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`

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.

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.

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};
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(-4)     ; Add 4, this moves the address to the first item in red[]
sbci r19,hi8(-4)
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 r = 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 r = 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. 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 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

; 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           ; Subtract 0 (this resets the zero 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 r18,lo8(15290)   ; Store low bits of tone into r18
ldi r19,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+2,r24          ; Store 0
ldi r24,lo8(5)
std Y+3,r24          ; Store 5
ldi r24,lo8(0)
std Y+4,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(-4)      ; Add 4, this moves the address to the first item in Green[]
sbci r19,hi8(-4)
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. 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::AuxLEDs = 0xAA;
```

Assembly:

```
ldi r24,lo8(-86)             ; Load the value into a temp register
sts _ZN7MeggyJr7AuxLEDsE,r24 ; Store the value into the AuxLEDs variable
```

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

Subtraction

Multiplication

Division

And

Or

Not

If-Statements

If-else

Testing Equality

Less than

While Loops

Short-Circuited And

Short-Circuited Or

Pointers

Functions

Remember to add delay!

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