

NICK VEITCH

ARDUINO HARDWARE ENABLEMENT

Plug an OLED display into your Arduino, script a driver and learn C++ library programming, all without requiring any experience.

WHY DO THIS?

- Learn how to plug awesome displays into cheap hardware and code your own interface using the Arduino IDE and a smattering of learner-level C++.

This tutorial exists because I am a lazy miser. These are two undersung qualities, which in my opinion make the best engineers, but maybe I am biased as well as lazy and a miser. In any case, because of my character flaws, I have learnt and will now pass on to you some amazing things:

- How to use a cheap OLED display with an Arduino.
- How to create and package an Arduino library.
- How to save precious dynamic memory.
- How to save some I2C pins.
- How to do your own hardware enablement projects.
- How to convert bitmap images into code.

So, everyone's a winner. Except people who make expensive OLED displays. This is going to be the first of a two-part tutorial where we learn to tame both displays and the Arduino by writing our own code and building our own libraries. This month we're going to work out how to communicate with the screen and create a library that encapsulates those commands into a library before next month, creating lots of pretty works and pictures with our library.

Cautionary beginnings

OLED displays can easily be powered long-term by batteries, giving you freedom to use the serial port connection to find out what is going on inside its tiny mind. I wanted to build a device with a display that would log and tell me the temperature in the water tank in my attic (relayed by another Arduino), because I am too lazy to go up to the attic to find out. And that is when I happened on a cheap supply of OLED displays online, which were selling for \$3 a unit instead of the \$15 to \$20 I was used to.

When they arrived I discovered that these displays differed significantly enough from the "standard" ones that no existing library would work with them. I

could have adapted an existing library, but there were some other implementation issues. Cheapskatiness trumped laziness and I decided to write my own hardware enablement. Hurrah!

I (2) See

The I2C interface is a fairly standard way of connecting microcontrollers to things. There are several slightly different ways of doing it, but a common way is to use two wires: a clock (SCL) and a dataline (SDA). By cunning signalling, both can be used to signify the beginning and end of transmissions too. The Arduino has a library (the *Wire* library) to take care of this for you, but it uses hardwired pins. The trouble is that, although you can have multiple devices on the I2C bus, there are limits, and if you want to drive more than one display, you will find they have a very limited range of addresses. For these reasons, I decided to implement the protocol in my library so I could use any pins I liked, and also to reduce the overhead on having another library (*Wire* isn't that big, but it does contain a lot of stuff we won't need). For the master device (the Arduino in our case), transmission of data goes like this:

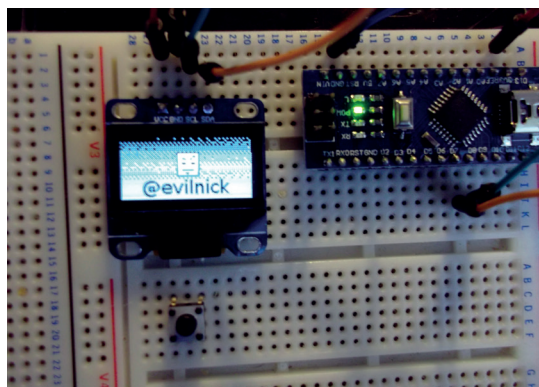
- Bring SCL and SDA HIGH.
- Bring SDA and SCL LOW.
- Load bit into SDA.
- Pulse SCL (High then Low).
- ... continue transmitting bits until end of byte.
- Send control bit (SDA HIGH) and signal end of transmission.
- Bring SCL and SDA LOW.
- Bring SCL and SDA HIGH.

The actual bits you are transmitting all follow this procedure, so we can start off by writing the low-level bits and then write higher level functions to send bytes and commands etc.

Initially, it is useful to write this code directly into an Arduino sketch – it keeps everything in one place and makes it a bit easier to test. So, our Arduino functions would look something like this:

```
void i2cStart()
{
    digitalWrite(dSCL, HIGH);
    digitalWrite(dSDA, HIGH);
    digitalWrite(dSDA, LOW);
    digitalWrite(dSCL, LOW);
}

void i2cStop()
```



You too can delight your friends and confound your enemies by writing custom drivers for cheap displays!

```
{
  digitalWrite(dSCL, LOW);
  digitalWrite(dSDA, LOW);
  digitalWrite(dSCL, HIGH);
  digitalWrite(dSDA, HIGH);
}

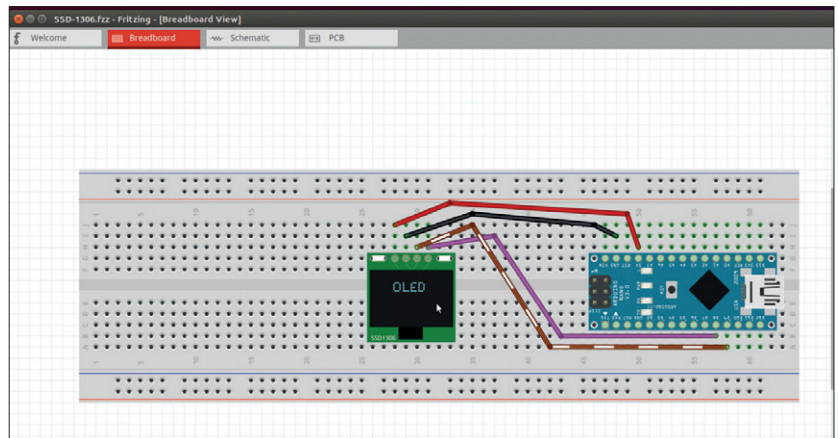
void sendByte(unsigned char b)
{
  char i;
  for(i=0;i<8;i++)
  {
    if((b << i) & 0x80){
      digitalWrite(dSDA, HIGH);
    }else{
      digitalWrite(dSDA, LOW);
    }
    digitalWrite(dSCL, HIGH);
    digitalWrite(dSCL, LOW);
  }
  digitalWrite(dSDA, HIGH);
  digitalWrite(dSCL, HIGH);
  digitalWrite(dSCL, LOW);
}
```

This presupposes that in the main code somewhere we define the pins (here called dSCL and dSDA) and set them as outputs. The loop in the **sendByte()** function merely uses a bitshift operator, **<<**, to iterate through the bits in the byte supplied to the function, and transmit them one at a time.

Now, to go any further than this, we need some specifics about the device we are communicating to. In this case, the OLED display uses the common SSD1306 control chip. This is a multi-protocol chip, though our hardware is hardwired to only supply a write-only I2C interface. The address of the device on the i2c bus is either 0x78 or 0x7A, which is set via connecting a pin on the SSD1306 device. Since these displays usually come on a board, you may have a jumper (with the cheap hardware I have, it is hardwired to be 0x78).

The address is important, as you need to signal this on the I2C bus to get a device to listen (remember, the bus is designed to have potential for more than one occupant). You will also need to know what commands you can and should send. For this, we need to find the datasheet for the SSD1306. A quick Google search should bring up some candidates, or you can request one direct from the manufacturers. A lot of the datasheet is not relevant to us because we will be using only one of the connection modes. What is highly relevant are the setup commands though – we need to send these to get the display to turn on and work correctly. To send a command we have to initiate the bus, send the slave address, send the control byte (telling the device we are writing to it), then send the actual command byte and close the bus again. We can wrap this up in a higher level function like this:

```
void sendCmd(unsigned char cmd)
{
```



The breadboard view shows why an Arduino Nano is really very very useful for working on this sort of project – it just slots right in.

```
i2cStart();
sendByte(0x78); //Slave address,SA0=0
sendByte(0x00); //write command
sendByte(cmd);
i2cStop();
}
```

The actual list of commands to initiate the display is long and consists of things we don't need to know much about (the slew rate seems to be a function of the size of the display, and there are various different ways of addressing the memory). For the moment we can just make an array out of the commands:

```
char init_codes[] {
  0xAE,0x00,0x10,0x40,0xB0,0x81,0xCF,0xA1,
  0xA6,0xA8,0x3F,0xC8,0xD3,0x00,0xD5,0x80,
  0xD9,0xF1,0xDA,0x12,0xDB,0x40,0x8D,0x14,
  0xAF
};
```

which we would declare in the main loop, and then create a function to loop through sending these when we want to initialise the display:

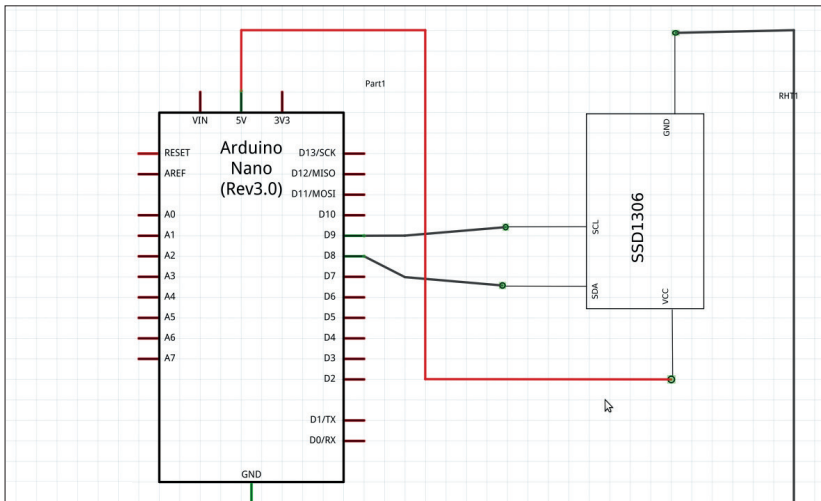
```
void init()
{
  for (i = 0; i < 25; i++) {
    sendCmd(init_codes[i])
  }
}
```

We now have enough code to bring up the display. But how will we even know that it is on? We need to stick something on it.

Addressing

The SSD1306 has three different modes of addressing the display – a horizontal mode, a vertical mode and a paged mode. The 'pages' are basically lines 8-bits deep across the display, which will be very useful for when we want to display characters. However, the horizontal mode will be useful for things like blanking the display.

Horizontal mode, it turns out, doesn't mean what you think it does. Each byte you send still corresponds to a vertical slice of 8 pixels, it just means that when you get to the end, the address pointer is updated to the beginning of the next row (see diagram). This means though that we can write 1024 bytes in



The fritzing diagram showing the connections for an Arduino Nano. It is straightforward though – just direct connections for the power and the two pins we use for I2C (in this case D8 and D9).

sequence to fill up the whole screen, which is perfect for a `cls()` type routine.

To do this we need to:

- Send the commands to initiate horizontal mode.
- Initiate the data connection.
- Send 1024 bytes.
- Close the connection.

For now, we can just build this in the main loop of the Arduino code

```
void loop() {
char init_codes[] {
0xAE,0x00,0x10,0x40,0xB0,0x81,0xCF,0xA1,
0xA6,0xA8,0x3F,0xC8,0xD3,0x00,0xD5,0x80,
0xD9,0xF1,0xDA,0x12,0xDB,0x40,0x8D,0x14,
0xAF
};
init();
sendCmd(0x20); // send the command to initiate horizontal
mode
sendCmd(0x00);
i2cStart();
sendByte(0x78); // identify the slave device
sendByte(0x40); // signal that what follows is data rather than
commands
for (unsigned int n=0;n<512;n++)
{
sendByte(0xAA);
sendByte(0x55);
}
i2cStop();
delay(1000);
}
```

This is a useful way to prototype functions that you may wish to develop – just bash them out in the main code, experiment with rationalisations and shortcuts, and then encapsulate them into a function.

Here we have initialised the display as discussed before. Then we send the commands to the device to set it into the horizontal addressing mode. To send the stream of data, we initiate the I2C connection and send the identifier byte (to address the correct device), and indicate that we are writing data to the display memory (0x40). Then we just send the bytes.

In this case, a chequerboard pattern). If you keep writing after 1024 bytes, the address counters on the SSD1306 will just reset and you will end up writing at the beginning again. Now that we have verifiable code that can be proved to work with the display, it is time to look into libraries.

Oh, I C++

Arduino libraries are written in C++. This may seem surprising and daunting to some, but it shouldn't be – the Arduino code you're used to writing is basically C++, albeit a version that hides a lot of the tricky stuff and wraps everything else in a layer of simplification. The point is, that for the most part, the actual code writing part should feel familiar; it's just the structures surrounding it which will be new to some.

The simplest library consists of just two files. There will be the `.cpp` file which contains the code itself, and a `.h` or header file. To start with, we need to create a directory in the place where user libraries live. This will be (on nearly all Linux distros) in the path `~/Arduino/libraries`. The only exception is if you are in the habit of running the Arduino IDE as root, which is very naughty! The reason some people do this is because then you don't have to change permissions for some of the devices used; consider instead following the Linux instructions here: <http://playground.arduino.cc/Linux/All>.

In the `libraries` directory, simply make a new directory (I called mine `evilOLED`), then we can create some files and directories

```
$ tree
```

```
.
├── evilOLED.cpp
├── evilOLED.h
├── examples
└── utils
```

The directories (`utils` and `examples`) we can forget about for now. The first thing to do is open up your favourite code editor and start editing the `evilOLED.cpp` file. The very first thing we need to put in the file is the `include` line, which adds this file's own header:

```
#include "evilOLED.h"
```

The next important order of business is to create a class (skip this paragraph if you already know what that is!). A class is really like a special datatype. Think of it like this: instead of defining an integer or a string, we are going to define a display. Along with that we have to provide the code for all of its interactions in the form of functions. We also have to allocate space for any variables and data that instances of the class will need when they are created. The class isn't an instance, it is the recipe for creating one, in almost the same way that the Arduino code knows that an `int` is an integer, and what to do when creating one, or adding or subtracting or printing one.

Our class is defined like this:

```
evilOLED::evilOLED(char sda, char scl)
{
_sda = sda;
```

```
_scl = scl;  
_col = 0x00;  
_row = 0x00;  
init();  
cls(0xff);  
}
```

We can explain this a little bit. The **constructor** is the first line. This is like a special function (with no return type at all) which is called whenever an instance of the class is first created. In it, we want to put definitions for any special data we want the instance to have, and any functions it should call. The argument or parameters in the constructor are special bits of data that will be passed in by the code creating the instance. In this case, we want to specify which pins we are going to use for communication.

The bit inside the braces (curly brackets `{` and `}`) is the code that will run. The first two bits may seem a little strange – we have taken the values passed in by the constructor and copied them to new variables (which haven't even got a type!). Then we set two more variables, and all of them begin with an underscore. The underscore is the convention which means that these variables are 'private' to the class – that means that only functions belonging to that class can see them – they can't be read or changed by any code outside of this class. We actually explicitly declare this in the header, but we have more to do here before we get on to that. The last two lines call other functions of the class – we will have to transfer them from the Arduino code we wrote, which should be up by now on www.linuxvoice.com/code/lv012/arduino.

They function are practically unchanged from the Arduino code – they have just been updated to use our new private variables, plus the function definitions now begin with **evioLED::<name>**, which declares them as part of our class.

To complete this fairly minimal version of our library, we also need to generate the header file. This is also on www.linuxvoice.com/code/lv012/arduino. The opening `#ifndef` statement is a common convention which basically prevents the header file from being parsed twice, as may happen on larger projects where several code files may include it. The following block of code, up until the `#endif`, will not be processed more than once. For completeness we've include the Arduino library and the `pgmspace`. These, as it turns out, are not explicitly used by the header file,

The best Arduino?

This is a small aside about the Arduino Nano, which I have come to believe is the best model for prototyping on. The reason is simply that it comes on a board, ready to plug into a breadboard. All the pins are single pitch around the edge so it doesn't take up much room, and it still has the very useful programming header on it if you need/want to program it that way. Coincidentally, it is also quite cheap, thus fulfilling my pinchfist proclivities.


dataset.pdf - http://www.9PDF.com		9 COMMAND TABLE		91.62%							
Index		9 COMMAND TABLE									
1 GENERAL DESIG.	6	(DC=0, R=W=SW=0) & C=0 (D=1=0 unless specific setting is stated)									
2 FEATURES	6	Fundamental Command Table									
3 ORDERING INTR.	6	DC	0	1	0	0	0	0	0	0	Command
4 ORDERING INTR.	6	R	1	1	0	0	0	0	0	0	Description
5 ORDERING INTR.	6	W	0	0	0	0	0	0	0	0	DC=0: Command indicates the start of 256
6 ORDERING INTR.	6	SW	0	0	0	0	0	0	0	0	bits. Command indicates the end of the
7 ORDERING INTR.	6	C	0	0	0	0	0	0	0	0	data. (SW=1: 7B1)
8 PIN ADDRESS	11	Set Command Control									
9 155013067R.	11	0	A	A	A	1	0	0	0	0	Enter Display ON
10 7PIN DESCRIPTION	13	0	A	A	A	1	0	0	0	0	1A, 1C, 1E: Reverse to RAM command display
11 9 FUNCTIONAL	15	0	A	A	A	1	0	0	0	0	(B=0)
12 8.1 MCU INTR.	16	0	A	A	A	1	0	0	0	0	1A, 1C, 1E: Normal display (1A, 1C)
13 8.1.1 MCU Per.	16	0	A	A	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
14 8.1.2 MCU Per.	16	0	A	A	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
15 8.1.3 MCU Ser.	17	0	A	A	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
16 8.1.4 MCU Ser.	17	0	A	A	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
17 7B1.8.1 MCU DC	19	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
18 8.1.5.1 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
19 8.1.5.2 Write	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
20 8.1.5.3 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
21 8.1.5.4 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
22 8.1.5.5 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
23 8.1.5.6 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
24 8.1.5.7 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
25 8.1.5.8 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
26 8.1.5.9 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
27 8.1.5.10 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
28 8.1.5.11 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
29 8.1.5.12 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
30 8.1.5.13 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
31 8.1.5.14 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
32 8.1.5.15 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
33 8.1.5.16 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
34 8.1.5.17 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
35 8.1.5.18 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
36 8.1.5.19 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
37 8.1.5.20 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
38 8.1.5.21 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
39 8.1.5.22 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
40 8.1.5.23 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
41 8.1.5.24 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
42 8.1.5.25 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
43 8.1.5.26 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
44 8.1.5.27 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
45 8.1.5.28 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
46 8.1.5.29 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
47 8.1.5.30 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
48 8.1.5.31 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
49 8.1.5.32 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
50 8.1.5.33 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
51 8.1.5.34 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
52 8.1.5.35 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
53 8.1.5.36 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
54 8.1.5.37 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
55 8.1.5.38 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
56 8.1.5.39 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
57 8.1.5.40 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
58 8.1.5.41 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
59 8.1.5.42 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
60 8.1.5.43 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
61 8.1.5.44 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
62 8.1.5.45 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
63 8.1.5.46 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
64 8.1.5.47 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
65 8.1.5.48 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
66 8.1.5.49 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
67 8.1.5.50 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
68 8.1.5.51 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
69 8.1.5.52 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
70 8.1.5.53 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
71 8.1.5.54 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
72 8.1.5.55 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
73 8.1.5.56 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
74 8.1.5.57 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
75 8.1.5.58 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
76 8.1.5.59 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
77 8.1.5.60 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
78 8.1.5.61 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
79 8.1.5.62 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
80 8.1.5.63 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
81 8.1.5.64 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
82 8.1.5.65 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
83 8.1.5.66 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
84 8.1.5.67 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
85 8.1.5.68 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
86 8.1.5.69 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
87 8.1.5.70 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
88 8.1.5.71 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
89 8.1.5.72 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
90 8.1.5.73 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
91 8.1.5.74 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
92 8.1.5.75 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
93 8.1.5.76 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
94 8.1.5.77 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
95 8.1.5.78 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
96 8.1.5.79 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
97 8.1.5.80 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
98 8.1.5.81 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
99 8.1.5.82 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
100 8.1.5.83 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
101 8.1.5.84 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
102 8.1.5.85 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
103 8.1.5.86 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
104 8.1.5.87 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
105 8.1.5.88 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
106 8.1.5.89 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
107 8.1.5.90 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
108 8.1.5.91 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
109 8.1.5.92 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
110 8.1.5.93 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
111 8.1.5.94 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
112 8.1.5.95 DC	20	0	A	E	A	1	0	0	0	0	1A, 1C: RAM-OUT to display panel
113 8.1.5.96											

but they are used in the main C++ file. It is a common convention to put all the necessary **includes** in one place, the header file, so the main code only has one **include** in it – it just means it is easier to track things down if you aren't searching two files for it.

We have also included a **#define** statement for the slave address of the display here. This is just an example of something you may want to do in your header file. As there is more than one possible value for this, it is probably more useful to have it as a variable rather than a compiled-in value, but it is also a useful reminder of what the default value actually is.

The class definition that follows is what is called a ‘prototype’. This outlines the parameters accepted and returned by all the member functions (including the constructor) as well as a list of the private data or functions used by the code. Basically, to add your function to the header, you can just copy and paste it, then edit out the **evioLED::** prefix. There is no functional code here, but this demonstrates how the code works, and if well commented, can tell you everything useful you need to know about the library.

The **examples** directory is where you will put any complete sample code using your library, which will then turn up in the relevant Arduino menu. It is a good idea to include as much, well commented, functionality as possible in these, as people tend not to read the instructions!

To make sure your library is usable by the Arduino IDE, you should put it where the rest of the libraries are. This is usually `~/arduino/libraries` but may depend on how and where you installed it. The foolproof way is to put the directory somewhere findable, then open up the IDE. Create a new sketch and then choose Sketch > Import Library > Add Library, and use the requester to specify the location of your library directory. You will find all the files are copied to wherever the library storage for Arduino happens to be. Note that you can continue to edit the library in situ – the code is recompiled each time you compile source that refers to it (uploading or checking sketches) so this can be a handy way to test changes. 

Nick Veitch has edited computer magazines for 1,000 years. He now works at Canonical and collects gin bottles.

The datasheet is useful, even though a lot of it doesn't apply. there is some info about other modes you may care to implement, such as hardware scrolling!