1 that allows us to subtract a constant from a Register. Unfortunately there is not an add immediate command.

An even better way to subtract one from I is to use the DECrement command DEC I. Adding and subtracting one from registers is so common that there are separate commands that do just that. DECrement (DEC I) will subtract one from I and INCrement (INC I) will add one to Register I.

```
DEC I
```

JUMPING & BRANCHING

Typically when we want to jump to a location and not necessarily return we use the RJMP command, but if we only want to jump based on certain criteria, it is commonly called a BRANCH, like the limb of a tree.

There are many branch instructions, if you have not done so yet, you should check out the entire instruction set found in the Data-Sheet for the chip you are using, where you will find a complete list.

We are going to decrement the Register I each time through the main loop and stop when it reaches zero. To do this we are going to use the BRanch if Not Equal to zero instruction (BRNE). So that if the I register has not hit zero yet, then we branch back to the start of our main loop (BLUPE). When the Register I hits zero the program will NOT branch, but will continue to the next instruction and exit the loop.

```
BRNE BLUPE
```

If we want to unconditionally jump to another part of the program, we typically use the Relative JuMP command (RJMP). The RJMP instruction will quickly take us to another part of the program, but it is limited in the distance we can go. If we run into an error because the label we want to jump to is too far, then we can use the slower JMP command without any limits on distance.

Some of you may ask, how is it that we start with zero, then subtract one each time through the loop and still stop at zero? The answer is that when we subtract one from zero the eight-bit register will “roll over” to 255, then we continue to subtract one from 255 until we reach zero again.

Once we have produced our beep sound, we don't need the processor to do anything more, so we send it in an infinite loop by having it jump to itself:

```
LOOP: RJMP LOOP          ;STAY HERE WHEN DONE
```

THE PAUSE ROUTINE

To slow down our program so we can hear the tone emitted, we use a subroutine that does nothing but go in a loop 255 times between activating the speaker and deactivating it. We create this loop by using another Register we have defined as J and decrementing it until it reaches zero, just as we did in our main loop, with the BRanch if Not Equal to zero (BRNE) instruction.

THE NOP INSTRUCTION

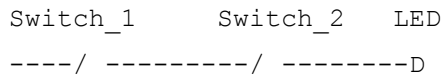
To slow-down our program even more we can insert a command that does nothing but wait for one clock cycle called a No Operation (NOP). If fact we can change the frequency of our tone by adding even more NOPs.

```
;----------------;
;PAUSE ROUTINE   ;
;----------------;
PAUSE:
    CLR J
PLUPE:
    NOP
    DEC J
     BRNE PLUPE
      RET
```

As mentioned earlier, all subroutines must end with a RETurn (RET) instruction.

THE "AND" OPERATION

The "AND" operation can be demonstrated with the following circuit of two switches and a light in series:



It is clear to see that the LED will only illuminate when both switches are closed to produce a complete circuit. Switch one AND switch two both have to be closed before the LED will work. This result can be displayed in a truth table where a zero means off and a one means on:

SW1	SW2	LED
0	0	= 0
0	1	= 0
1	0	= 0
1	1	= 1

The "AND" operation can be used to clear a bit to zero. From the truth table above, you can see that anything that is ANDed with a zero is zero. Lets say you wanted to clear the high bit of a register, the following code will so just that:

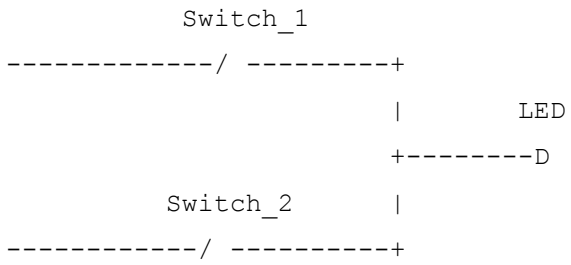
```
LDI  A,0b1111_1111     ;A = 11111111
ANDI A,0b0111_1111     ;A = 01111111
```

"AND" operations can also be used with a "bit-mask" to strip off bits we are interested in. For example if we are only interested in the highest four bits of a byte. We can use the binary number 0b1111_0000 to strip away the high nybble of that register and ignore the remainder:

```
LDI  A,0b1010_1111      ;A = 1111_1111
ANDI A,0b1111_0000      ;A = 1010_0000
```

THE "OR" OPERATION

The "OR" operation can be demonstrated with the following circuit with two switches in parallel connected to a light:



It is clear to see that the LED will light when one "OR" the other switch is closed, and even if both are closed. This can be represented by a truth table:

SW1	SW2	LED
0	0	= 0
0	1	= 1
1	0	= 1
1	1	= 1

The "OR" operation can be used to set a bit to one. From the truth table above, you can see that anything that is ORed with a one is one. Lets say you need to set the high bit of a register, the following code will do that:

```
LDI  A,0b0101_0101     ;A = 0101_0101
ORI  A,0b1000_0000     ;A = 1101_0101
```

THE EXCLUSIVE OR "EOR" OPERATION

The "EOR" operation is the same as the "OR" operation except that it is off when both switches are closed. This means the LED is on if one "OR" the other is on, but not if both are. This can be demonstrated with the following truth table:

```
SW1  SW2  LED
 0    0 = 0
 0    1 = 1
 1    0 = 1
 1    1 = 0
```

If you look at the truth table above, you will see that a one EORed with zero give a one, and a one EORed with a one gives us zero. EORing something with a one gives us the opposite or inverse. This gives us the property of flipping a bit. If you need to "blink" the high bit of a register on and off, the following code will do that without disturbing the other bits of the "A" register:

```
LDI  B,0b1000_0000
LDI  A,0b0101_0101      ;A = 0101_0101
EOR  A,B                ;A = 1101_0101
EOR  A,B                ;A = 0101_0101
EOR  A,B                ;A = 1101_0101
```

AN INVERSE OR "NOT" OPERATION

The NOT or inverse operation means you want the opposite, ones become zero and zeros become one. The truth table for this is:

```
A       NOT_A
0         1
1         0
```

If we think back to the EOR command, we realize that when we EOR something with a one, we flip that bit. So to get the inverse or NOT of an entire value, we can EOR it with all ones:

```
LDI  B,0b1111_1111      ;ALL ONES
LDI  A,0b1010_1010      ;A=1010_1010
EOR A,B                 ;A=0101_0101
```

Previously in the main loop of our program, to make the speaker generate a tone we wrote a one to the speaker port, waited a short time, then wrote a zero, and waited a short time, then repeated the cycle 256 times.

Another way to accomplish a similar result would be to read-in the value at the speaker, and if it is a one, invert it to a zero, and if it is a zero, invert it to a one. This is where the Exclusive OR (EOR) instruction can be used because we now know that anything EORed with ones will give us the inverse or opposite.

In our new version of the main part of our program, we can use the Register J to hold the value of all ones and use it to invert what is on Port B:

```
;--------------;
; MAIN ROUTINE ;
;--------------;
BEEP: CLR I
BLUPE:
    LDI J,0b1111_1111   ;LOAD BITMASK
    IN A,PORTB          ;READ IN PORT B
    EOR A,J             ;INVERT/TOGGLE
    OUT PORTB,A         ;WRITE OUT TO PORT B
     RCALL PAUSE        ;WAIT
    DEC I
     BRNE BLUPE
LOOP: RJMP LOOP         ;STAY HERE WHEN DONE
```

We could even further refine our program by only inverting the pin that the speaker is on. Perhaps we might want to use the other pins on Port B for something other than creating a tone.

On the Butterfly the speaker is connected to Pin 5 of Port B, so we could load the Register J with the value 0b0001_0000 instead.

```
LDI J,0b0001_0000   ;LOAD BITMASK
IN A,PORTB          ;READ IN PORT B
```

THE SBI & CBI INSTRUCTIONS

The Set Bit in I/O Port (SBI) and Clear Bit in I/O Port (CBI) instructions can be used to set or clear bits in an I/O Port that will send a one or zero out on the corresponding pin. For example we could use them in the main loop of our BEEP program to activate the speaker:

```
;--------------;
; MAIN ROUTINE ;
;--------------;
BEEP: CLR I
BLUPE:
    SBI  PORTB,5        ;ACTIVATE THE SPEAKER
     RCALL PAUSE        ;WAIT
    CBI PORTB,5         ;SHUT OFF SPEAKER
     RCALL PAUSE        ;WAIT AGAIN
    DEC I
     BRNE BLUPE
LOOP: RJMP LOOP         ;STAY HERE WHEN DONE
```

The Data-Sheet tells us that we can Toggle an output pin by writing a one to the input pin (PINx). Each output port has an associated input register, for PORTB it would be PINB. We can simplify toggling the speaker:

```
;--------------;
; MAIN ROUTINE ;
;--------------;
BEEP: CLR I
BLUPE:
    SBI PINB,5          ;TOGGLE SPEAKER
     RCALL PAUSE        ;WAIT
    DEC I
     BRNE BLUPE
LOOP: RJMP LOOP         ;STAY HERE WHEN DONE
```

THE STACK

The Stack is a memory structure that is like a stack of plates. You can only ever place new plates on the top of the stack. For example if there were three items on the stack and you added a fourth:

```
  [3]              [4]  <-- NEW
  [2]     ==>      [3]
  [1]              [2]
[STACK]            [1]
                 [STACK]
```

Now if we remove [4] from the top of the stack, then [3] become the top of the stack again. Then if we remove [3] then [2] is on the top:

```
  [4]              [3]           [2]
  [3]     ==>      [2]    ==>    [1]
  [2]              [1]         [STACK]
  [1]            [STACK]
[STACK]
```

THE PUSH & POP INSTRUCTIONS

The PUSH instruction copies a register to the top of the stack, and POP copies the top of the stack into a register.

One common use is to preserve the value of a Register. For example if we wanted our BEEP program to use only Register A in both the main the loop of our program and in the PAUSE subroutine. We could push A on the stack before we call the PAUSE routine then pop it off the stack after we come back:

```
;--------------;
; MAIN ROUTINE ;
;--------------;
BEEP:  CLR A              ;USE A AS COUNTER
BLUPE: SBI PINB,5         ;TOGGLE SPEAKER
       PUSH A             ;SAVE CONTENTS OF A
       RCALL PAUSE        ;WAIT
       POP A              ;RESTORE A
       DEC A
        BRNE BLUPE
LOOP:    RJMP LOOP        ;STAY HERE WHEN DONE

PAUSE: CLR A
PLUPE: NOP
       DEC A
        BRNE PLUPE
         RET
```

An even better way to do this would be to place the PUSH & POP instructions inside the PAUSE subroutine. The first thing we do is save A on the stack before its value gets changed, then we restore it just before we return. This makes the PAUSE routine reusable and portable because it saves & restores the value of the register it uses:

```
PAUSE: PUSH A         ;SAVE CONTENTS OF A
       CLR A
PLUPE: NOP
       DEC A
        BRNE PLUPE
       POP A          ;RESSTORE A
        RET
```

Now that we have a basic understanding of the AVR Assembler Language it will be easier for you to follow more advanced tutorials. Another great way to learn is to look at others working code and observe the techniques they use. I encourage you to take some working code and play around with it because the best way to learn Assembler is to code in Assembler, learn by doing.

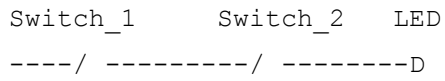
(Included is a Tutorial on Bit Manipulation)

A MORON'S GUIDE TO BIT MANIPULATION v1.6

by RetroDan@GMail.com

THE "AND" OPERATION

The "AND" operation can be demonstrated with the following circuit of two switches and a light in series:



It is clear to see that the LED will only illuminate when both switches are closed to produce a complete circuit. Switch one AND switch two both have to be closed before the LED will work. This result can be displayed in a truth table where a zero means off and a one means on:

SW1	SW2	LED
0	0	= 0
0	1	= 0
1	0	= 0
1	1	= 1

The "AND" operation can be used to clear a bit to zero. From the truth table above, you can see that anything that is ANDed with a zero is zero. Lets say you wanted to clear the high bit of a register, the following code will so just that:

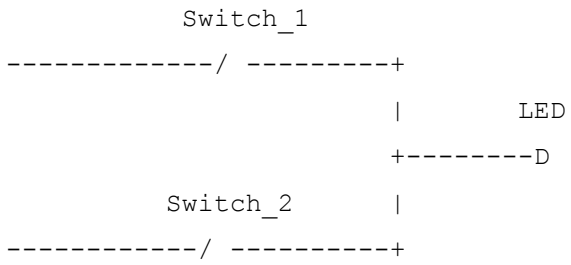
```
LDI  A,0b1111_1111     ;A = 11111111
ANDI A,0b0111_1111     ;A = 01111111
```

"AND" operations can also be used with a "bit-mask" to strip off bits we are interested in. For example if we are only interested in the highest four bits of a byte. We can use the binary number 0b1111_0000 to strip away the high nybble of that register and ignore the remainder:

```
LDI  A,0b1010_1111      ;A = 1111_1111
ANDI A,0b1111_0000      ;A = 1010_0000
```

THE "OR" OPERATION

The "OR" operation can be demonstrated with the following circuit with two switches in parallel connected to a light:



It is clear to see that the LED will light when one "OR" the other switch is closed, and even if both are closed. This can be represented by a truth table:

SW1	SW2	LED
0	0	= 0
0	1	= 1
1	0	= 1
1	1	= 1

The "OR" operation can be used to set a bit to one. From the truth table above, you can see that anything that is ORed with a one is one. Lets say you need to set the high bit of a register, the following code will do that:

```
LDI  A,0b0101_0101     ;A = 0101_0101
ORI  A,0b1000_0000     ;A = 1101_0101
```

THE EXCLUSIVE OR "EOR" OPERATION

The "EOR" operation is the same as the "OR" operation except that it is off when both switches are closed. This means the LED is on if one "OR" the other is on, but not if both are. This can be demonstrated with the following truth table:

```
SW1  SW2  LED
 0    0 = 0
 0    1 = 1
 1    0 = 1
 1    1 = 0
```

If you look at the truth table above, you will see that a one EORed with zero give a one, and a one EORed with a one gives us zero. EORing something with a one gives us the opposite or inverse. This gives us the property of flipping a bit. If you need to "blink" the high bit of a register on and off, the following code will do that without disturbing the other bits of the "A" register:

```
LDI  B,0b1000_0000
LDI  A,0b0101_0101      ;A = 0101_0101
EOR  A,B                ;A = 1101_0101
EOR  A,B                ;A = 0101_0101
EOR  A,B                ;A = 1101_0101
```

THE "NOT" OPERATION

The NOT or inverse operation means you want the opposite, ones become zero and zeros become one. The truth table for this is:

```
A       NOT_A
0         1
1         0
```

If we think back to the EOR command, we realize that when we EOR something with a one, we flip that bit. So to get the inverse or NOT of an entire value, we can EOR it with all ones:

```
LDI  B,0b1111_1111      ;ALL ONES
LDI  A,0b1010_1010      ;A=1010_1010
EOR A,B                 ;A=0101_0101
```

To do this with a constant we can use the negation operator ~ (tilde):

```
 LDI A,~0b1010_1010     ;A=0101_0101
```

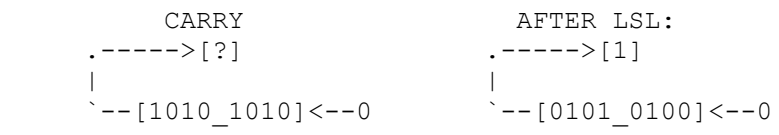
THE LEFT SHIFT OPERATIONS

The left-shift operation moves bits in a register, one place to the left. Why would anyone want to do this? Lets look at the example of left-shifting the value of three:

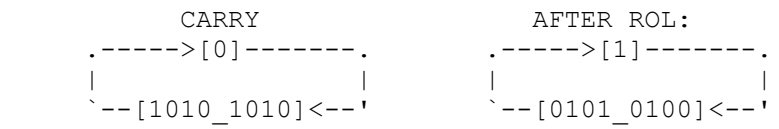
```
LDI A,3         ;A=0b0000_0011=3
LSL A           ;A=0b0000_0110=6
LSL A           ;A=0b0000_1100=12
```

Since computers run on a Binary System, each time you shift a value one bit to the left, you are multiplying that value by two. If you wanted to multiply something by eight, you would left-shift it three times. What the Logical Shift Left command (LSL) does is shift the highest bit out into the carry flag, shift the contents of the register to the left one bit and shifts a zero into the lowest bit. The result is a multiplication by two:

```
CARRY    A-REGISTER
 [?]     [0101_0101]      BEFORE LEFT SHIFT VALUE = 85
 [0]  << [1010_1010] <<0  AFTER  LEFT SHIFT VALUE = 85x2 = 170
```



What if the highest bit was a one? If you are working with more than eight bits you can use the rotate-left command ROL which rotates the value stored in the carry bit into the lowest bit and then loads the carry flag with the value that was shifted out the high end:



In the example above the zero that was in the carry flag is shifted into the low end of the register, the remaining bits are shifted one to the left and the high bit is shifted out and into the carry flag.

LSL always shifts a zero into the lowest bit, while ROL shifts the contents of the carry flag into the lowest bit. Using both we can multiply numbers larger than one byte.

To multiply a sixteen bit number by two, you first LSL the lower byte, then ROL the high byte, this has the net effect of "rolling" the high bit of the lower byte into the first bit of the 2nd byte. This technique can be expanded to multiply even larger numbers.

Multiplying a 32-bit number by two:

```
LSL  A1   ;MULTIPLY VALUE IN A1 BY TWO
ROL  A2   ;VALUE SHIFTED OUT OF A1 GETS SHIFTED INTO A2
ROL  A3   ;VALUE SHIFTED OUT OF A2 GETS SHIFTED INTO A3
ROL  A4   ;VALUE SHIFTED OUT OF A3 GETS SHIFTED INTO A4
```

RIGHT SHIFT OPERATIONS

The right-shift operation moves bits in a register, one place to the right. Why would anyone want to do this? Lets look at the example of right-shifting the value of twelve:

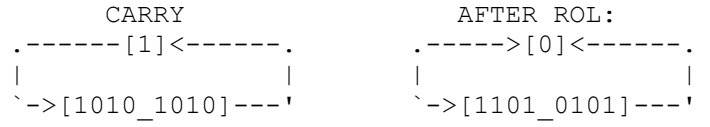
```
LDI  A,12     ;A = 0b0000_1100 = 12
LSR  A        ;A = 0b0000_0110 = 6
LSR  A        ;A = 0b0000_0011 = 3
```

Since computers run on a Binary System, each time you shift a value one bit to the right, you are dividing that value by two. If you wanted to divide something by eight, you would right-shift it three times. What the Logical Shift Right command (LSL) does is shift a zero into the highest bit, shift the contents to the right and shifts the lowest bit into the carry flag.

The result is a division by two:

```
   A-REGISTER  CARRY
   [1010_1010]  [?]  BEFORE RIGHT SHIFT VALUE = 170
0>>[0101_0101]>>[0]  AFTER  RIGHT SHIFT VALUE = 170/2 = 85
```

If you are working with more than eight bits and you wish to preserve the value of the lowest bit, you can use the rotate-right command ROR which rotates the value stored in the carry bit into the highest bit and then loads the carry flag with the value that was shifted out the low end.



In the example above the one that was in the carry flag is shifted into the low end of the register, the remaining bits are shifted one to the right and the low bit is shifted out and into the carry flag.

LSR always shifts a zero into the highest bit, while ROR shifts the contents of the carry flag into the highest bit. Using both we can divide numbers larger than one byte.

To divide a sixteen bit number by two, you first LSR the highest byte, then ROR the lower byte, this has the net effect of "rolling" the low bit of the high byte into the high bit of the lower byte. This technique can be expanded to divide even larger numbers.

Dividing a 32-bit number by two:

```
LSR  A4   ;DIVIDE VALUE IN A4 BY TWO
ROL  A3   ;VALUE SHIFTED OUT OF A4 GETS SHIFTED INTO A3
ROL  A2   ;VALUE SHIFTED OUT OF A3 GETS SHIFTED INTO A2
ROL  A1   ;VALUE SHIFTED OUT OF A2 GETS SHIFTED INTO A1
```

THE "SWAP" COMMAND

The swap command exchanges the two nybbles of a byte. It exchanges the high four bits with the lower four bits:

```
[1111_0000] ---> SWAP ---> [0000_1111]
```

```
SWAP A    ;SWAPS HIGH NYBBLE IN "A" WITH LOW NYBBLE
```

For example if you want to isolate the high byte of register A:

```
ANDI A,0b1111_0000  ;MASK OFF HIGH NYBBLE
LSR  A              ;RIGHT SHIFT = /16
LSR  A
LSR  A
LSR  A
```

Or you could use the SWAP statement as follows:

```
ANDI A,0b1111_0000
SWAP A
```

THE SBI/CBI COMMANDS

The SBI & CBI commands can be used to set or clear bits in an output port respectively. For example to set PORTB0 for output we need to set the zero-bit of the Data Direction register for Port B to one:

```
SBI DDRB,0
```

To set PORTB0 for input we need to clear bit zero:

```
CBI DDRB,0
```

The number following the register DDRB is the bit position to set or clear, it is not the value to write to the port. For example to set bit seven the command would be:

```
SBI DDRB,7      ;CORRECT
```

and not:

```
SBI DDRB,128    ;WRONG!!!
```

THE SBIS/SBIC COMMANDS

The SBIS and SBIC commands are used to branch based on if a bit is set or cleared. SBIS will skip the next command if a selected bit in a port is set, and SBIC will skip the next command if the bit is clear. For example if er are waiting for an ADC to complete, we poll the ADIF Flag of the ADCSRA register to. We could read in the register and check if the flag has been set with:

```
WAIT:  IN    A,ADCSRA        ;READ THE STATUS
       ANDI  A,0b0001_0000   ;CHECK ADIF FLAG
          BREQ WAIT
```

Or we can accomplish the same with the SBIS command that checks the status of the 4th bit without the need to read the port into a register:

```
WAIT: SBIS  ADCSRA,4
       RJMP WAIT
```

The SBIC command is identical, except it checks if a bit is zero.

THE SBRS/SBRC COMMANDS

The SBRS and SBRC commands function the same as the SBIS/SBIC commands, except that they work on registers instead of ports.