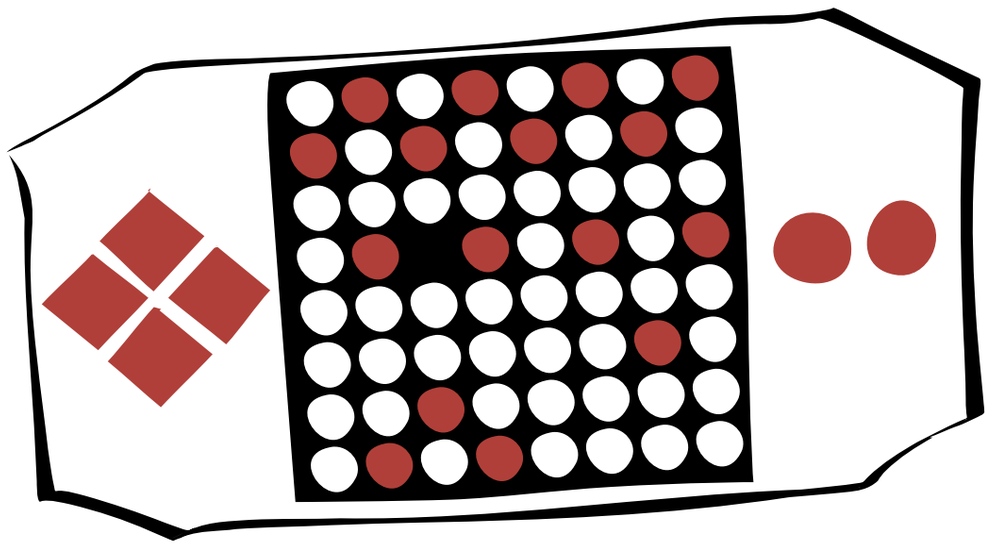
Meggy Jr RGB

- Programming Guide -



An open-source hardware+software project designed by

Evil Mad Scientist Laboratories

Making the World a Better Place, One Evil Mad Scientist at a Time

Project Home: http://www.evilmadscientist.com/go/MeggyJr

Technical Support: http://www.evilmadscientist.com/forum/

Mailing List: http://groups.google.com/group/meggydev

Evil Mad Science

Library version 1.31

Manual v. 1.34

This guide is an introduction to getting started with programming on the Meggy Jr RGB, an open-source LED matrix game development kit designed by Evil Mad Scientist Laboratories.

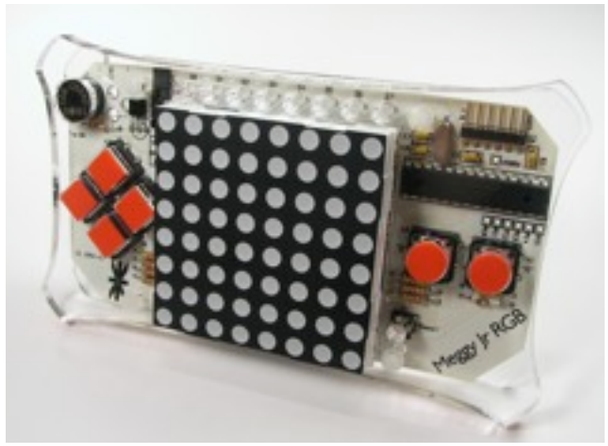
Just to be clear: Meggy Jr kits come preprogrammed. Reprogramming *is not required, but can be fun*. Even if you aren't (yet) interested in programming, you might want to follow along so that you can install applications that other users have contributed.

While this guide is not meant to be a comprehensive reference for programming in general, it should be a useful starting point as you dig in to programming Meggy Jr RGB. Our principal focus will be on programming Meggy Jr through the Arduino development environment (www.arduino.cc), using easy high-level commands to get Meggy to do cool stuff.

If you're already comfortable working with AVR microcontrollers in a different environment, you may want to start out in a different direction. It is possible to program Meggy Jr RGB through lower-level library functions and in different development other environments such as AVR-GCC, or even to start with the circuit diagram and build up your own code from scratch.

What is required for programming Meggy Jr RGB?

1. Meggy Jr RGB



(Well, obviously, right?)

Meggy Jr is available at the Evil Mad Science Shop, http://evilmadscience.com/

2. USB-TTL Cable

FTDI model TTL-232R or equivalent. A "smart" converter cable with a USB interface chip inside. One end hooks up to your USB port, the other to Meggy Jr. This allows you to program Meggy Jr directly through the Arduino development environment.

USB-TTL Cable

USBtinyISP

Alternately, Meggy Jr can be programmed through an AVR ISP programmer, such as the USBtinyISP. To use the Arduino environment with the USBtinyISP, follow these instructions: http://www.arduino.cc/en/Hacking/Programmer

3. Computer, Internet access, USB port....

All of the software that you'll need is available online for free. You'll need a reasonably recent vintage computer (Mac, Windows, or Linux) and internet access.

You can find the software links here: http://www.evilmadscientist.com/go/meggyjr

If your programmer is one of the two above, you'll need a USB port too.

1. Download Arduino 0018 or newer (from http://arduino.cc/en/Main/Software) and install.

2. Download and install the latest Meggy Jr RGB Arduino Library from
http://code.google.com/p/meggy-jr-rgb/downloads

To install, first unzip the library. You should end up with a folder called "MeggyJr."
Place this folder in the "libraries" subfolder inside your Arduino sketchbook folder.
If you aren't sure where your sketchbook folder is located, open Arduino and go to
File>Preferences. The sketchbook location is listed there; it's usually a folder named "Arduino."
Open that folder and– if there isn't one already –make a folder inside of it called *libraries*.
Place the MeggyJr folder inside your libraries folder.
Restart Arduino. If the library is in the right place, you'll see a set of example
programs listed in the menu under File>Examples>MeggyJr

3. Select board type from the Tools>Board> menu.
- If you have the ATmega168 chip, select "Duemilanove w/ ATmega168"
- If you have the ATmega328 chip, select "Duemilanove w/ ATmega328"

Note: This chip is the big one in the upper right hand corner of the board.

4. Load an example program from your menu. A good one to start with is:
File>Sketchbook>Examples>Library-MeggyJr>MeggyJr_MeggyBrite

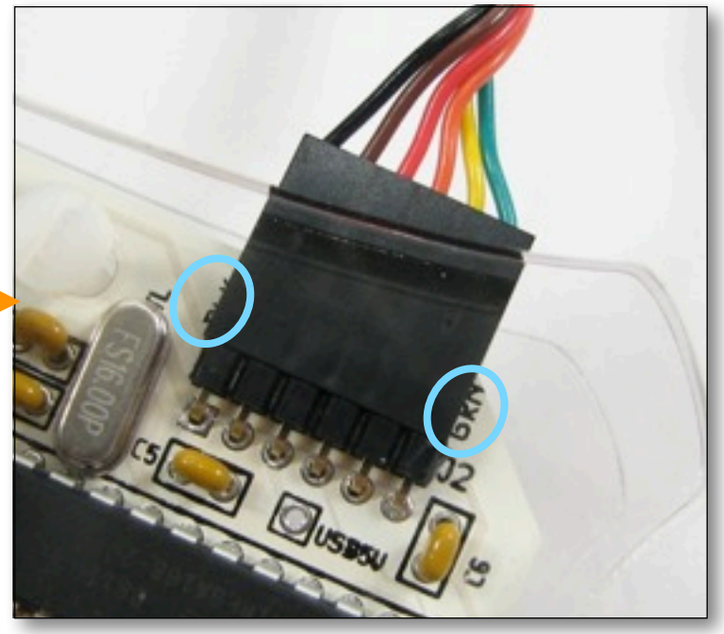
5. Verify (compile) the program by pressing the Verify button in the upper-left of the Arduino window; it has the "play" symbol, a right facing triangle.

6. Connect your USB-TTL cable to your computer and Meggy Jr.
The black and green ends of the Meggy side connector are labeled on the circuit board.

7. To program Meggy Jr, press the "Upload to I/O Board" button at the top of the Arduino program window (the other "right arrow").
It typically takes about 15 seconds. Note that the Meggy Jr needs to be powered on for programming.

Using Windows? Avoid tearing your hair out by setting:
Device Manager > Comm Ports > USB Serial Port >
Port Settings > Advanced button > Set RTS On Close.

"VERIFY"



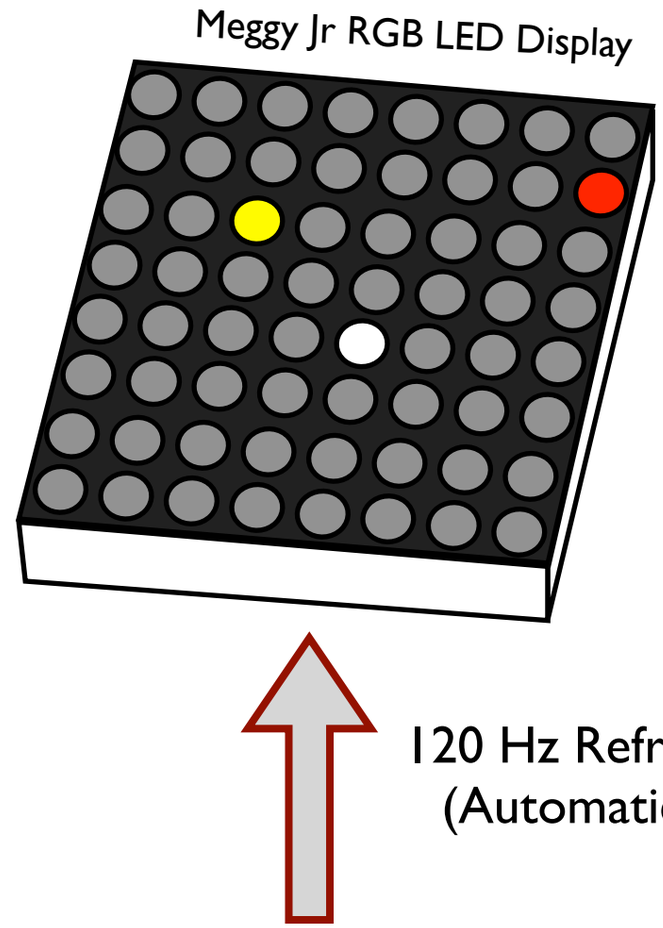
"UPLOAD TO I/O BOARD"

The Meggy Jr Library

The purpose of the Meggy Jr Library is to provide a software interface to the Meggy Jr RGB hardware. One of the things that it does is to allocate a chunk of the AVR microcontroller's RAM to act as *Display Memory*, analogous to the video memory in a desktop computer system. The Display Memory is a fairly large array of data that fully describes the state of all 200 LEDs on the Meggy Jr RGB. The LED display is constantly redrawn at a rate of 120 times per second, fully reading out the contents of that Display Memory and using it to control the LEDs.

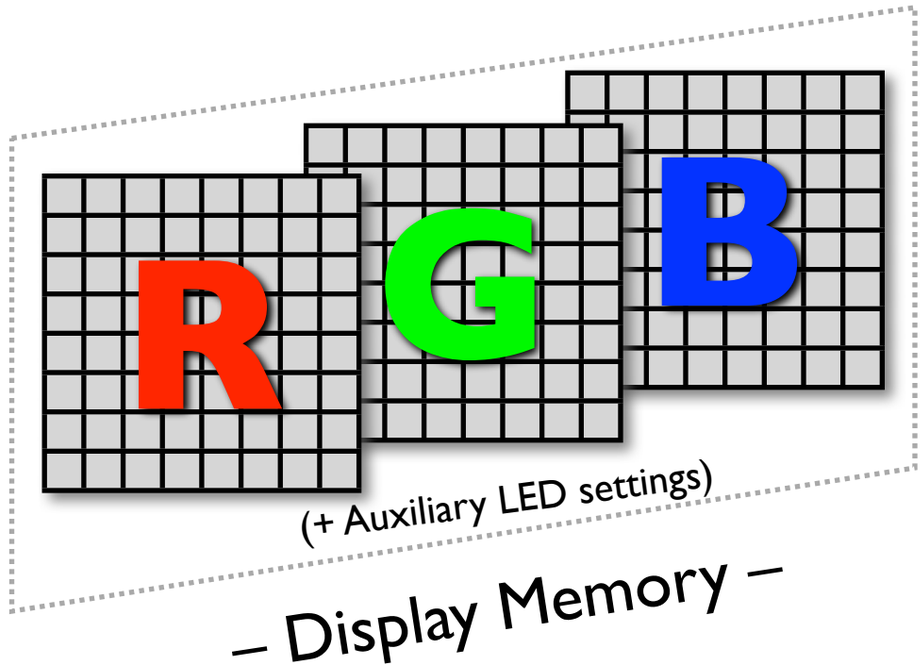
The Meggy Jr Library provides interface calls to directly set and read values in the Display Memory. While this can give you great control over the LEDs, it can also involve unwieldy data manipulation.

To ease this process, we can use a higher-level interface, the *Meggy Jr Simplified Library*, which allows the use of pre-defined colors, and greatly simplifies the process of filling the display memory. The Simplified Library is a much better place for us to start, so we will save the details of the Display Memory for later.



Meggy Jr Library
"Low level API":

User creates instance of Meggy Jr class and and writes R,G,B values directly to Display Memory



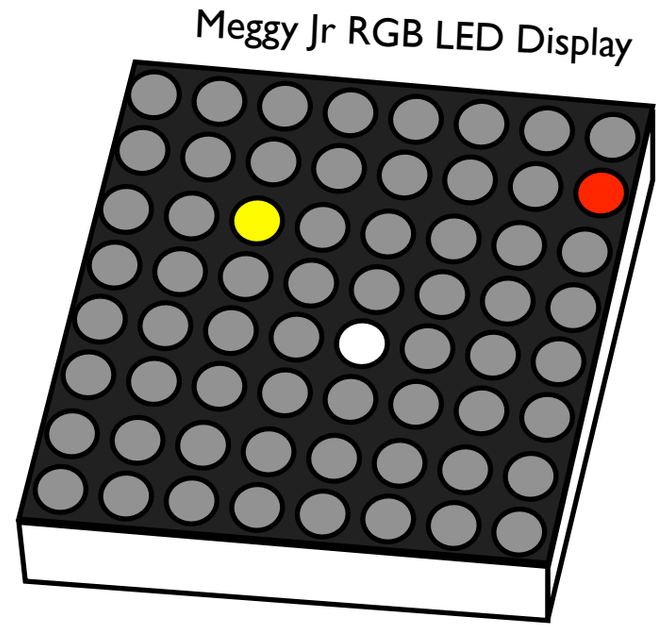
The Meggy Jr Simplified Library

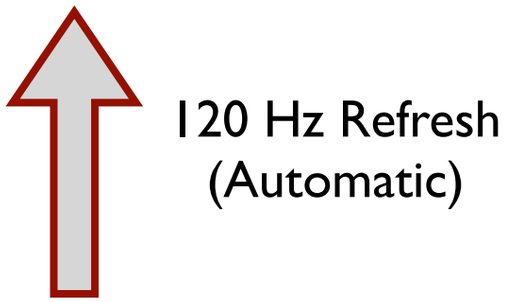
The Meggy Jr Simplified Library (MJSL) is a set of macros and functions *on top of* the main Meggy Jr Library that let you get started quickly without deep knowledge of the Display Memory or how the hardware works. It also removes the excess complexity that is normally associated with libraries in the Arduino environment, so you can *just use the darned thing*. And it works.

Under the hood, the Meggy Jr Simplified Library uses pre-defined calls to the (full) Meggy Jr Library, with its Display Memory and internal functions, to constantly refresh the screen at 120 Hz.

The Simplified Library also adds an additional memory array that makes drawing to the LED display more intuitive. Besides the Display Memory already mentioned, the MJSL uses a second off-screen drawing space called the *Game Slate*, where we actually perform the drawing.

In what follows, we'll walk through the process of using the MJSL.





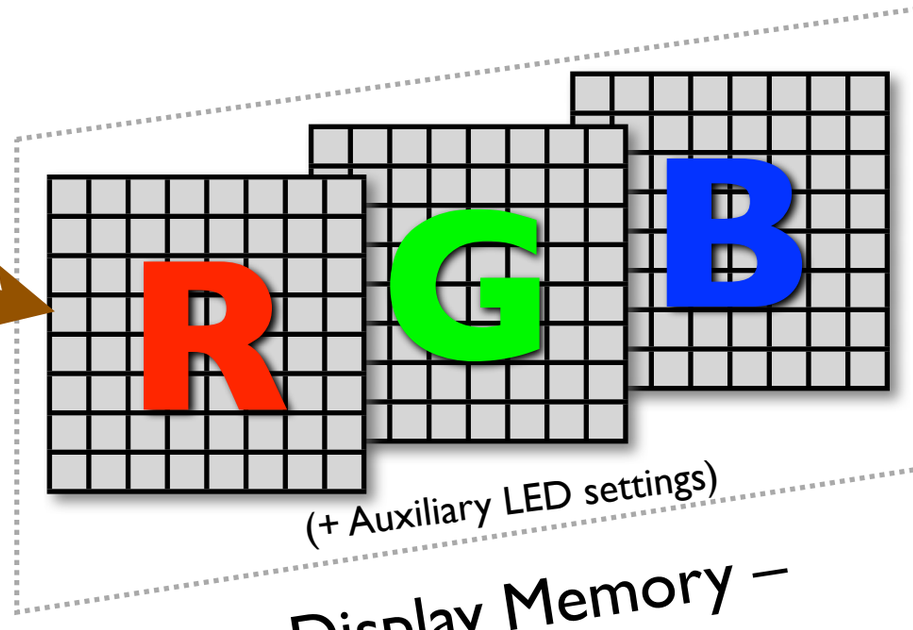
Meggy Jr Simplified Library ("High level API"):

User employs simple drawing commands to write to Game Slate

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	3	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	7	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Game Slate

(Game Slate data is used to fill the Display Memory)



Let's get started with a *very* brief intro to programming in the Arduino development environment.

An Arduino document-- a program written in the Arduino environment-- is called a *sketch*.

A basic sketch has two sections: *Setup* and *Loop*.

The *Setup* section contains programming statements that are executed *once*, when the program first runs.

The *Loop* section contains statements that are executed over and over again, so long as the program runs.

```
void setup()
{
  Statements that run once, when the sketch starts
}

void loop()
{
 Statements that run over and over again
}
```

Within these sections, we can enter programming statements: commands that address the Meggy Jr hardware as well as general-purpose programming with things like variables, loops, and conditional statements.

Please see http://arduino.cc/en/Reference/ for complete documentation about the Arduino programming environment and language.

While much of the Arduino language reference consists of useful and relevant syntax and functions, note that the "I/O" functions may interfere with the Meggy Jr library functions.

Three things you should know right way about the syntax:

1. Most statements end with a semicolon ';'
2. Single line comments start with two slashes: '//'
3. Multi-line comments go between '/*' and '*/'

Next, let's walk through a complete sketch for the Meggy Jr RGB that blinks a single LED pixel:

```
#include <MeggyJrSimple.h>

void setup()
{
  MeggyJrSimpleSetup();
}

void loop()
{
  DrawPx(3,4,Yellow);
  DisplaySlate();
  delay(1000);

  ClearSlate();
  DisplaySlate();

  delay(1000);
}
```

The *Setup* section. Runs *once*.

`#include <MeggyJrSimple.h>`

This "#include" line invokes the Meggy Jr Simplified Library (MJSL). It's the first of two required lines of code to set up and use the MJSL.

Note that library calls like this go at the head of the program, before setup() or loop().

`MeggyJrSimpleSetup();`

The second of two required lines of code to set up and use the MJSL. This line sets up the Meggy Jr Display Memory and performs various hardware initializations.

[To Be Continued...]

No need to re-type this example! It's one of the demo programs:
File>Sketchbook>Examples>Library-MeggyJr>MeggyJr_Blink

```
#include <MeggyJrSimple.h>


void setup()
{
  MeggyJrSimpleSetup();
}


void loop()
{
  DrawPx(3,4,Yellow);
  DisplaySlate();
  delay(1000);

  ClearSlate();
  DisplaySlate();

  delay(1000);
}
```

The *Loop* section. Repeats.

`Delay(1000);`

A standard Arduino function: Wait idly for a given number of milliseconds. Here it delays 1000 ms, or one second.

`DrawPx(3,4,Yellow);`

The DrawPx function is part of the MJSL. As used here, it will draw a pixel at x=3, y=4, in color Yellow. The origin (0,0) is at the lower-left corner of the LED display, and Yellow is a pre-defined color. DrawPx does not on its own write directly to the LED display, but instead writes to the Game Slate. Nothing will show up on the LED display until you call DisplaySlate.

(Yes, there's a list of defined colors-- we'll get there soon.)

`DisplaySlate();`

The DisplaySlate function is part of the MJSL. It copies the Game Slate into the Display Memory, where the contents will be shown on the LED display.
By calling it right here, it makes the LED display actually show that one yellow pixel.

`ClearSlate();`

Also part of the MJSL. This function empties the Game Slate. Again, no change will show up on the LED display until you call DisplaySlate.

So, after that we call DisplaySlate() again to actually show the screen full of dark pixels, and then wait another 1000 ms before starting the loop again.

Let's recap and look at what we do to put colored dots on the screen.

There are two steps:

1. Draw to the Game Slate with *DrawPx*:

```
DrawPx(2,5,Yellow);
DrawPx(4,3,White);
DrawPx(7,6,Red);
```

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	3	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	7	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(0,0)

Game Slate

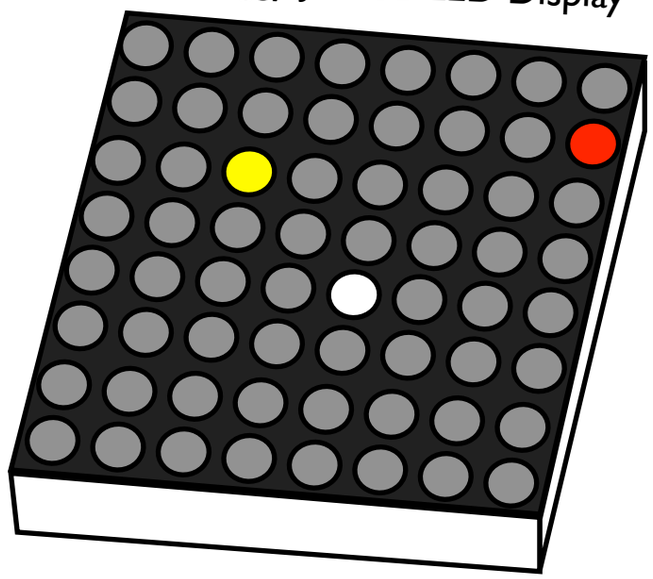
(Pre-defined colors are stored in the Game Slate *as numbers*.)

```
DisplaySlate();
```

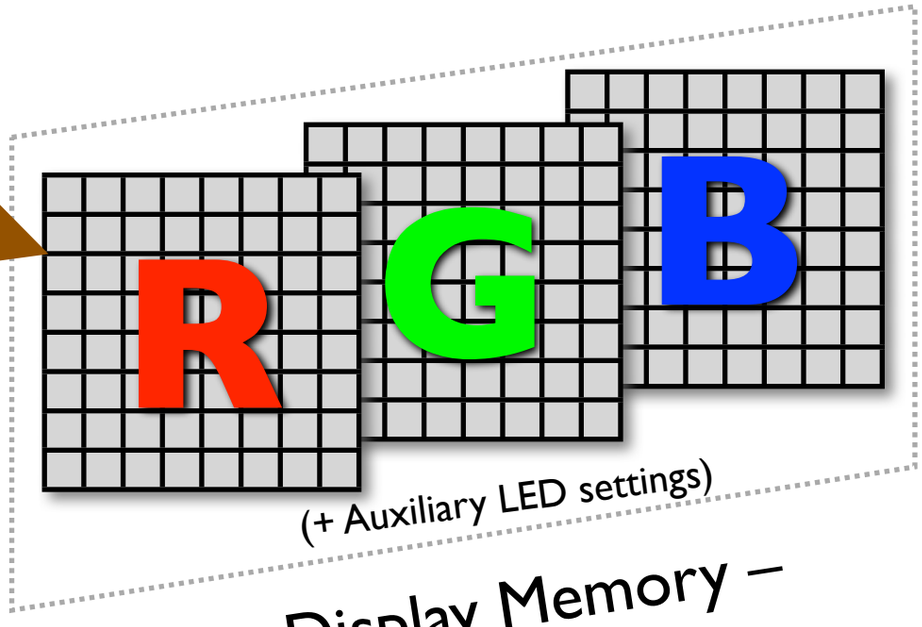
2. When done drawing, use *DisplaySlate* to write your drawing to the Display Memory

DisplaySlate uses a color look-up table that contains the RGB definition of the named colors.

Meggy Jr RGB LED Display



120 Hz Refresh (Automatic)



– Display Memory –

There are ten important functions in the Meggy Jr Simplified Library, which have to do with graphics, buttons, and sound. We've met a few of these already.

The graphics functions are as follows:

1. *ClearSlate* :: Erase the whole Game Slate
2. *DrawPx* :: Color in a pixel on the Game Slate.
3. *DisplaySlate* :: Copy the current contents of the Game Slate to the LED Display Memory.
4. *SetAuxLEDs* :: Write a value to the Meggy Jr Auxilary LEDs.
5. *ReadPx* :: Read the color of a pixel in the Game Slate.
6. *EditColor* :: Configure custom pixel colors.

These two functions check the buttons:

7. *CheckButtonsDown* :: Check to see which buttons are currently pressed down.
8. *CheckButtonsPress* :: Check to see which buttons are pressed that weren't, last we checked.

And one function is for making simple sounds:

9. *Tone_Start* :: Begin sound output at a given frequency

In the next section, we'll go over the usage of these functions, along with a few examples.

1. *ClearSlate* :: Erase the whole Game Slate

Example usage:

```
ClearSlate();
```

This routine clears the Game Slate, which is the off-screen drawing slate where you can take your time making a new graphic before copying it to the Display Memory, from where it is automagically drawn on the LEDs. It is equivalent to (and functions by) filling in each pixel in the Game Slate with the number zero or the color "Dark."

Because ClearSlate writes to the Game Slate, not to the Display Memory, it does not have any effect on which LEDs are actually displayed until you next run the DisplaySlate routine. To actually draw a blank screen, you might want to use the following two-line code example that first clears the game slate and then copies the blank slate to the Display memory:

```
ClearSlate();
DisplaySlate();
```

2. *DrawPx* :: Color in a pixel on the Game Slate.

Syntax: `DrawPx(x,y,Color);` *Where the origin x=0, y=0 is in the lower left corner of the LED display, and Color is the name of a color.*

Example usage: `DrawPx(3,4,Blue);` Draws a bright blue dot at position x=3, y = 4.

The x and y inputs are 8-bit unsigned integers (type 'byte'), and should each only be given numbers or variables that will be in the range 0 to 7. (Eight possible values for eight rows and eight columns.)

The list of pre-defined colors is given on the next page. Each color has an equivalent numerical value, and can be referred to (at your choice) either by name or number.

Example: `DrawPx(0,7,Red);` is equivalent to `DrawPx(0,7,1);`

The pixels that you write to the Game Slate with DrawPx will not actually be displayed on the LED display until you call the function DisplaySlate.

The **pre-defined color names** are as follows:

0	Dark
1	Red
2	Orange
3	Yellow
4	Green
5	Blue
6	Violet
7	White
8	DimRed
9	DimOrange
10	DimYellow
11	DimGreen
12	DimAqua
13	DimBlue
14	DimViolet
15	FullOn
16	CustomColor0
17	CustomColor1
18	CustomColor2
19	CustomColor3
20	CustomColor4
21	CustomColor5
22	CustomColor6
23	CustomColor7
24	CustomColor8
25	CustomColor9

"Dark" means all LEDs off, and "FullOn" means all LEDs full on. It's not a balanced white, but it is bright; makes a good blinking cursor or flash. The other colors in the range 1 to 14 are various mixes of the red, green, and blue elements.

The RGB color mix of each of the pre-defined colors can be changed by using the EditColor routine. Besides the standard colors, there are also ten user-configurable colors, CustomColor0 through CustomColor9. These are each equivalent to Dark until you configure them with EditColor.

The color names are *enumerated*, which means that you can use the numbers from the left-hand column interchangeably with the color names- and so colors can be referred to (at your choice) by number.

Example: `DrawPx(0,7,Red);` is equivalent to `DrawPx(0,7,1);`

(So... use whichever is convenient.)

3. *DisplaySlate* :: Copy the current contents of the Game Slate to the LED Display Memory.

Example usage:

```
DisplaySlate();
```

This routine takes the current contents of the Game Slate, and uses it to fill the Display Memory, from where it is automagically drawn on the LEDs. Running *DisplaySlate* does not affect the contents of the Game Slate.

In the process of filling the main Display Memory, *DisplaySlate* translates between the color names that are stored (as numbers) in the Game Slate and the RGB values that must go in the Display Memory. This is accomplished by looking up the RGB definition of each color in a color lookup table at the time that *DisplaySlate* is called.

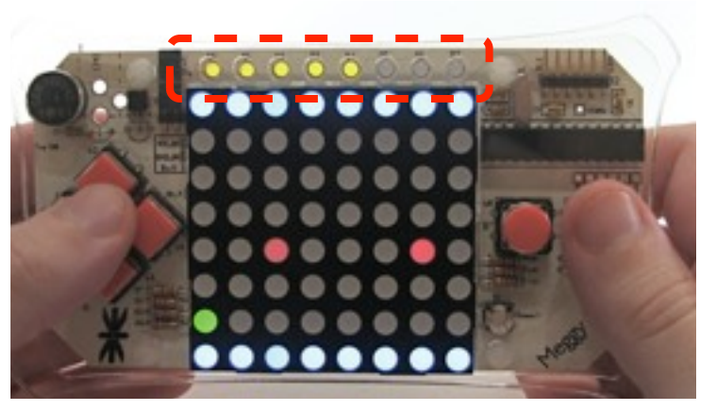
This means that if you have used the *EditColor* routine to change or define custom colors, the colors written to the LED display will be the definitions in effect at the moment that *DisplaySlate* is executed, not the definitions in place at the time that *DrawPx* or any other function was called. (See also examples on Page 17).

4. *SetAuxLEDs* :: Write a value to the Meggy Jr Auxilary LEDs.

Syntax: `SetAuxLEDs(value)`

'value' is an 8-bit unsigned integer (type 'byte') in the range 0 to 255.

Example usage: `SetAuxLEDs(31);`

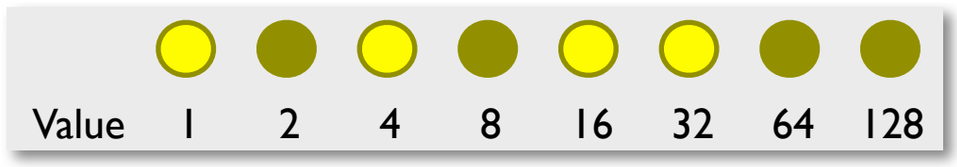


The Auxiliary LEDs are the 8 small LEDs above the top of the LED matrix display. They can display the binary equivalent of any number in the range 0 - 255.

Note that this function writes immediately to the Auxiliary LED portion of the Display Memory-- it does not wait for DisplaySlate, since Auxiliary LED data is not stored in the Game Slate.

Each LED represents a binary bit, the corresponding decimal values are shown here. The rightmost LED is the most significant bit, with value 128.

To find the right binary number to turn on (say) LEDs 1,3,5, and 6 (from the left) look at the binary values: 1+4+16+32=53, so use `SetAuxLEDs(53);`



Additional Examples:

`SetAuxLEDs(0);`	All aux LEDs off
`SetAuxLEDs(255);`	All aux LEDs on
`SetAuxLEDs(1);`	Leftmost aux LED on
`SetAuxLEDs(128);`	Rightmost aux LED on

There is also a separate function `SetAuxLEDsBinary(value)`, which reverses the bit order so that you can set the auxiliary LED state with a *binary constant*:

`SetAuxLEDsBinary(B00001111);` →

`SetAuxLEDsBinary(B10101100);` →

5. *ReadPx* :: Read the color of a pixel in the Game Slate.

Syntax:

```
byte ReadPx(x,y)
```

The ReadPx function returns the value– of type 'byte' – of the color name stored at position (x,y) in the Game Slate.

The x and y inputs are 8-bit unsigned integers (type 'byte'), and should each only be given numbers or variables that will be in the range 0 to 7. (Eight possible values for eight rows and eight columns.)

Example 1:

```
byte i;

i = ReadPx(3,5);
```

First declare a variable called 'i'

After this line executes, 'i' contains a number representing the color of the pixel at x=3, y=5 in the Game Slate.

Example 2:

```
DrawPx(3,4,ReadPx(3,3));
```

Make the pixel at x=3, y=4 the same color as the pixel at x=3, y=3.

(The list of pre-defined colors is described earlier on page 13.)

This function can be very handy for game mechanics-- if you draw all of the pixels for your environment first, then the color where your player pixel(s) will be drawn can give a quick way to tell if you've run into something:

Example 3:

```
if ( ReadPx(PlayerX,PlayerY) == Red )
   Alive = 0;
```

(This supposes that you have variables already assigned for PlayerX, PlayerY, and Alive.)

The double equals ('==') is used for testing equivalency. Again, see the Arduino language reference for syntax on the use of variables and conditionals like the 'if' statement.

6. *EditColor* :: Configure custom pixel colors.

Syntax:
```
EditColor(ColorName, R, G, B);
```

The four arguments to the function are each unsigned 8-bit integers (type 'byte'). The color name should be one of the pre-defined color names (which is internally resolved to a number).

The three color components R, G, and B, should each be a number in the range 0 to 15, giving 16 levels of shading for each color component.

Example usage:
```
EditColor(CustomColor8, 5, 15, 1);
```
Configures CustomColor8 with R=5, G=15, B=1.

Any of the pre-defined colors can be edited. This function also allows you to use the ten "custom color" placeholders to define new colors.

Note 1: The color balance of the LED display can be lopsided. This function allows you to adjust the actual colors drawn under names like 'White' or 'Violet' to better suit your taste. Achieving true color reproduction is not necessarily possible or easy-- Meggy Jr is designed to display cheerfully colored pixels, not video.

Note 2: The color definitions that will actually be drawn on the LED display are the ones that are in effect at the times that *DisplaySlate* is executed. Thus, Example 2 on the right will draw a *blue* dot on the screen.

This is even true after running *DisplaySlate*, since that function does not affect the Game Slate. Example 3 draws a pixel first as blue and then as green, while only using *DrawPx* once.

Example 2:

```
DrawPx(4,3,Red);
EditColor(Red, 0, 0, 7);
DisplaySlate();
```
Draws a *blue* dot.

Example 3:

```
EditColor(CustomColor7, 0, 0, 7);
DrawPx(2,2, CustomColor7);
DisplaySlate();
delay(100);
EditColor(CustomColor7, 0, 15, 0);
DisplaySlate();
```
Draws a blue dot, then changes the color of the dot to green.

7. *CheckButtonsDown* :: Check to see which buttons are currently pressed down.

Example usage: `CheckButtonsDown();`

Executing this command sets six variables that tell you about the buttons:

```
Button_A      Button_Down
Button_B      Button_Left
Button_Up     Button_Right
```

Buttons A and B are the two round ones on the right side of the LED display. Buttons Up, Down, Left, and Right are the "arrow key" buttons on the left side of the LED display.

If one of the six buttons was down when you checked, the corresponding variable will be nonzero. You can test this by using an "if(Button_X)" type statement. Note that this method only detects that buttons was *down* at the moment you checked. It makes no attempt to look for changes in the button state or look for key pressing events.

The example to the right, if placed in the loop portion of a sketch, will light up two different LEDs if button A or B is held down. The LEDs will stay lit up as long as the buttons are held down.

For a more detailed example, see the example sketch:
MeggyJr_CheckButtonsDown

```
ClearSlate();

CheckButtonsDown();

if (Button_A)
  DrawPx(6,4,Green);

if (Button_B)
  DrawPx(5,4,Green);

DisplaySlate();
```

8. *CheckButtonsPress* :: Check to see which buttons are pressed that weren't, last we checked.

Example usage: `CheckButtonsPress();`

Executing this command sets six variables that tell you about the buttons:

```
Button_A      Button_Down
Button_B      Button_Left
Button_Up     Button_Right
```

Buttons A and B are the two round ones on the right side of the LED display. Buttons Up, Down, Left, and Right are the "arrow key" buttons on the left side of the LED display.

If one of the six buttons was down when you checked, and *was not down* the last time that you checked, the corresponding variable will be nonzero. You can test this by using an "if(Button_X)" type statement. Note that this method only detects buttons that changed and were down when you looked.

The example to the right, if placed in the loop portion of a sketch, will light up two different LEDs if button A or B is pressed. No matter how long you hold down the buttons, the corresponding LEDs will only stay on for 30 milliseconds– it only detects that a button was pressed, not that it's held down.

This method of detecting button presses is suitable for most video games (and similar applications), and is used in some of the example sketches, such as Froggy Jr and MeggyBrite.

For a more detailed example, see also the example sketch:
MeggyJr_CheckButtonsPress

```
CheckButtonsPress();
ClearSlate();

if (Button_A)
 DrawPx(6,4,Blue);
if (Button_B)
 DrawPx(5,4,Blue);


DisplaySlate();
delay(30);
```

Note: Mechanical buttons can occasionally produce a "bounce"-- more than one transition when a button is pressed or released. This routine makes no attempt to "debounce." However, it works well in practice because you only check occasionally to see if there has been a change. This is in contrast to situations where bounce is a big problem: when events are directly triggered electrical signals from buttons.

9. *Tone_Start* :: Begin sound output at a given frequency

Syntax: `Tone_Start(Divisor, Duration)`

The two arguments to the function are unsigned 16-bit integers (type 'unsigned int').

The frequency output is 8 MHz/Divisor.
Duration is specified in (approximately) milliseconds.

Example usage: `Tone_Start(18182, 50);`

Begin sound output at frequency 8 MHz/18182 = 440 Hz, which is an '*A4*' note, scheduled to last for roughly 50 ms.

Tone_Start only starts sound output, scheduled to last for a given duration of time. A separate routine, built into the video refresh routine, automatically stops the tone if the scheduled time has elapsed. If you are already playing a tone and call *Tone_Start* again, the old tone will stop immediately and be replaced by the new call-- starting at the new frequency and lasting for the new duration, starting at the moment of the new call.

While a note is still playing, the read-only variable 'MakingSound' is nonzero; you can use that fact to detect when the sound finishes for multi-note sound effects.

A number of frequency divisors are predefined:

```
ToneC3, ToneCs3, ToneD3, ToneDs3, ToneE3, ToneF3,
ToneFs3, ToneG3, ToneGs3, ToneA3, ToneAs3, ToneB3,
```

where "Fs" stands for *F#*, and so forth. Notes are defined in the range ToneB2 (124 Hz) to ToneDs9 (9963 Hz).

Example 2 begins a tone of *F#* (Octave 3), for a duration of 100 ms.
The actual frequency output is 8 MHz/43243 = 185 Hz.

Example 2: `Tone_Start(ToneFs3, 100);`

Upgrading note: In library versions before 1.3, the function call SoundCheck() should be also added to the main loop of the program and checked often (e.g., at the same frequency of the buttons) to check and see if it's time to stop playing a tone.

A number of example programs are available in the Meggy Jr RGB Arduino Library.

You can load find them listed in your menu, by navigating to:
File>Sketchbook>Examples>Library-MeggyJr>

Some of these are "real" programs, showing off the capabilities of the hardware:

MeggyJr_Attack :: Attack of the Cherry Tomatoes

(The game that comes on the Meggy Jr RGB). Move your fighter up and down and fire at the ever advancing army of cherry tomatoes. Stop them--at all costs-- before they splat against your wall. You have an infinite number of Blueberry Bullets ('A' button), and a limited number of bombs and laser shots.

You start out the game with five bombs (Left arrow) and six laser shots ('B' button). Lasers are super bullets that destroy everything in your line of fire. Bombs destroy all the Cherry Tomatoes presently on the screen. If you use all of your laser shots, you can take take the power cells out of a remaining bomb to power five more shots.

The Cherry Tomatoes come at you in waves of 75, increasing in speed and density. For each wave you survive you get an extra bomb (up to 8 max), and the number of bombs is always shown on the auxiliary LED display at the top of the screen. If things get dull, you can boost yourself forward (right arrow). When things get tight, you can zoom up and down between shots by holding the up or down arrow buttons.

(Note: this program was written before the simplified library.)

MeggyJr_FroggyJr :: Froggy Jr

Why did the green pixel cross the road... and then the river?
And how do the logs drifting in the river go both ways?
Regardless, you've got to get your young froglets to the other side, without drowning or becoming road kill. (Good luck.)
Navigate with the arrow keys. Things get faster as you go along.

MeggyJr_MeggyBrite

A pixel art drawing program. Move your cursor with the arrow keys. The right-most button is draw/erase and the remaining button changes colors. When you change colors, the auxiliary LEDs change to indicate which one you are on. There are two other "color" modes too: an erase-only mode and a cursor-off display mode.

MeggyJr_RandomColors

Draw randomly colored dots on the LED screen. Slightly interactive; change the speed or number of colors. Pause, resume. Demonstrates using buttons, drawing dots, loop structures.

Many other programs for the Meggy Jr RGB are linked to from the Meggy Jr Link wiki page; you can get to that page from the main Meggy Jr RGB project page, http://www.evilmadscientist.com/go/meggyjr

Some of the other programs are just simple programming demonstrations-- good starting points for learning about the different library functions.

MeggyJr_Blink

Blink an LED pixel. Shows off basic drawing and displaying. It's the example from P. 7.

MeggyJr_CheckButtonsDown

Test your six buttons, demonstrate reading the button states.

MeggyJr_CheckButtonsPress

Demonstrate detecting individual button-press events

MeggyJr_CustomColors

Demonstrate configuring custom colors with the *EditColor* routine.

MeggyJr_EasyDrawDots

Draw a few colored dots on the screen.

MeggyJr_SetPxClr

Like MeggyJr_EasyDrawDots, but less easy. This demo is written with the Meggy Jr library, without the simplified functions. It demonstrates the use of a custom color look-up table.

There are also three sketches that demonstrate communication between Meggy Jr RGB and a host computer over the serial link. These are located in the *SerialCommunication* subdirectory Library: File>Sketchbook>Examples>Library-MeggyJr>SerialCommunication

MeggyJr_SerialMonitor

This program shows the process of sending text from the Meggy Jr to your computer and reading it out from within the Arduino environment. See the next page of this guide for more about serial monitoring.

The other two serial apps are designed for communicating with your computer when it is running sketches (programs) in the Processing Development Environment, http://processing.org/

MeggyJr_BinaryClock,

A full-fledged binary clock application, where you can set the time from the Meggy Buttons. You can *also* run the included processing sketch to set the Meggy Jr clock to your computer time.

MeggyJr_RemoteDraw

A Meggy Jr program to allow a remote user to draw dots to the screen over the serial interface. The included Processing sketch draws random dots *from your computer* to your Meggy Jr RGB.

Where to go from here?

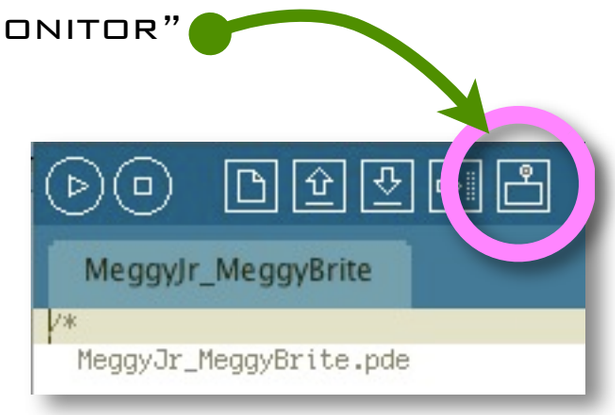
To go forward, try out these various demo programs and look through their code. Start small, by modifying these programs to make them still work but do different things. Once you understand that basic process-- modifying code without breaking it-- you're good to go!

If you encounter difficulty with Meggy Jr RGB in hardware, software, or elsewhere, odds are that somebody knows how to help you out. Your first stop should be the Evil Mad Scientist Laboratories forums:
http://www.evilmadscientist.com/forum/

Advanced Topic: Serial monitoring

Meggy Jr RGB is normally hooked up to a computer through the USB-TTL cable for programming. However, that cable can also be used for other types of serial communication between Meggy Jr RGB and the host computer.

After you've finished uploading your program, you can-- if you like-- leave the cable connected to your computer. You can then use any serial communication program to interact with the Meggy Jr RGB. One handy application of this is to use the Serial Monitor function of the Arduino development environment for debugging purposes.



To make your Meggy output serial data, first add the line *Serial.begin(*baud rate*)*, to the setup section of your sketch.

Using the *Serial.println* routine, you can output text or the value of variables one line at a time, as shown in the example to the right. (There is also a *Serial.print* routine, without the return at the end of each line.)

To see the serial output on your computer, click the "Serial Monitor" button in the Arduino environment; it's the one next to the "Upload to I/O board" button. Also, make sure that you have the correct baud rate selected-- the baud rate selection should be visible once you have the serial monitor on.

For additional information, see LadyAda's tutorial:
http://www.ladyada.net/learn/arduino/lesson4.html

and the Arduino reference on Serial.println:
http://arduino.cc/en/Serial/Println

```
#include <MeggyJrSimple.h>

void setup()
{
  MeggyJrSimpleSetup();
  Serial.begin(9600);
}

void loop()
{
  byte b;
  Serial.println("Hello world!");
  delay(1000);
  b = 5;
  Serial.println(b);
  delay(1000);
}
```

Add this line to initialize the Serial routine at 9600 baud. 19200 baud is also a good choice.

Print a line of text followed by a return.

Print the value of variable 'b' on its own line.

See also the MeggyJr_SerialMonitor example sketch.

Advanced Topic: The underlying Meggy Jr Library - "low level" API

The Meggy Jr library can be used on its own, without the simplified functions, in case you want to work without the Game Slate, for example. For a quick start, see the example program MeggyJr_SetPxClr. What follows here are some quick notes on the library, intended for folks who know their way around C++.

The first step is to create an instance of the MeggyJr, and initialize the hardware, like so:

```
MeggyJr Meg;

void setup()

{
   Meg = MeggyJr();
}
```

You can use a different name instead of Meg-- that's just our habit.

There are a number of useful structures and functions:

```
MeggyFrame[];
AuxLEDs;
ClearMeggy (void);
ClearPixel(byte x, byte y);
GetButtons(void);
GetPixelR(byte x, byte y);
GetPixelG(byte x, byte y);
GetPixelB(byte x, byte y);
SetPxClr(byte x, byte y, byte rgb[3]);
SoundState(byte t);
StartTone(unsigned int Tone, unsigned int duration);
SoundCheck(void) ;
```

You may note that some of these are nearly identical to versions in the simplified library. To see what these do and look at their code, open up MeggyJr.cpp.

MeggyFrame is the name of the 1-D display memory array. Each of the 192 bytes in the array stores a number from 0 to 15 representing the brightness of that particular LED element. It's easiest to understand the structure by looking how we set the color of a pixel in the array, like in the definition of the function SetPxClr:

```
void MeggyJr::SetPxClr(byte x, byte y, byte *rgb)
{
byte PixelPtr =  24*x + y;
MeggyFrame[PixelPtr] = rgb[2];
PixelPtr += 8;
MeggyFrame[PixelPtr] = rgb[1];
PixelPtr += 8;
MeggyFrame[PixelPtr] = rgb[0];
}
```

Note: Since only values 0-15 are used, it's actually possible to use the upper 4 bits of each byte for your own storage, should that become handy.