Introduction

All of these examples were created by using C++ files, running them through the avr-gcc compiler, and then inspecting the .s files it generated. Please note that most, if not all, of these examples have been modified from their original output. The compiler tends to produce obscure code. The examples are snippets of the full .s files and have been changed slightly to illustrate what assembly code is generated from its equivalent C++ code.

In the same directory as this document are the source .cpp files and their .s files. If you want to try one of the examples, run `make filename.hex` – where *filename* is any .cpp file in the directory. That will generate the files needed, including the .s file. Please note that the generated .s file will not match the example .s files; the reason is explained above. To upload the program onto the MeggyJr device, run `make filename.install`

The examples in this document are just snippets and will not compile if you copy/paste them. Use the associated file if you want to see the example in action.

Background

The AVR instruction set has three special variables, X, Y, and Z. Each of them corresponds with two registers. X represents r26 and r27, Y represents r28 and r29, and Z represents r30 and r31.

The most important one, when dealing with variables, is Y. Y references the top of the stack. AVR automatically handles the stack for you. To reference a variable on the stack, you simply access by using Y+*n* where *n* is the bytes to offset from the top of the stack. The first variable is at Y+1.

Registers are 8-bits in size. However, AVR uses 16-bit addresses when accessing variables. Often times, two registers are used in conjunction to store one 16-bit value. An example is `r28` and `r28`. `r28` stores the lower 8 bits, and `r29` stores the higher 8 bits. The two registers are used together in functions when a 16-bit value is expected. Registers use in this manner are written as `r`*`hi`*`:r`*`lo`*, where `r`*`hi`* is the high-bits register, and `r`*`lo`* is the low-bits register.

The AVR instruction set has two commonly used functions called `hi8()` and `lo8()`. `hi8()` takes the high 8 bits of a number and returns them. `lo8()` takes the low 8 bits and returns those. These functions can be used anywhere in the assembly language where an immediate value would normally be.

It is important to have an infinite loop in the code to keep the MeggyJr running. If you are doing one thing, you can add `while(1);` to the end of your main function. The infinite loop is required to make the MeggyJr continue running. Not all of the example snippets in this document contain an infinite loop.

MeggyJr Functions

Creating a MeggyJr Object

All of the functions needed to interact with the MeggyJr hardware are in *MeggyJr.h*. A simplified library with utility functions is available in *MeggyJrSimple.h*. *MeggyJrSimple.h* uses *MeggyJr.h* to perform all of its functions, so we can just use *MeggyJr.*h and derive from it.

To create a MeggyJr object, simply do the following:

C Code:

```
#include <MeggyJr.h>
// …
MeggyJr mj;
```

Assembly:

```
movw r24,r28         ; Store Y in r24/r25
adiw r24,1           ; Move to first variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object
```

When dealing with object method calls in assembly, the address of the object needs to be stored in `r25:r24`. In this example, the object `mj` isn't created yet, so the value `r25:r24` points to will be zero (null).

Setting a Pixel - MeggyJr.SetPxClr()

The MeggyJr uses RGB for its color scheme. Each color channel can be a value from 0 to 15; 0 being off and 15 being full on. Setting all of the channels to 0 will turn the pixel off. The following code sets a pixel to red:
(See mj_setPixel.cpp and mj_setPixel.s)

C Code:

```
#include <MeggyJr.h>
// …
MeggyJr mj;
byte red[3] = {5, 0, 0};      // Red, Green, Blue Amounts
mj.SetPxClr(3, 4, red);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28         ; Store Y in r24/r25
adiw r24,1           ; Move to first variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object
```

```
; byte red[3] = {5, 0, 0};
ldi r24,lo8(5)
std Y+2,r24          ; Store 5
ldi r24,lo8(0)
std Y+3,r24          ; Store 0
ldi r24,lo8(0)
std Y+4,r24          ; Store 0

; mj.SetPxClr(3, 4, red);
movw r24,r28         ; Load Y so we can pass mj to the function
adiw r24,1           ; 1 is where the address of mj is stored
ldi r22,lo8(3)       ; Store X (3) in r23:r22
ldi r20,lo8(4)       ; Store Y (4) in r21:r20
movw r18,r28         ; Get Y (top of stack) so we can get red[], r19:r18 needs to
                     ; be the address of the array of color values.
subi r18,lo8(-2)     ; Add 2, this moves the address to the first item in red[]
sbci r19,hi8(-2)
call _ZN7MeggyJr8SetPxClrEhhPh
```

There are a couple things to note about this code. First, arrays are stored as consecutive variables on the stack. In this example, the array starts at Y+2, which is 5. Items in the array can be accessed by simply adding their index to the starting place of the array. To get `red[2]`, you would use Y+4 (starts at 2 + index 2).

Second, pointers for any variable can be obtained through the `r29:r28` (Y) registers. To do this, just copy with `movw` the value stored in `r29:r28` to another register, and add a numerical value to it. Since `r29:r28` contains the address to the top of the stack, adding a number to it moves the address to a variable.

The MeggyJr accepts three arguments when setting a pixel color; X position, Y position, and a pointer to an array of 3 bytes. The 3 bytes are the color values for each channel, red, green, and blue (RGB in that order!). X position needs to be stored in `r23:r22`, Y position needs to be stored in `r21:r20`, and the pointer needs to be stored in `r19:r18`. The name of the function to call is `_ZN7MeggyJr8SetPxClrEhhPh`.

Getting a Pixel – MeggyJr.GetPixelR(), GetPixelG(), and GetPixelB()

There are three functions that the MeggyJr uses to get a pixel from the screen, one for each color channel. They are `GetPixelR()`, `GetPixelG()`, and `GetPixelB()`. The last letter of the function name tells what color channel it returns; R for Red, G for Green, and B for Blue. This example lights up an LED at (2, 2), gets the color values of that LED, and then lights up another LED at (5, 5) with those retrieved values.

(See mj_getPixel.cpp and mj_getPixel.s)

C Code:

```
#include <MeggyJr.h>
// …
MeggyJr mj;
byte color[3] = {3, 4, 5};
mj.SetPxClr(2, 2, color);
byte r = mj.GetPixelR(2, 2);
byte g = mj.GetPixelG(2, 2);
byte b = mj.GetPixelB(2, 2);
color[0] = r;
color[1] = g;
color[2] = b;
mj.SetPxClr(5, 5, color);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28         ; Store Y in r24/r25
adiw r24,1           ; Move to first variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object

; byte color[3] = {3, 4, 5};
ldi r24,lo8(3)
std Y+2,r24          ; Store 3
ldi r24,lo8(4)
std Y+3,r24          ; Store 4
ldi r24,lo8(5)
std Y+4,r24          ; Store 5

; mj.SetPxClr(2, 2, color);
movw r24,r28         ; Load Y so we can pass mj to the function
adiw r24,1           ; 1 is where the address of mj is stored
ldi r22,lo8(2)       ; Store X (2) in r23:r22
ldi r20,lo8(2)       ; Store Y (2) in r21:r20
movw r18,r28         ; Get Y (top of stack) so we can get color[], r19:r18 needs
                     ; to be the address of the array of color values.
subi r18,lo8(-2)     ; Add 2, this moves the address to the first item in color[]
sbci r19,hi8(-2)
call _ZN7MeggyJr8SetPxClrEhhPh

; byte r = mj.GetPixelR(2, 2);
movw r24,r28         ; Load Y
adiw r24,1           ; Move to mj
ldi r22,lo8(2)       ; Put the X value (2) of the pixel we want to get in r22
ldi r20,lo8(2)       ; Put the Y value (2) of the pixel we want in r20
call _ZN7MeggyJr9GetPixelREhh
std Y+5,r24          ; Put the result in r
```

```
; byte g = mj.GetPixelG(2, 2);
movw r24,r28          ; Load Y
adiw r24,1            ; Move to mj
ldi r22,lo8(2)        ; Put the X value (2) of the pixel we want to get in r22
ldi r20,lo8(2)        ; Put the Y value (2) of the pixel we want in r20
call _ZN7MeggyJr9GetPixelGEhh
std Y+6,r24           ; Put the result in g

; byte b = mj.GetPixelB(2, 2);
movw r24,r28          ; Load Y
adiw r24,1            ; Move to mj
ldi r22,lo8(2)        ; Put the X value (2) of the pixel we want to get in r22
ldi r20,lo8(2)        ; Put the Y value (2) of the pixel we want in r20
call _ZN7MeggyJr9GetPixelBEhh
std Y+7,r24           ; Put the result in b

; color[0] = r; color[1] = g; color[2] = b;
ldd r24,Y+5           ; Load r
std Y+2,r24           ; Store it in color[0]
ldd r24,Y+6           ; Load g
std Y+3,r24           ; Store it in color[1]
ldd r24,Y+7           ; Load b
std Y+4,r24           ; Store it in color[2]

; mj.SetPxClr(5, 5, color);
movw r24,r28          ; Load Y so we can pass mj to the function
adiw r24,1            ; 1 is where the address of mj is stored
ldi r22,lo8(5)        ; Store X (5) in r23:r22
ldi r20,lo8(5)        ; Store Y (5) in r21:r20
movw r18,r28          ; Get Y (top of stack) so we can get color[], r19:r18 needs
                      ; to be the address of the array of color values.
subi r18,lo8(-2)      ; Add 2, this moves the address to the first item in color[]
sbci r19,hi8(-2)
call _ZN7MeggyJr8SetPxClrEhhPh
```

The names of the functions to get the red, green, and blue channels are `_ZN7MeggyJr9GetPixelREhh`, `_ZN7MeggyJr9GetPixelGEhh`, and `_ZN7MeggyJr9GetPixelBEhh` respectively. Each of these functions returns a byte, which is 0 to 15.

Playing a Tone – MeggyJr.StartTone()

Playing a tone is much easier in recent MeggyJr code. There are two pieces of information needed to play a tone. First is the frequency, which is based off of 8 MHz. This means that providing it 18182, the tone produced will be an A_4 – 400 Hz (8 MHz / 18182 = 400 Hz). The second parameter is the duration of the tone, in milliseconds. Both of these numbers are unsigned 16-bit integers. The tone function is non-blocking. After the call is executed, your code will continue to run. You can use the static variable `MeggyJr.ToneTimeRemaining` to check if the tone is playing. The

tone will automatically be stopped by the hardware after the duration of the tone is done. At the end of this document is a table that lists the values of common notes. Setting the duration to 0 will make the tone infinite. The following code plays an A_4 note for one second:
(See mj_tone.cpp and mj_tone.s)

C Code:

```
#include <MeggyJr.h>
// …
MeggyJr mj;
mj.StartTone(18182, 1000);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28         ; Store Y in r24/r25
adiw r24,1           ; Move to first variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object

; mj.StartTone(18182, 1000);
movw r24,r28         ; Load Y so we can pass mj to the function
adiw r24,1           ; 1 is where the address of mj is stored
ldi r18,lo8(18182)   ; Store low bits of tone into r18
ldi r19,hi8(18182)   ; Store high bits of tone into r19
ldi r20,lo8(1000)    ; Store low bits of duration into r20
ldi r21,hi8(1000)    ; Store high bits of duration into r21
call _ZN7MeggyJr9StartToneEjj
```

The tone to play is stored in `r19:r18`, and the duration is stored in `r21:r20`. The name of the function to call is `_ZN7MeggyJr9StartToneEjj`.

Testing when a Tone is Done – MeggyJr.ToneTimeRemaining

The MeggyJr object has a static variable named `ToneTimeRemaining`, which is an unsigned 16-bit integer. It is the number of milliseconds remaining until the tone that is currently playing is done. A simple if-statement can be used to check if the tone is done, which is useful for playing multi-tone sounds. This example plays 2 one second tones:
(See mj_toneTime.cpp and mj_toneTime.s)

C Code:

```
#include <MeggyJr.h>
// …
MeggyJr mj;
mj.StartTone(18182, 1000);      // A4
unsigned int time;
```

```
do    {
   time = MeggyJr::ToneTimeRemaining;
}     while(time);            // Wait for tone to finish
mj.StartTone(15290, 1000);    // C5
while(1);                     // Keep the MeggyJr running
```

Assembly:

```
; MeggyJr mj;
movw r24,r28         ; Store Y in r24/r25
adiw r24,1           ; Move to first variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object

; mj.StartTone(18182, 1000);
movw r24,r28         ; Load Y so we can pass mj to the function
adiw r24,1           ; 1 is where the address of mj is stored
ldi r22,lo8(18182)   ; Store low bits of tone into r18
ldi r23,hi8(18182)   ; Store high bits of tone into r19
ldi r20,lo8(1000)    ; Store low bits of duration into r20
ldi r21,hi8(1000)    ; Store high bits of duration into r21
call _ZN7MeggyJr9StartToneEjj

; do {
.L2:

; time = MeggyJr::ToneTimeRemaining;
lds r24,_ZN7MeggyJr17ToneTimeRemainingE
lds r25,_ZN7MeggyJr17ToneTimeRemainingE+1
std Y+2,r24          ; Store the value retrieved into time
std Y+3,r25

; } while(time);
ldd r24,Y+2          ; Load time from the stack
ldd r25,Y+3
sbiw r24,0           ; This resets the 0 flag
brne .L2             ; Do the loop again if time is not zero

; mj.StartTone(15290, 1000);
movw r24,r28         ; Load Y so we can pass mj to the function
adiw r24,1           ; 1 is where the address of mj is stored
ldi r22,lo8(15290)   ; Store low bits of tone into r18
ldi r23,hi8(15290)   ; Store high bits of tone into r19
ldi r20,lo8(1000)    ; Store low bits of duration into r20
ldi r21,hi8(1000)    ; Store high bits of duration into r21
call _ZN7MeggyJr9StartToneEjj

; while(1);
.L3:
rjmp .L3
```

The location that `MeggyJr::ToneTimeRemaining` is stored in is `ZN7MeggyJr17ToneTimeRemainingE` and takes up two bytes. You must use `lds` to load the value because it is stored in data space instead of the stack.

Getting Button Presses – MeggyJr.GetButtons()

Getting the buttons pressed can be done by calling a function. The function returns a byte, each bit corresponds to a different button on the MeggyJr. At the end of this document is a table with each button and its bit value. The following is an example of using GetButtons. When a button is pressed, it lights up a green pixel.
(See mj_buttons.cpp and mj_buttons.s)

C Code:

```
#include <MeggyJr.h>
// …
byte Green[3] = {0, 5, 0};
MeggyJr mj;
while(1)    {
      byte buttons = mj.GetButtons();
      if(buttons)
            mj.SetPxClr(5, 5, Green);
}
```

Assembly:

```
; byte Green[3] = {0, 5, 0};
ldi r24,lo8(0)
std Y+1,r24          ; Store 0
ldi r24,lo8(5)
std Y+2,r24          ; Store 5
ldi r24,lo8(0)
std Y+3,r24          ; Store 0

; MeggyJr mj;
movw r24,r28         ; Store Y in r24/r25
adiw r24,4           ; Move to fourth variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object

; while(1) {
.L3:

; byte buttons = mj.GetButtons();
movw r24,r28         ; Load Y so we can pass mj to the function
adiw r24,4           ; 4 is where the address of mj is stored
call _ZN7MeggyJr10GetButtonsEv
```

```
std Y+5,r24          ; The button states are stored in r24 after the call,
                     ; copy it to Y+5 on the stack (byte buttons)

; if(buttons)
ldd r24,Y+5          ; Load the buttons variable
tst r24              ; Test if the value is non-zero
breq .L3             ; If it is zero, skip to the next iteration (skips SetPxClr)

; mj.SetPxClr(5, 5, Green);
movw r24,r28         ; Load Y so we can pass mj to the function
adiw r24,4           ; 4 is where the address of mj is stored
ldi r22,lo8(5)       ; Store X (5) in r23:r22
ldi r20,lo8(5)       ; Store Y (5) in r21:r20
movw r18,r28         ; Get Y (top of stack) so we can get Green[], r19:r18 needs
                     ; to be the address of the array of color values.
subi r18,lo8(-1)     ; Add 1, this moves the address to the first item in Green[]
sbci r19,hi8(-1)
call _ZN7MeggyJr8SetPxClrEhhPh

; }
rjmp .L3             ; Run the loop again
```

The important part to note is that the value that contains each button's state is stored in `r24` after the call to `_ZN7MeggyJr10GetButtonsEv`. If you want to do anything with that value, you could put it on the stack, or use it immediately in a conditional.

Setting Aux Lights – MeggyJr.AuxLEDs

The MeggyJr has a static variable that can be updated to set the auxiliary LEDs (the ones on top). That variable's name is `MeggyJr::AuxLEDs`. There are 8 LEDs, and the size of the variable is 8 bits. That means that each bit corresponds to a different LED. The left-most LED has a value of 1, and the right-most LED has a value of 128. There is a strange quirk with the auxiliary LEDs. To update them, you must play a tone. You can "cheat" by playing a tone for 1 ms, since 0 will not work. This example sets the odd-numbered LEDs to on. The value to do that is Binary: 10101010, Hex: AA, and Decimal: -86.
(See mj_auxLEDs.cpp and mj_auxLEDs.s)

C Code:

```
#include <MeggyJr.h>
// …
MeggyJr mj;
MeggyJr::AuxLEDs = 0xAA;
mj.StartTone(0, 1);     // Play a very short tone to force the
                        // AuxLEDs to update
```

Assembly:

```
; MeggyJr mj;
movw r24,r28                ; Store Y in r24/r25
adiw r24,1                  ; Move to variable #1 (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev        ; Initializes the MeggyJr object

; MeggyJr::AuxLEDs = 0xAA;
ldi r24,lo8(-86)            ; Load 0xAA into a register
sts _ZN7MeggyJr7AuxLEDsE,r24 ; Store the value into the AuxLEDs variable

; mj.StartTone(0, 1);
movw r24,r28
adiw r24,1
ldi r22,lo8(0)
ldi r23,hi8(0)
ldi r20,lo8(1)
ldi r21,hi8(1)
call _ZN7MeggyJr9StartToneEjj

; while(1);
.L2:
rjmp .L2
```

The name of the variable to store the auxiliary LED data is `_ZN7MeggyJr7AuxLEDsE`. Since it is in the data memory, instead of the stack, you must use `sts` to write to it. The value can also be retrieved by using `lds`.

C/C++ to AVR Instruction Set Examples

Assignment by Value

To assign a value to a variable, you first store it in a temporary register. The register commonly used is `r24`. From there, the value in the register is copied onto the stack. Registers `r25:r24` are used when returning a value from a function.

C Code:

```
byte x = 5;
```

Assembly:

```
ldi r24,lo8(5)  ; Store 5 in temp register
std Y+1,r24     ; Put the contents of the register (5) into x (Y+1 on the stack)
```

Multiple assignments would look like this:

C Code:

```
byte x = 5;
byte y = 10;
```

Assembly:

```
ldi r24,lo8(5) ; Store 5 in temp register
std Y+1,r24    ; Put the contents of the register (5) into x (Y+1 on the stack)
ldi r24,lo(10) ; Store 10 in a temp register
std Y+2,r24    ; Put the contents of the register (10) into y (Y+2 on the stack)
```

The following MeggyJr example stores the x and y coordinates of a pixel, and then sets the color at that pixel to blue.
(See mj_assign.cpp and mj_assign.s)

C Code:

```
MeggyJr mj;
byte x = 1;
byte y = 2;
byte Blue[3] = {MeggyBlue}; // Same as 'byte Blue[3] = {0, 0, 5};'
mj.SetPxClr(x, y, Blue);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28         ; Store Y in r24/r25
adiw r24,1           ; Move to first variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object

; byte x = 1;
ldi r24,lo8(1)       ; Put 1 in a temp register
std Y+2,r24          ; Put the contents of r24 on the stack (at Y+2)

; byte y = 2;
ldi r24,lo8(2)       ; Put 2 in a temp register
std Y+3,r24          ; Put the contents of r25 on the stack (at Y+3)

; byte Blue[3] = {MeggyBlue};
ldi r24,lo8(0)
std Y+4,r24          ; Store 0
ldi r24,lo8(0)
std Y+5,r24          ; Store 0
ldi r24,lo8(5)
std Y+6,r24          ; Store 5
```

```
; mj.SetPxClr(x, y, Blue);
movw r24,r28          ; Load Y so we can pass mj to the function
adiw r24,1            ; 1 is where the address of mj is stored
ldd r22,Y+2           ; Store X (Y+2) in r23:r22
ldd r20,Y+3           ; Store Y (Y+3) in r21:r20
movw r18,r28          ; Get Y (top of stack) so we can get Blue[], r19:r18 needs to
                      ; be the address of the array of color values.
subi r18,lo8(-4)      ; Add 4.  This moves the address to the first item in Blue[]
sbci r19,hi8(-4)
call _ZN7MeggyJr8SetPxClrEhhPh
```

Assignment by Variable

Assigning a variable to another is as simple as assignment by value was. The first thing you need to do is load the source variable into a register. Use `ldd` instead of `ldi`. Then store it into the destination variable on the stack.

C Code:

```
byte x = 7;
byte y = x;
```

Assembly:

```
ldi r24,lo8(7)  ; Store 7 in temp register
std Y+1,r24     ; Put the contents of the register (7) into x (Y+1 on the stack)
ldd r24,Y+1     ; Load the variable x into a register
std Y+2,r24     ; Store what was in x into y (Y+2 on the stack)
```

You could also chain assignments like this:

C Code:

```
byte x = 20;
byte y, z;
z = y = x;
```

Assembly:

```
ldi r24,lo8(20) ; Store 20 in temp register
std Y+1,r24     ; Put the contents of the register (20) into x (Y+1 on the stack)
ldd r24,Y+1     ; Load x into a temp register
std Y+2,r24     ; Store what was in x into y (Y+2 on the stack)
std Y+3,r24     ; Store what was in x into z (Y+3 on the stack)
```

The following MeggyJr example stores the x and y coordinates of a pixel, and then sets the color at that pixel to blue, using the values in x and y for the red and green channels.
(See mj_assign2.cpp and mj_assign2.s)

C Code:

```
MeggyJr mj;
byte x = 1;
byte y = 2;
byte Blue[3] = {x, y, 7}; // 1, 2, 7
mj.SetPxClr(x, y, Blue);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28          ; Store Y in r24/r25
adiw r24,1            ; Move to first variable (contains the MeggyJr object)
call _ZN7MeggyJrC1Ev ; Initializes the MeggyJr object

; byte x = 1;
ldi r24,lo8(1)        ; Put 1 in a temp register
std Y+2,r24           ; Put the contents of r24 on the stack (at Y+2)

; byte y = 2;
ldi r24,lo8(2)        ; Put 2 in a temp register
std Y+3,r24           ; Put the contents of r25 on the stack (at Y+3)

; byte Blue[3] = {x, y, 7};
ldd r24,Y+2           ; Load x
std Y+4,r24           ; Store x (1) into Blue[0]
ldd r24,Y+3           ; Load y
std Y+5,r24           ; Store y (2) into Blue[1]
ldi r24,lo8(7)
std Y+6,r24           ; Store 7

; mj.SetPxClr(x, y, Blue);
movw r24,r28          ; Load Y so we can pass mj to the function
adiw r24,1            ; 1 is where the address of mj is stored
ldd r22,Y+2           ; Store X (Y+2) in r23:r22
ldd r20,Y+3           ; Store Y (Y+3) in r21:r20
movw r18,r28          ; Get Y (top of stack) so we can get Blue[], r19:r18 needs to
                      ; be the address of the array of color values.
subi r18,lo8(-4)      ; Add 4.  This moves the address to the first item in Blue[]
sbci r19,hi8(-4)
call _ZN7MeggyJr8SetPxClrEhhPh
```

Addition

Addition is one of the simplest operations. It takes only one instruction. This example displays a bluish pixel on the MeggyJr by using addition.

(See mj_add.cpp and mj_add.s)

C Code:

```
MeggyJr mj;
byte x = 4;
byte y = 5;
byte color[3];
color[0] = x;
color[1] = y;
color[2] = x + y;
mj.SetPxClr(x, y, color);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 4;
ldi r24,lo8(4)
std Y+2,r24

; byte y = 5;
ldi r24,lo8(5)
std Y+3,r24

; byte color[3]; color[0] = x;
ldd r24,Y+2
std Y+4,r24

; color[1] = y;
ldd r24,Y+3
std Y+5,r24

; color[2] = x + y;
ldd r24,Y+2 ; Load x
ldd r25,Y+3 ; Load y
add r24,r25 ; Add them together (result is stored in the first register - r24)
std Y+6,r24 ; Store the answer in color[2] (Y+6)

; mj.SetPxClr(x, y, color);
movw r24,r28
adiw r24,1
ldd r22,Y+2
ldd r20,Y+3
movw r18,r28
subi r18,lo8(-4)
sbci r19,hi8(-4)
call _ZN7MeggyJr8SetPxClrEhhPh
```

Subtraction

Subtraction is one instruction like addition, but which instruction you use depends on whether you are using 16 bits, 8 bits, and signed or unsigned numbers. This example shows three different ways to subtract numbers by displaying a red pixel:
(See mj_subtract.cpp and mj_subtract.s)

C Code:

```
MeggyJr mj;
byte x = 7;
byte y = 2;
byte color[3];
color[0] = 16 - 5;
color[1] = 10 - 10;
color[2] = 7 - 7;
mj.SetPxClr(x, y, color);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 7;
ldi r24,lo8(7)
std Y+2,r24

; byte y = 2;
ldi r24,lo8(2)
std Y+3,r24

; byte color[3]; color[0] = 16 - 5;
ldi r24,lo8(16) ; Load 16
ldi r25,lo8(-5) ; Load -5
add r24,r25     ; Add them together (16 + -5 = 11).  The result is stored back
                ; into the first register, which is r24 in this case
std Y+4,r24     ; Store the answer in color[0]

; color[1] = 10 - 10;
ldi r24,lo8(10) ; Load 10 into a register
ldi r25,lo8(10) ; Load the other 10 into a register
sub r24,r25     ; Subtract r25 from r24 (10 – 10).  The result is stored back into
                ; the first register, r24
std Y+5,r24     ; Store the answer (0) in color[1]
```

```
; color[2] = 0;
ldi r24,lo8(7)  ; Load 7 into a register
subi r24,lo8(7) ; Subtract the immediate value 7
std Y+6,r24     ; Store the answer (0) into color[2]

; mj.SetPxClr(x, y, color);
movw r24,r28
adiw r24,1
ldd r22,Y+2
ldd r20,Y+3
movw r18,r28
subi r18,lo8(-4)
sbci r18,hi8(-4)
call _ZN7MeggyJr8SetPxClrEhhPh
```

You can use `add` to add a negative number (only if the number is signed). But there are also the `sub` and `subi` instructions. `sub` subtracts one register from another, and stores the answer into the first register listed. `subi` subtracts an immediate value from a register and puts the value back into the register.

Multiplication

The compiler likes to use optimizations when you give it immediate values. One of the things it will do to multiply is bit-shift, since that is faster at times. This example sets a pixel to green and uses multiplication to set the x,y position.
(See mj_mult.cpp and mj_mult.s)

C Code:

```
MeggyJr mj;
byte x = 3;
byte y = 2 * x;   // Normally the compiler would use a bit-shift
byte color[3] = {0, 10, 0};
mj.SetPxClr(x, y, color);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 3;
ldi r24,lo8(3)
std Y+2,r24

; byte y = 2 * x;
ldi r24,lo8(2) ; Load 2
```

```
ldd r25,Y+2    ; Load x (3)
muls r24,r25   ; Do the multiplication
std Y+3,r0     ; muls stores the answer in r1:r0, so we need to grab it from there

; byte color[3] = {0, 10, 0};
ldi r24,lo8(0)
std Y+4,r24
ldi r24,lo8(10)
std Y+5,r24
ldi r24,lo8(0)
std Y+6,r24

; mj.SetPxClr(x, y, color);
movw r24,r28
adiw r24,1
ldd r22,Y+2
ldd r20,Y+3
movw r18,r28
subi r18,lo8(-4)
sbci r19,hi8(-4)
call _ZN7MeggyJr8SetPxClrEhhPh
```

The `muls` instruction multiplies two registers together. It treats both numbers as signed integers. When it is done, it stores the result in `r1:r0`. Since we are only using 8 bits, we can grab the lower bits, which are in `r0`.

Division

Division is different than the other mathematical operations. There is no "divide" instruction in the AVR instruction set. However, the MeggyJr does have a divide function in its C library. We can divide numbers by calling that function. The following example shows a while pixel, with its coordinates set by using division.
(See mj_div.cpp and mj_div.s)

C Code:

```
MeggyJr mj;
byte x = 6;
byte y = x / 2;   // Normally the compiler would use a bit-shift
byte color[3] = {15, 15, 15};
mj.SetPxClr(x, y, color);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev
```

```
; byte x = 6;
ldi r24,lo8(6)
std Y+2,r24

; byte y = x / 2;
ldd r24,Y+2      ; Load x into the low bits (r25:r24 is dividend)
clr r25          ; Clear the high bits of r25:r24
ldi r22,lo8(2)   ; Load 2 into the low bits (r23:r22 is divisor)
clr r23          ; Clear the high bits of r23:r22
call __divmodhi4 ; Call the function that does the division
std Y+3,r22      ; Store just the low bits in y

; byte color[3] = {15, 15, 15};
ldi r24,lo8(15)
std Y+4,r24
ldi r24,lo8(15)
std Y+5,r24
ldi r24,lo8(15)
std Y+6,r24

; mj.SetPxClr(x, y, color);
movw r24,r28
adiw r24,1
ldd r22,Y+2
ldd r20,Y+3
movw r18,r28
subi r18,lo8(-4)
sbci r19,hi8(-4)
call _ZN7MeggyJr8SetPxClrEhhPh
```

The name of the function to use for division is `__divmodhi4`. It takes whatever is in `r25:r24` and divides it by `r23:r22`. The result is stored back into `r23:r22`. Since we are dividing 8-bit numbers, and the function uses 16-bit numbers, we need to make sure the high registers are cleared (set to zero). We can do that by using `clr`. If we don't clear the high registers, we are likely to get the wrong answer if there was lingering data in those registers.

And

And is a simple instruction like add. This example sets the auxiliary LEDs using an and operation. *(See mj_and.cpp and mj_and.s)*

C Code:

```
byte leds = 0xFF; // This would turn all of the LEDs on
leds &= 0xAA;     // This turns on only half them (10101010)
MeggyJr::AuxLEDs = leds;
// Remember to play a tone to force the LEDs to update
```

Assembly:

```
; byte leds = 0xFF;
ldi r24,lo8(-1)
std Y+2,r24

; leds &= 0xAA;
ldd r24,Y+2       ; Load leds into a register
andi r24,lo8(-86) ; And the register with the immediate value -86 (0xAA)
std Y+2,r24       ; Store the result back into leds

; MeggyJr::AuxLEDs = leds;
ldd r24,Y+2
sts _ZN7MeggyJr7AuxLEDsE,r24
```

To and a register with an immediate value, use the `andi` instruction. It does the operation and stores it back into the initial register (`r24` in this example). You can also use the `and` instruction, which does the operation on two registers. It then stores the result back into the first register.

Or

The or operation has the exact same structure as the and operation. This next example lights up the first and last auxiliary LEDs using or.
(See mj_or.cpp and mj_or.s)

C Code:

```
byte leds = 1;    // Left-most LED
leds |= 128;      // Turn on right-most LED too (10000001)
MeggyJr::AuxLEDs = leds;
// Remember to play a tone to force the LEDs to update
```

Assembly:

```
; byte leds = 0xFF;
ldi r24,lo8(-1)
std Y+2,r24

; leds |= 128
ldd r24,Y+2       ; Load leds
ori r24,lo8(-128) ; Or the register with the immediate value -128
std Y+2,r24       ; Store the answer back into leds

; MeggyJr::AuxLEDs = leds;
ldd r24,Y+2
sts _ZN7MeggyJr7AuxLEDsE,r24
```

The example uses `ori`, but you can also use `or` to do the operation on two registers. `or` will store the answer into the first register listed.

If-Statements

If-statements are written slightly differently in assembly and there are multiple steps to them. The first step is to load the load the variables into registers. Next, the conditional operation is performed on the registers. Lastly, the block is *skipped* if the result is *false*. That is important to note. When the condition is true, it continues execution into the if-block. When it is false, the block is jumped over (skipped). Because of this, you want to flip whatever you are testing for. For example, if you are testing for equality, you want to jump if the result is not equal.

Pseudo Code:

```
Load x into register
Is x greater than 10? (conditional)
If no, jump over block (branch)
      In block, do stuff (code)
Continue execution
```

This snippet displays a yellow pixel is x is non-zero:
(See mj_if.cpp and mj_if.s)

C Code:

```
MeggyJr mj;
byte color[3] = {3, 3, 0};
byte x = 5;
if(x)
   mj.SetPxClr(2, 2, color);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte color[3] = {3, 3, 0};
ldi r24,lo8(3)
std Y+2,r24
ldi r24,lo8(3)
std Y+3,r24
ldi r24,lo8(0)
std Y+4,r24
```

```
        ; byte x = 5;
        ldi r24,lo8(5)
        std Y+5,r24

        ; if(x) {
        ldd r24,Y+5 ; Load x into a register
        tst r24     ; Check if it is zero
        breq .L2    ; Jump to .L2 (over the if-block) if it equals zero

        ; mj.SetPxClr(2, 2, color);
        movw r24,r28
        adiw r24,1
        movw r18,r28
        subi r18,lo8(-2)
        sbci r19,hi8(-2)
        ldi r22,lo8(2)
        ldi r20,lo8(2)
        call _ZN7MeggyJr8SetPxClrEhhPh

        ; }
        .L2         ; Execution continues here after the if-block
```

The `tst` instruction checks if a register is zero. If it is zero, it sets the zero flag to true. `breq` then checks that flag, and if it true (the value is zero) it will jump to a label.

If-else

If-else statements use a similar "jump over the block" method as if-statements. But instead of jumping over the block when the condition is false, it jumps into the else-block. And at the end of if-block, it jumps over the else block.

Pseudo Code:

```
      Load x into register
      Is x greater than 10? (conditional)
      If no, jump into else-block (branch)
if-block:
            In if-block, do stuff (code)
            Jump over else block (jump)
else-block:
            In else-block, do other stuff (code)
      Continue execution
```

This next example displays a gold pixel if x is true (non-zero), or a green pixel if x is false (zero). *(See mj_else.cpp and mj_else.s)*

C Code:

```
MeggyJr mj;
byte x = 0;
byte gold[3] = {3, 3, 0};
byte green[3] = {0, 7, 0};
if(x)
   mj.SetPxClr(5, 5, gold);
else
   mj.SetPxClr(5, 5, green);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 0
ldi r24,0
std Y+2,r24

; byte gold[3] = {6, 6, 0};
ldi r24,lo8(6)
std Y+3,r24
ldi r24,lo8(6)
std Y+4,r24
ldi r24,lo8(0)
std Y+5,r24

; byte green[3] = {0, 7, 0};
ldi r24,lo8(0)
std Y+6,r24
ldi r24,lo8(7)
std Y+7,r24
ldi r24,lo8(0)
std Y+8,r24

; if(x) {
ldd r24,Y+2 ; Load x into a register
tst r24     ; Test if that register is zero
breq .L2    ; Jump into else-block if it is zero

; mj.SetPxClr(5, 5, gold);
movw r24,r28
adiw r24,2
movw r18,r28
subi r18,lo8(-3)
sbci r19,hi8(-3)
```

```
ldi r22,lo8(5)
ldi r20,lo8(5)
call _ZN7MeggyJr8SetPxClrEhhPh
rjmp .L4    ; We only want to run this block, so jump over the else-block

; } else {
.L2:        ; Condition was false, start execution here

; mj.SetPxClr(5, 5, green);
movw r24,r28
adiw r24,2
movw r18,r28
subi r18,lo8(-6)
sbci r19,hi8(-6)
ldi r22,lo8(5)
ldi r20,lo8(5)
call _ZN7MeggyJr8SetPxClrEhhPh

; }
.L4:        ; Execution continues
```

Testing Equality

The AVR instruction set makes it very easy to check various conditions. There are two general-purpose instructions that AVR uses to compare values. The first is `cp`. It compares the values in two registers and sets the condition flags. The second is `cpi`. It compares a register to an immediate value and also sets the condition flags. The condition flags are used to track the result of a comparison. That way, the next branch instruction can look at those flags and jump if it needs to. Here's some examples of `cp` and `cpi`.

C Code:

```
byte x = 5;
byte y = 0;
if(5 == x)
   y = 1;
if(y < x)
   ++y;
```

Assembly:

```
; byte x = 5;
ldi r24,lo8(5)
std Y+1,r24

; byte y = 0;
ldi r24,lo8(0)
std Y+2,r24
```

```
; if(5 == x) {
ldd r24,Y+1    ; Load x
cpi r24,lo8(5) ; Compare x to 5
brne .L2       ; Jump over the block if x != 5

; y = 1;
ldi r24,lo8(1)
std Y+2,r24

; }
.L2:

; if(y < x) {
ldd r24,Y+1    ; Load x
ldd r25,Y+2    ; Load y
cp r24,r25     ; Compare x and y
brge .L3       ; Jump over the block if y >= x

; ++y
ldd r24,Y+2
inc r24
std Y+2,r24

; }
.L3:
```

This next example tests if x equals 75 and displays a gold pixel if it is.
(See mj_equals.cpp and mj_equals.s)

C Code:

```
byte x = 5;
byte y = 0;
if(5 == x)
   y = 1;
if(y < x)
   ++y;
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 75;
ldi r24,lo8(75)
std Y+2,r24
```

```
; byte gold[3] = {6, 6, 0};
ldi r24,lo8(6)
std Y+3,r24
ldi r24,lo8(6)
std Y+4,r24
ldi r24,lo8(0)
std Y+5,r24

; if(75 == x) {
ldd r24,Y+2     ; Load x
cpi r24,lo8(75) ; Compare x to the immediate value 75
brne .L2        ; Jump over the block if x != 5

; mj.SetPxClr(5, 5, gold);
movw r24,r28
adiw r24,1
movw r18,r28
subi r18,lo8(-3)
sbci r19,hi8(-3)
ldi r22,lo8(5)
ldi r20,lo8(5)
call _ZN7MeggyJr8SetPxClrEhhPh

; }
.L2:            ; Continue execution
```

You can also use the `breq` command if you want to jump when 75 != x.

Less than

There are two branch instructions in AVR that check for less-than conditions. The first is `brlt`, which will jump if the condition is less-than. The other is `brge`, which is just the opposite – it jumps if the condition is greater-than or equal. There are also other instructions for negative and positive numbers. The next example will play a tone if x is less than y.
(See mj_lt.cpp and mj_lt.s)

C Code:

```
MeggyJr mj;
byte x = 20;
byte y = 50;
if(x < y)
   mj.StartTone(2500, 750);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
```

```
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 20;
ldi r24,lo8(20)
std Y+1,r24

; byte y = 50;
ldi r24,lo8(50)
std Y+2,r24

; if(x < y) {
ldd r25,Y+1 ; Load x
ldd r24,Y+2 ; Load y
cp r25,r24  ; Compare x and y
brge .L2    ; Jump over the block if x >= y

; mj.StartTone(2500, 750);
movw r24,r28
adiw r24,3
ldi r22,lo8(2500)
ldi r23,hi8(2500)
ldi r20,lo8(750)
ldi r21,hi8(750)
call _ZN7MeggyJr9StartToneEjj

; }
.L2:        ; Continue execution
```

Not

The not operation flips the truth value of a variable. If instead you wanted to flip the bits on a value, use `com`. That takes the one's complement of a register. When not-ing an if-statement, you can flip the branch statement you're using. For example, `breq` would become `brne`, and `brge` would become `brlt`. In this example however, we are not-ing a variable. When it is true, it will be set to zero (false), and when it is false, it is set to non-zero (true). This example plays a tone if !x is false.

(See mj_not.cpp and mj_not.s)

C Code:

```
MeggyJr mj;
byte x = 50;
x = !x;
if(0 == x)
   mj.StartTone(2500, 750);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 50;
ldi r24,lo8(50)
std Y+2,r24

; x = !x;
ldd r24,Y+2    ; Load x
tst r24        ; Test if it is zero (false)
breq .L2       ; Jump to .L2 if it is zero
ldi r24,lo8(0) ; x is non-zero (true), load a false value (zero)
std Y+2,r24    ; Store the false value in it
rjmp .L3       ; Skip over the else part
.L2:           ; x is zero (false)
ldi r24,lo8(1) ; Load 1 (a true value)
std Y+2,r24    ; Store the true value in it

; if(0 == x) {
.L3:
ldd r24,Y+1
tst r24
brne .L4

; mj.StartTone(2500, 750);
movw r24,r28
adiw r24,2
ldi r22,lo8(2500)
ldi r23,hi8(2500)
ldi r20,lo8(750)
ldi r21,hi8(750)
call _ZN7MeggyJr9StartToneEjj

; }
.L4:
```

Short-Circuited And

Sometimes we need to check if multiple things are true before going into an if-block. To do that, we can use short-circuited ands. There are no special instructions in assembly to do a short-circuit. Instead, you stack multiple if-statements together. The catch is this: as soon as any if-statement is false, the program should jump over the if-block.

Pseudo Code:

```
Load x into register
Is x greater than 10? (conditional)
If no, jump over the if-block (branch)
Load y into register
Is y true? (conditional)
If no, jump over the if-block (branch)
      In if-block, do stuff (code)
Continue execution
```

The next example displays a pink pixel if x and y are true, or a purple one if either is false. *(See mj_sc_and.cpp and mj_sc_and.s)*

C Code:

```
MeggyJr mj;
byte x = 5;
byte y = 7;
byte color[3];
if(x && y)  {
   color[0] = 7;  // Pink
   color[1] = 1;
   color[2] = 1;
}   else   {
   color[0] = 1;  // Purple
   color[1] = 1;
   color[2] = 7;
}
mj.SetPxClr(2, 3, color);
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte x = 5;
ldi r24,lo8(5)
std Y+2,r24

; byte y = 7;
ldi r24,lo8(7)
std Y+3,r24
```

```
; if(x && y) {
ldd r24,Y+2 ; Load x
tst r24     ; First check if x is true
breq .L2    ; Jump into the else-block if it is false
ldd r24,Y+3 ; Load y
tst r24     ; Check if y is true
breq .L2    ; Jump into the else-block if it is false

; color[0] = 7; color[1] = 1; color[2] = 1;
ldi r24,lo8(7)
std Y+4,r24
ldi r24,lo8(1)
std Y+5,r24
ldi r24,lo8(1)
std Y+6,r24
rjmp .L3

; } else {
.L2:

; color[0] = 1; color[1] = 1; color[2] = 7;
ldi r24,lo8(1)
std Y+4,r24
ldi r24,lo8(1)
std Y+5,r24
ldi r24,lo8(7)
std Y+6,r24

; }
,L3:

; mj.SetPxClr(2, 3, color);
movw r24,r28
adiw r24,1
movw r18,r28
subi r18,lo8(-4)
sbci r19,hi8(-4)
ldi r22,lo8(2)
ldi r20,lo8(3)
call _ZN7MeggyJr8SetPxClrEhhPh
```

Short-Circuited Or

Short-circuited ors are the opposite of their counterpart, short-circuited and. Instead of jumping over the if-block if any of stacked conditions are false, it jumps into the if-block as soon as one of them is true. This example plays a high pitched tone if x or y is true. Alternatively, it will play a very low tone if both x and y are false.

(See mj_sc_or.cpp and mj_sc_or.s)

C Code:

```
MeggyJr mj;
byte x = 0;
byte y = 7;
if(x || y)   {
   mj.StartTone(2500, 750);
}   else   {
   mj.StartTone(50000, 750);
}
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; char x = 0;
ldi r24,lo8(0)
std Y+2,0

; char y = 7;
ldi r24,lo8(7)
std Y+3,r24

; if(x || y) {
ldd r24,Y+2 ; Load x
tst r24     ; Check if it is true
brne .L2    ; Jump into the block if it is true
ldd r24,Y+3 ; Load y
tst r24     ; Check if it is true
breq .L3    ; Jump over the block if it is false

; {
.L2:

; mj.StartTone(2500, 750);
movw r24,r28
adiw r24,3
ldi r22,lo8(2500)
ldi r23,hi8(2500)
ldi r20,lo8(750)
ldi r21,hi8(750)
call _ZN7MeggyJr9StartToneEjj
rjmp .L4

; } else {
.L3:
```

```
        ; mj.StartTone(50000, 750);
        movw r24,r28
        adiw r24,3
        ldi r22,lo8(-15536)
        ldi r23,hi8(-15536)
        ldi r20,lo8(750)
        ldi r21,hi8(750)
        call _ZN7MeggyJr9StartToneEjj

        ; }
        .L4:
```

Note that the last condition is inverted. A quick note about not-ing a short-circuit: No results are cached on the stack. `!(a && b)` becomes `(a || b)`. `!(a || b)` becomes `(a && b)`. These transformations are applied by the compiler. *mj_not2.cpp* and *mj_not2.s* illustrate this.

While Loops

While loops a basically if-statements that jump to the condition after they are finished executing. Like if-statements, while loops have different parts. The first part is the condition; it is an if-statement. If the condition is false, it jumps to after the while loop. When it is true, the execution continues into the block. The next part is the execution of the code in the block, the instructions that should be repeated. The last part, is a jump. It jumps back to the if-statement at the beginning of the whole while loop. Make sure that the if-statement at the beginning jumps over this final jump label, otherwise you'll have an infinite loop.

Pseudo Code:

```
if:
      Load x into register
      Is x less than 10? (conditional)
      If no, jump over the while (branch)
loop:
            In while loop, do stuff (code)
            Jump back to if-statement (jump)
      Continue execution
```

This example uses a loop to light up 8 pixels diagonally, each with increasing brightness.
(See mj_while.cpp and mj_while.s)

C Code:

```
      MeggyJr mj;
      byte i = 8;
      byte color[3];
```

```
while(1)    {
   color[0] =
   color[1] =
   color[2] = i;
   --i;
   mj.SetPxClr(i, i, color);
}
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; byte i = 8;
ldi r24,lo8(8)
std Y+2,r24

; while(i) {
.L2:
ldd r24,Y+2 ; Load i
tst r25     ; Check if it is zero
breq .L3    ; If it is zero, jump over the loop

; color[0] = color[1] = color[2] = i;
ldd r24,Y+2
std Y+5,r24
std Y+4,r24
std Y+3,r24

; --i;
ldd r24,Y+2
dec r24
std Y+2,r24

; mj.SetPxClr(i, i, color);
movw r24,r28
adiw r24,1
movw r18,r28
subi r18,lo8(-3)
sbci r19,hi8(-3)
ldd r22,Y+2
ldd r20,Y+2
call _ZN7MeggyJr8SetPxClrEhhPh

; }
jmp .L2     ; Re-run the loop
.L3:        ; Continue execution
```

This is the while loop syntax. A do-while loop is slightly different. Instead of putting the if-statement at the top, it goes at the bottom, right before the final jump. And none of the labels need to be moved.

Pointers

Pointers are very easy to use in the AVR instruction set. The registers `r29:r28` contain the top of the stack, which are what Y references. To get a pointer to a variable, get the top of the stack, from `r29:r28`, add the byte-offset, and store that value back onto the stack. To dereference a value by using a pointer, you'll need the help of the Z variable. Z references `r31:r30`. Load the pointer's value (the address) into `r31:r30`, then `ld` into another register via Z. This example sets the auxiliary LEDs through pointers and will hopefully clear things up.
(See mj_pointer.cpp and mj_pointer.s)

C Code:

```
byte leds = 195;
byte *p = &leds;
MeggyJr::AuxLEDs = *p;
```

Assembly:

```
; byte leds = 195;
ldi r24,lo8(-61)
std Y+2,r24

; byte *p = &leds;
movw r24,r28 ; Load the top of the stack
adiw r24,2   ; Move to where leds is store (+2)
std Y+4,r25  ; Store the address on the stack (high bits)
std Y+3,r24  ; and the low bits

; MeggyJr::AuxLEDs = *p;
ldd r24,Y+3  ; Load the low bits of the address
ldd r25,Y+4  ; Load the high bits of the address
movw r30,r24 ; Copy the address from r25:r24 to r31:r30
ld r24,Z     ; Load indirectly (loads the value pointed to by the address in Z)
sts _ZN7MeggyJr7AuxLEDsE,r24
```

Pausing for a Bit with _delay_ms()

Waiting is an important part of displaying objects on screen. If you don't pause for a bit between frame updates, the changes will be so fast that they can't be seen. That's where `_delay_ms()` comes in. `_delay_ms()` is a function that pauses (blocks) for the specified number of milliseconds you give it. To use it, you must include `<util/delay.h>`. The following is an example of a blinking light.
(See mj_delay.cpp and mj_delay.s)

C Code:

```
#include “MeggyJr.h”
#include <util/delay.h>
// …
MeggyJr mj;
byte color[3];
while(1)   {
   color[0] =
   color[1] =
   color[2] = 5;
   mj.SetPxClr(3, 3, color);
   _delay_ms(100);
   color[0] =
   color[1] =
   color[2] = 0;
   mj.SetPxClr(3, 3, color);
   _delay_ms(100);
}
```

Assembly:

```
; MeggyJr mj;
movw r24,r28
adiw r24,1
call _ZN7MeggyJrC1Ev

; while(1) {
.L2

; color[0] = color[1] = color[2] = 5;
ldi r24,lo8(5)
std Y+2,r24
std Y+3,r24
std Y+4,r24

; mj.SetPxClr(3, 3, color);
movw r24,r28
adiw r24,1
movw r18,r28
subi r18,lo8(-2)
sbci r19,hi8(-2)
ldi r22,3
ldi r20,3
call _ZN7MeggyJr8SetPxClrEhhPh
```

```
; _delay_ms(100);
ldi r24,lo8(1000) ; Time to delay (in ms x10)
ldi r25,hi8(1000) ; Time to delay (in ms x10)
.L3:
movw r30,r16
1: sbiw r30,1
brne 1b
sbiw r24,1
brne .L3

; color[0] = color[1] = color[2] = 0;
ldi r24,lo8(0)
std Y+2,r24
std Y+3,r24
std Y+4,r24

; mj.SetPxClr(3, 3, color);
movw r24,r28
adiw r24,1
movw r18,r28
subi r18,lo8(-2)
sbci r19,hi8(-2)
ldi r22,3
ldi r20,3
call _ZN7MeggyJr8SetPxClrEhhPh

; _delay_ms(100);
ldi r24,lo8(1000) ; Time to delay (in ms x10)
ldi r25,hi8(1000) ; Time to delay (in ms x10)
.L4:
movw r30,r16
1: sbiw r30,1
brne 1b
sbiw r24,1
brne .L4

; }
jmp .L2
```

The _delay_ms() function is inlined. Because of this, you shouldn't try to mess with the inner-workings of it. It basically runs in a loop until it has counted down all the way from the delay time. Note that; the number it uses is ten times the original value given.

Useful Tables

Common Tones (C_3 – B_5)

Tone Name	Frequency (Hz)	MeggyJr Value
C_3	130.81	61157
$C^{\#}_3$	138.59	57724
D_3	146.83	54484
$D^{\#}_3$	155.56	51427
E_3	164.81	48540
F_3	174.61	45816
$F^{\#}_3$	185.00	43243
G_3	196.00	40816
$G^{\#}_3$	207.65	38526
A_3	220.00	36363
$A^{\#}_3$	233.08	34322
B_3	246.94	32396
C_4	261.63	30577
$C^{\#}_4$	277.18	28862
D_4	293.66	27242
$D^{\#}_4$	311.13	25712
E_4	329.63	24269
F_4	349.23	22907
$F^{\#}_4$	369.99	21622
G_4	392.00	20408
$G^{\#}_4$	415.30	19263
A_4	440.00	18181
$A^{\#}_4$	466.16	17161
B_4	493.88	16198
C_5	523.25	15289
$C^{\#}_5$	554.37	14430
D_5	587.33	13620
$D^{\#}_5$	622.25	12856
E_5	659.26	12134
F_5	698.46	11453
$F^{\#}_5$	739.99	10810
G_5	783.99	10204
$G^{\#}_5$	830.61	9631
A_5	880.00	9090
$A^{\#}_5$	932.33	8580
B_5	987.77	8099

Source: Frequencies of Musical Notes

MeggyJr Button Codes

Button	Binary	Hex	Decimal
B	00000001	01	1
A	00000010	02	2
Up	00000100	04	4
Down	00001000	08	8
Left	00010000	10	16
Right	00100000	20	32